Джесс Либерти

Освой самостоятельно C++ за 21 день

В книге широко представлены возможности новейшей версии программного продукта Microsoft Visual C++. Подробно описаны средства и подходы программирования современных профессиональных приложений. Материалы книги дополнены многочисленными демонстрационными программами, в процессе разработки которых максимально используются возможности программных инструментов Microsoft Visual Studio. Особое внимание уделено новинкам версии 6.0 и новейшим технологиям объектно-ориентированного программирования, включая использование библиотеки MFC и шаблонов классов, а также создание связанных списков. Отдельное занятие посвящено вопросам объектно-ориентированного анализа и проектирования приложений. Подробно рассмотрены все средства и подходы конструирования собственных пользовательских классов.

Книга рассчитана на широкий круг читателей, интересующихся современными проблемами программирования.

Оглавление

Подробное содержание

Предметный указатель

Об авторе

Джесс Либерти (Jesse Liberty) — автор шести книг, посвященных языку C++ и объектно-ориентированному анализу и проектированию, а также постоянный автор журнала *C++ Report*. Он возглавляет компанию Liberty Associates, Inc. (`http://www.libertyassociates.com`), которая предоставляет услуги по обучению в Internet объектно-ориентированной разработке программных продуктов, а также занимается наставничеством, консультированием и контрактным программированием.

Джесс удостоился звания заслуженного программиста (Distinguished Software Engineer at AT&T) и работал в должности вице-президента отдела разработки Ситибанка (Citibank's Development Division). Он живет в окрестностях Кембриджа (шт. Массачусетс) с женой Стеси (Stacey) и дочерьми Робин (Robin) и Речел (Rachel). С ним можно связаться через Internet по адресу: `jliberty@libertyassociates.com`. Джесс осуществляет поддержку книги через свой же Web-узел по адресу: `http://www.libertyassociates.com` — щелкните на гиперссылке Books and Resources (Книги и ресурсы).

Посвящение

Немеркнущей памяти Давида Ливайна (David Levine) посвящается.

Благодарности

Третье издание этой книги — еще одна возможность выразить благодарность всем тем, без чьей поддержки и помощи написать ее в буквальном смысле было бы невозможно. Прежде всего эти слова вновь относятся к Стеси, Робин и Речел Либерти.

Хочу также поблагодарить моих редакторов в издательстве Sams, профессионалов высшего класса. Огромное спасибо Крису Денни (Chris Denny) и Брэду Джоунс (Brad Jones) за их работу над предыдущими изданиями книги. Особую признательность хочу выразить Трейси Данкельбергер (Tracy Dunkelberger), Холли Олендер (Holly Allender), Сину Диксону (Sean Dixon), Хезер Толбот (Heather Talbot), Барбаре Хече (Barbara Hacha) и Моне Браун (Mona Brown).

Кроме того, я очень благодарен Скипу Джилбреху (Skip Gilbrech) и Дэвиду Мак-Кьюну (David McCune), научившим меня программировать, а также Стиву Роджерсу (Steve Rogers) и Стефану Заджибойло (Stephen Zagieboylo), обучившим меня языку C++. Хотелось бы сказать спасибо многим читателям, которые помогли мне отыскать ошибки и описки в первых изданиях книги: Гордону Андерсону (Gordon Anderson), Ричарду Асчери (Richard Ascheri), Рону Барлоу (Ron Barlow), Эйрай Блэчер (Ari Blachor), Чарльзу Кампузано (Charles Campuzano), Тэмми Церконе (Tammy Cercone), Михаэлю Чомишевски (Michael Chomiczewski), Раймонду Чорчу (C. Raymond Church), Чарльзу Дешу (Charles I. Desch), Чарльзу Дингмену (Charles Dingman), Джону Эмбоу (John Embow), Джею Эриксону (Jay Erikson), **Марку Фидлеру** (Mark Fiedler), Адаму Фордайсу (Adam Fordyce), **Роберту Франсису** (Robert Francis), Кристоферу Гарьипи (Christopher Gariepy), Грегу Гордону (Greg Gordon), Чарльзу Хейзгеве (Charles Hasegawa), **Элиоту Кирсли** (Elliot Kearsley), Андре Р. Кинни (Andrew R. Kinnie), Лари Кирби (Lari Kirby), Джо Корти (Joe Korty), Ричарду Ли (Richard Lee), Роджеру Лейнингеру (Roger Leininger), Рубену Лопезу (Ruben Lopez), **Рэю Люшу** (Ray Lush), Франку Маррено (Frank Marrero), Джоан Мэтью (Joan Mathew), Джеймсу Мэтью (James Mayhew), Шерлиль Мак-Кена (Sheryl McKenna), Джудит Милз (Judith Mills), Терри Милнеру (Terry Milner), Патрику Муру (Patrick Moore), Крису Нили (Chris Neely), Гари Пейджу (Gary Page), Джеймсу Парсонзу (James Parsons), Нирелу Петелу (Neeral Patel), Раулю ван Пруйену (Raoul van Prooijen), Карену Ризеру (Karen Risser), Дэну Роджерсу (Dan Rogers), Ральфу Руссо (Ralph Russo), Грегори Саффорду (Gregory Safford), Джо Скелоуну (Joe Scalone), Роберту Сайксу (Robert Sikes), Сэму Сабо (Sam Suboh), Уолтеру Сану (Walter Sun), Полу Саттону (Paul Sutton), **Джеймсу Томпсону** (James Thompson), Орландо Ванину (Orlando Vanin), Паскалю Вердье (Pascal Verdieu), Стефану Вейну (Stephen Wain), Джерри Уайерсу (Jerry Wares), Джеку Уайту (Jack White), Скотту Вильсону (Scott Wilson), Нику Витхауз (Nick Witthaus), Микаэлле Зулли (Michelle Zulli) и особенно **Дональду Зи** (Donald Xie).

Наконец, хочу поблагодарить миссис Калиш (Mrs. Kalish), которая в 1965 году научила своих шестиклассников (в том числе и меня) операциям в двоичной арифметике: в то время ни она, ни я не знали, зачем это может понадобиться.

Введение

Цель этой книги — помочь читателю научиться программировать на языке C++. Всего за 21 день вы узнаете о таких необходимых для этого вещах, как управление вводом-выводом, циклы, массивы, объектно-ориентированное программирование, шаблоны и создание приложений на C++. Все темы поданы в хорошо организованных уроках, которые выстроены в порядке усложнения. Для иллюстрации рассматриваемых тем во все главы включены листинги программ, дополненные результатами работы этих программ и подробным анализом инструкций. Для удобства ссылки на инструкции при анализе программ все их строки пронумерованы.

Чтобы помочь вам быстрее усвоить наши уроки, в конце каждого из них представлена подборка часто задаваемых вопросов и ответы на них, а также тест для самоконтроля и упражнения. В правильности своих ответов на вопросы теста и упражнения вы сможете убедиться, заглянув в приложение Г.

Для кого написана эта книга

Чтобы, опираясь на материал этой книги, научиться программировать на языке C++, вам вовсе не обязательно иметь предыдущий опыт программирования. Изложение здесь начинается с исходной точки, и, работая с книгой, вы изучите не только сам язык, но и концепции, положенные в основу программирования на C++. Верным и надежным гидом на пути к знаниям будут многочисленные примеры синтаксиса и подробнейший анализ всех приведенных здесь программ. Начинаете ли вы с нуля или у вас уже есть некоторый опыт программирования — в любом случае четкая организация этой книги создаст все условия для быстрого и простого изучения языка C++.

Соглашения

Выделенная с помощью этой пиктограммы информация поможет сделать программирование на языке C++ более эффективным.

Вопросы и ответы

Для чего нужны эти "Вопросы и ответы"?

Когда начинающий программист берется за разработку своего первого приложения, у него наверняка возникнет масса вопросов. Чтобы предупредить их появление, мы приводим в конце каждой главы наиболее часто возникающие вопросы, ответы на которые помогут вам глубже проникнуть в суть изложенного и упростят использование того или иного средства программирования.

Эта пиктограмма сфокусирует ваше внимание на проблемах или побочных эффектах, которые могут возникнуть при определенных обстоятельствах.

Во врезках предлагаются четкие определения основополагающих терминов.

Рекомендуется

Используйте эти рекомендации для нахождения наиболее эффективных решений поставленных задач.

Не рекомендуется

Не пропускайте важные замечания и предупреждения, показанные в этом столбце.

В книге используются различные шрифты, чтобы помочь вам отличить код C++ от обычного языка. Для ключевых слов и выражений программных кодов на языке C++ используется `такой шрифт`. Если при описании синтаксиса выражений используются слова, которые нужно заменять в программах реальными переменными или параметрами, то эти замещаемые слова пишутся *`таким шрифтом`*. Новые и важные термины вводятся *курсивом*. Опции меню и диалоговых окон показаны таким шрифтом.

В листингах данной книги каждая строка программы имеет свой номер. Если вы видите в листинге строку без номера, знайте, что это продолжение предыдущей пронумерованной строки программы (некоторые программные строки слишком длинны для ширины страниц книги). В этом случае вам следует ввести две строки как одну, не разделяя их.

Основные вопросы

При подготовке к первой неделе изучения основ программирования на языке C++ вам понадобится компилятор, текстовый редактор и эта книга. Если у вас нет ни компилятора C++, ни текстового редактора, вы можете работать с этой книгой теоретически, но результат не будет таким высоким, как при выполнении всех предлагаемых здесь упражнений на компьютере.

Лучший способ научиться программировать — самому писать программы! В конце каждой главы вы найдете раздел практических занятий, который содержит вопросы для самопроверки и набор упражнений. Не пожалейте времени, чтобы ответить на все вопросы и выполнить все упражнения. Сверьте полученные результаты с правильными ответами, приведенными в приложении Г. Книга организована так, что последующие главы построены с учетом материала, изложенного в предыдущих занятиях, поэтому прежде чем двигаться вперед, убедитесь в том, что вы до конца понимаете уже прочитанный материал.

Несколько слов для программистов на языке C

Материал, изложенный в первых пяти занятиях, вероятно, будет вам знаком. Тем не менее вам стоит просмотреть содержимое этих занятий и выполнить упражнения, чтобы удостовериться в полном понимании основ, необходимых для усвоения более сложных тем. И только после этого переходите к чтению занятия 6.

Что дальше

В течение первой недели вам предстоит освоить материал, необходимый для первых шагов в программировании вообще и на языке C++ в частности. На первых двух занятиях вы ознакомитесь с базовыми концепциями программирования и с ходом выполнения программ. На занятии 3 вы получите представление о переменных и константах, а также о том, как использовать данные в программах. На занятии 4 рассматриваются возможные ветвления программ на основе используемых данных и условий, проверяемых во время работы программы. На занятии 5 вы узнаете о том, что представляют собой функции и как их использовать, а занятие 6 познакомит вас с классами и объектами. На занятии 7 вы получите более подробную информацию о ходе выполнения программ, а к концу первой недели сможете писать настоящие объектно-ориентированные программы.

Первые шаги

Введение

Добро пожаловать на страницы книги *Освой самостоятельно C++ за 21 день*! Предлагаю незамедлительно отправиться в путь, если вы хотите стать профессиональным программистом на языке C++. Сегодня вы узнаете:

- Почему C++ стал стандартом в области разработки программных продуктов
- Каковы этапы разработки программы на языке C++
- Как написать, скомпилировать и скомпоновать свою первую программу на языке C++

Краткий экскурс в историю языка C++

Эволюция языков программирования с момента появления первых электронных компьютеров, построенных для выполнения расчетов траектории движения артиллерийских снарядов во время второй мировой войны, была довольно драматической. Раньше программисты работали с самыми примитивными компьютерными командами, представлявшими собой часть машинного языка. Эти команды состояли из длинных строк единиц и нулей. Вскоре были изобретены ассемблеры, которые могли отображать машинные команды в мнемоническом представлении, более понятном для человека (например, команды `ADD` или `MOV`).

Со временем появились такие языки высокого уровня, как BASIC и COBOL. Благодаря этим языкам появилась возможность программировать, используя логические конструкции из слов и предложений, например `Let I = 100`. Эти команды переводились в машинный язык интерпретаторами и компиляторами. Интерпретатор по мере чтения программы последовательно превращает ее команды (или код) в команды машинного языка. Компилятор же целиком переводит программный код (листинг программы) в некоторую промежуточную форму — объектный файл. Этот этап называется компиляцией. Затем компилятор вызывает программу компоновки, которая превращает объектный файл в исполняемый файл программы.

С интерпретатором работать проще, так как команды программы выполняются в той последовательности, в которой они записаны, что облегчает контроль за выполнением программы. Компилятор же вносит дополнительные этапы компиляции и компоновки программы, в результате чего получается исполняемый файл, недоступный для анализа и редактирования. Однако скомпилированные программы выполняются быстрее, так как перевод команд программы на машинный язык уже произошел на этапе компиляции.

Еще одно преимущество компилируемых языков программирования, таких как C++, состоит в том, что полученные программы могут выполняться на компьютерах без компилятора. При работе же с интерпретируемыми языками для выполнения готовой программы нужно обязательно иметь соответствующую программу-интерпретатор.

В некоторых языках (например, Visual Basic) роль интерпретатора выполняет динамическая библиотека. Интерпретатором языка Java является виртуальная машина (Virtual Machine, или VM). В качестве виртуальной машины обычно используется броузер (такой как Internet Explorer или Netscape).

В течение многих лет основным достоинством программы считалась ее краткость и быстрота выполнения. Программу стремились сделать как можно меньше, поскольку память стоила весьма недешево, да и заинтересованность в высокой скорости выполнения объяснялась высокой стоимостью процессорного времени. Но по мере того как компьютеры становились все портативнее, дешевле (особенно ощутимо дешевела память) и быстрее, приоритеты менялись. Сегодня стоимость рабочего времени программиста намного превышает стоимость большинства компьютеров, используемых в бизнесе. Сейчас большим спросом пользуются профессионально написанные и легко эксплуатируемые программы. Простота эксплуатации означает, что при изменении требований, связанных с решением конкретных задач, программа легко перенастраивается без больших дополнительных затрат.

Программы

Слово *программа* используется в двух значениях: для обозначения отдельных блоков команд (или исходного кода), написанных программистом, и для обозначения исполняемого программного продукта как единого целого. Это различие в понятиях может ввести читателя в заблуждение, поэтому постараемся явно определять, что имеется в виду: исходный код или исполняемый продукт.

Итак, программу можно определить либо как набор написанных программистом команд, либо как выполняемый на компьютере продукт.

Исходный текст программы можно превратить в выполняемую программу двумя способами. В одном случае интерпретаторы переводят исходный код в машинные команды, и компьютер сразу же их выполняет. В качестве альтернативного варианта компиляторы переводят исходный код в исполняемый файл программы, который затем можно использовать самостоятельно. Хотя с интерпретаторами работать легче, большинство серьезных программ создается с использованием компиляторов, поскольку скомпилированный код выполняется намного быстрее. Примером компилируемого языка программирования служит C++.

Решение многих проблем

С течением времени проблемы, ставящиеся перед программистами, меняются. Двадцать лет назад программы создавались в основном для обработки больших объемов данных. При этом зачастую как те, кто писал программы, так и те, кто их использовал,

были профессионалами в компьютерной области знаний. Сегодня многое изменилось. С компьютером нередко работают те, кто даже понятия не имеет о его аппаратном и программном обеспечении. Компьютеры стали инструментом, который используется людьми, больше заинтересованными в решении своих деловых проблем, чем в глубоком освоении компьютера.

По иронии судьбы, чтобы облегчить новому поколению пользователей работу с программами, сложность самих этих программ значительно повысилась. Канули в лету те дни, когда пользователи вводили "таинственные знаки" (т.е. команды) в ответ на понятные только посвященным подсказки-приглашения, в результате получая поток "сырых", т.е. совершенно необработанных данных. В современных программах используются высокоорганизованные, дружественные по отношению к пользователю интерфейсы, оснащенные многочисленными окнами, меню, диалоговыми окнами и мириадами визуальных графических средств, с которыми все уже хорошо знакомы. Программы, написанные для поддержки этого нового уровня взаимодействия человека с компьютером, гораздо сложнее написанных всего лишь десять лет назад.

С развитием всемирной информационной сети Web для компьютеров началась новая эра проникновения на рынок. Пользователей компьютеров сейчас больше, чем когда бы то ни было, и при этом их претензии чрезвычайно высоки. Даже по прошествии всего нескольких лет с момента выхода первого издания этой книги программы заметно увеличились и усложнились, а необходимость использования методов объектно-ориентированного программирования для решения проблем, ставящихся перед современными программистами, стала просто очевидной.

С изменением требований к программированию претерпели изменение как языки, так и технология написания программ. Хотя в истории эволюции программирования есть много интересного, в этой книге мы остановимся на переходе от процедурного программирования к объектно-ориентированному.

Процедурное, структурированное и объектно-ориентированное программирование

До недавних пор программы рассматривались как последовательности процедур, выполнявших некоторые действия над данными. Процедура, или функция, представляет собой набор определенных команд, выполняемых друг за другом. Данные были отделены от процедур, и главным в программировании было проследить, какая функция какую вызывает и какие данные при этом меняются. Чтобы внести ясность в эту потенциально запутанную ситуацию, были разработаны принципы структурированного программирования.

Основная идея структурированного программирования вполне соответствует принципу "разделяй и властвуй". Компьютерную программу можно представить состоящей из набора задач. Любая задача, которая слишком сложна для простого описания, должна быть разделена на несколько более мелких составных задач, и это деление необходимо продолжать до тех пор, пока задачи не станут достаточно простыми для понимания.

В качестве примера возьмем вычисление средней заработной платы всех служащих компании. Это не такая уж простая задача. Однако ее можно разбить на ряд подзадач.

1. Выясняем, сколько зарабатывает каждый служащий.
2. Подсчитываем количество людей в компании.
3. Суммируем все зарплаты.
4. Делим суммарную зарплату на количество служащих в компании.

Суммирование зарплат тоже можно разделить на несколько этапов.

1. Читаем запись каждого служащего.
2. Получаем доступ к информации о зарплате.
3. Прибавляем очередное значение зарплаты к накопительной сумме.
4. Читаем запись следующего служащего.

В свою очередь, операцию чтения записи каждого служащего можно разбить на еще более мелкие подоперации.

1. Открываем файл служащих.
2. Переходим к нужной записи.
3. Считываем данные с диска.

Структурированное программирование остается довольно успешным способом решения сложных проблем. Однако к концу 1980-х годов слишком очевидными стали некоторые недостатки структурированного программирования.

Во-первых, не было реализовано естественное желание думать о данных (например, записях служащих) и действиях над ними (сортировке, редактировании и т.п.) как о едином целом. Процедурное программирование, наоборот, отделяло структуры данных от функций, которые манипулировали этими данными.

Во-вторых, программисты обнаружили, что они постоянно переизобретают новые решения старых проблем. Такая ситуация часто называется изобретением колеса (или велосипеда). Желание иметь возможность многократного использования рутинных блоков, повторяющихся во многих программах, вполне естественно. Это чем-то напоминает сборку приемника из радиодеталей. Конструктору не нужно каждый раз изобретать диоды и транзисторы. Он просто использует стандартные, заранее подготовленные радиодетали. Но для разработчиков программных продуктов такой возможности долгое время не было.

Внедрение в практику дружеского пользовательского интерфейса с рамочными окнами, меню и экранными кнопками определило новый подход к программированию. Программы стали выполняться не последовательно от начала до конца, а отдельными блоками в ответ на выполнение некоторого события. При возникновении определенного события (например, щелчка на кнопке или выбора из меню команды) программа должна отреагировать на него соответствующим образом. При таком подходе программы становятся все более интерактивными, что необходимо учитывать при их разработке.

НОВЫЙ ТЕРМИН

Работая с программами прошлого поколения, пользователь вынужден был проходить через последовательность экранов, чтобы добраться до нужного. В современных программах сразу предоставляется возможность выбора из всех предусмотренных разработчиком вариантов и обеспечивается соответствующая реакция на выбор пользователя.

Объектно-ориентированное программирование стремится отвечать этим требованиям, предоставляя технологию управления элементами любой сложности, создавая условия для многократного использования программных компонентов и объединяя данные с методами манипуляции ими.

Суть объектно-ориентированного программирования состоит в том, чтобы обращаться с данными и процедурами, которые выполняют действия над этими данными, как с единым объектом, т.е. самодостаточным элементом, который в чем-то идентичен другим таким же объектам, но в то же время отличается от них определенными уникальными свойствами.

Язык C++ и объектно-ориентированное программирование

В языке C++ полностью поддерживаются принципы объектно-ориентированного программирования, включая три кита, на которых оно стоит: инкапсуляцию, наследование и полиморфизм.

Инкапсуляция

Когда инженер использует в своей разработке резистор, он не изобретает его заново, а идет на склад (в магазин, к коллеге — возможны варианты) и в соответствии с заданными параметрами подбирает себе нужную деталь. При этом его не очень-то волнует, как устроен данный конкретный резистор, лишь бы он работал в соответствии с заводскими характеристиками.

Именно это свойство скрытости или автономности объектов, используемых во внешних конструкциях, называется *инкапсуляцией*. С помощью инкапсуляции можно обеспечить сокрытие данных. Это очень важная характеристика, благодаря которой пользователь может не задумываться о внутренней работе используемого объекта. Подобно тому как использование холодильника не требует знаний о принципах работы рефрижератора, применение хорошо разработанного программного объекта позволяет не заботиться о взаимоотношениях его внутренних переменных-членов.

Еще раз приведем аналогию: для эффективного использования резистора инженеру совсем не обязательно знать принципы его работы и внутреннее устройство. Все свойства резистора инкапсулированы (т.е. скрыты) в самом резисторе, важно только, чтобы он правильно справлялся со своими функциями.

В языке C++ свойство инкапсуляции поддерживается посредством создания нестандартных (пользовательских) типов данных, называемых классами. О том, как создаются классы, вы узнаете на занятии 6. После создания хорошо определенный класс действует как полностью инкапсулированный объект, т.е. его можно использовать в качестве целого программного модуля. Настоящая же внутренняя работа класса должна быть скрыта. Пользователям хорошо определенного класса не нужно знать, как этот класс работает; им нужно знать только одно — как его использовать.

Наследование и многократное использование

Когда инженеры из компании Acme Motors решили сконструировать новый автомобиль, у них было два варианта: они могли начать с нуля или модифицировать существующую модель Star. Возможно, эта модель почти идеальна, но хотелось бы добавить турбокомпрессор и шестискоростную передачу. Главный инженер выбрал второй вариант, т.е. не начинать с нуля, а построить другую модель автомобиля Star, усовершенствовав ее за счет дополнительных возможностей. При этом он предложил назвать новую модель Quasar. Quasar — это разновидность модели Star, но оснащенная новыми деталями.

Язык C++ поддерживает наследование. Это значит, что можно объявить новый тип данных (класс), который является расширением существующего. Об этом новом подклассе говорят, что он унаследован от существующего класса, и называют его производным. Модель Quasar произведена от модели Star, и поэтому она наследует все ее качества, но при необходимости может их расширить. О наследовании и его применении в языке C++ речь пойдет на занятиях 11 и 15.

Полиморфизм

Новая модель Quasar, в отличие от Star, может по-другому реагировать на нажатие акселератора. В модели Quasar можно добавить инжекторную систему впрыскивания топлива в двигатель и турбокомпрессор вместо использования карбюратора в модели Star. Однако пользователю не обязательно знать об этих отличиях. Он может просто надавить на газ и ожидать соответствующей реакции автомобиля, за рулем которого он сидит.

Язык C++ поддерживает возможность вносить изменения в выполнение одноименных функций для разных объектов благодаря так называемому полиморфизму функций и классов. *Поли* означает много, *морфе* — форма, следовательно, полиморфизм означает многообразие форм. Подробно это понятие рассматривается на занятиях 10 и 13.

Эволюция языка C++

Когда назрела идея объектно-ориентированного анализа, проектирования и программирования, Бьярн Страуструп (Bjarne Stroustrup) взял язык C (наиболее популярный для разработки коммерческих программных продуктов) и расширил его, обогатив средствами, необходимыми для объектно-ориентированного программирования.

Хотя язык C++ справедливо называют продолжением C и любая работоспособная программа на языке C будет поддерживаться компилятором C++, при переходе от C к C++ был сделан весьма существенный скачок. Язык C++ выигрывал от своего родства с языком C в течение многих лет, поскольку программисты могли легко перейти от C к использованию C++. Однако многие программисты обнаружили, что для того, чтобы в полной мере воспользоваться преимуществами языка C++, им нужно отказаться от некоторых своих прежних знаний и приобрести новые, а именно: изучить новый способ концептуализации и решения проблем программирования.

Нужно ли сначала изучить язык C

У многих возникает вопрос: "Поскольку C++ является продолжением языка C, нужно ли сначала осваивать C?" Страуструп и большинство других программистов, использующих C++, считают, что это не только не нужно, но гораздо лучше этого вовсе не делать.

Эта книга не предполагает наличия у читателя предварительного опыта программирования. Но если вы знакомы с программированием на C, первые пять глав вам достаточно лишь просмотреть. Только начиная с занятия 6, мы приступим к настоящей разработке объектно-ориентированных программ.

C++ и Java

C++ в настоящее время считается господствующим языком, используемым для разработки коммерческих программных продуктов. В последние годы это господство слегка заколебалось благодаря аналогичным претензиям со стороны такого языка программирования, как Java, но маятник общественного мнения качнулся в другую сторону, и многие программисты, которые бросили C++ ради Java, в последнее время поспешили вернуться к своей прежней привязанности. В любом случае эти два языка так похожи, что изучив один из них, вы на 90% освоите другой.

Стандарт ANSI

Аккредитованный комитет стандартов (Accredited Standards Committee), действующий под руководством Американского национального института стандартов (American National Standards Institute — ANSI), создал международный стандарт для языка C++.

Стандарт C++ также именуется в настоящее время как ISO — International Standards Organization (Международная организация по стандартизации). Кроме того, когда говорят о стандарте языка C++, иногда имеют в виду или NCITS (National Committee for Information Technology Standards — Национальный комитет по стандартам на информационные технологии), или X3 (старое название комитета NCITS), или ANSI/ISO. В этой книге мы будем придерживаться стандарта ANSI, поскольку это наиболее популярный термин.

Аббревиатура ANSI обычно произносится как "анси".

Стандарт ANSI — это попытка гарантировать, что язык C++ будет аппаратно независимым (т.е. переносимым с компьютера на компьютер). Это значит, что программа, написанная в соответствии со стандартом ANSI для компилятора компании Microsoft, будет компилироваться без ошибок с использованием компилятора другого производителя. Более того, поскольку приведенные в этой книге программы являются ANSI-совместимыми, они должны компилироваться без ошибок на компьютерах, работающих на платформах Mac, Windows или Alpha.

Для большинства изучающих язык C++ стандарт ANSI остается прозрачным. Тем не менее соответствие программного продукта общепринятым стандартам ANSI важно для профессиональных программистов. Мы позаботились о том, чтобы все программы, вошедшие в эту книгу, были ANSI-совместимыми.

Подготовка к программированию

Язык C++, возможно, больше любого другого требует от программиста до написания программы провести подготовительный этап, заключающийся в ее проектировании. При решении тривиальных проблем, рассматриваемых в первых нескольких главах этой книги, можно обойтись и без затрат на проектирование. Однако сложные проблемы, с которыми профессиональные программисты сталкиваются в реальной жизни чуть ли не каждый день, действительно требуют предварительного проектирования, и чем тщательнее оно будет проведено, тем более вероятно, что программа сможет их решить, причем с минимальными затратами времени и денежных средств. При добросовестно проведенном проектировании создается программа, которую легко отладить и изменять в будущем. Было подсчитано, что около 90% стоимости программного продукта составляет стоимость отладки и настройки. Удачно выполненное проектирование может значительно уменьшить эти расходы, а значит, и стоимость проекта в целом.

Первый вопрос, который нужно задать при подготовке к проектированию любой программы, звучит примерно так: "Какую проблему я хочу решить?" Каждая программа должна иметь четкую, ясно сформулированную цель, и вы увидите, что это относится даже к простейшим программам, приведенным в этой книге.

Второй вопрос каждый уважающий себя программист поставит следующим образом: "Можно ли решить эту проблему с использованием уже имеющихся программных продуктов, т.е. не изобретая собственного колеса?" Может быть, для решения этой проблемы достаточно воспользоваться своей старой программой, ручкой и бумагой или купить у кого-то уже готовую программу? Часто такое решение может оказаться лучше, чем создание абсолютно новой программы. Программист, предлагающий такую альтернативу, никогда не пострадает от отсутствия работы: умение находить экономные решения проблем обеспечит ему популярность в будущем.

Уяснив проблему и придя к выводу, что она требует написания абсолютно новой программы, вы будете готовы к этапу проектирования.

Создание любого коммерческого приложения требует тщательного анализа проблемы и проектирования ее эффективного решения. Хотя эти этапы логически предваряют этап написания программы, все же лучше начать с изучения базового синтаксиса и семантики языка C++ еще до изучения методов формального анализа и проектирования.

Среда разработки

В этой книге предполагается, что в вашем компиляторе предусмотрен режим работы с командной для непосредственного ввода данных, минуя графический интерфейс таких систем, как Windows или Macintosh. Найдите опцию console или easy window либо обратитесь к документации, прилагаемой к компилятору.

Возможно, ваш компилятор имеет собственный встроенный текстовый редактор либо вы можете использовать любой коммерческий текстовый редактор, сохраняющий файлы в текстовом формате без атрибутов форматирования. Примерами таких редакторов могут служить Windows Notepad, команда DOS `Edit`, Brief, Epsilon, EMACS и vi. Такие коммерческие текстовые процессоры, как WordPerfect, Word и многие другие, также позволяют сохранять файлы в текстовом формате.

Файлы, создаваемые с помощью текстовых редакторов, называются файлами источников. Они обычно имеют расширение `.cpp`, `.cp` или `.c`. В этой книге файлы, содержащие листинги программ, имеют расширение `.cpp`, но всегда лучше просмотреть документацию компилятора, с которым вы собираетесь работать, и выяснить его предпочтения.

Для большинства компиляторов C++ неважно, какое расширение имеет файл, содержащий исходный текст программы, хотя многие из них по умолчанию используют расширение `.cpp`. Однако будьте внимательны: некоторые компиляторы рассматривают файлы с расширением `.c` как программы на языке C, а файлы с расширением `.cpp` как программы на языке C++. Так что работу с компилятором всегда лучше начать с чтения документации.

Компиляция исходного кода программы

Хотя исходный текст программы, содержащийся в вашем файле, не будет понятен каждому, кто в него заглянет (особенно тем, кто незнаком с языком C++), все же он представлен в таком виде, который может быть воспринят человеком. Файл с исходным текстом программы — это еще не программа, и его нельзя выполнить или запустить.

Рекомендуется	Не рекомендуется
Используйте для написания исходного текста программы простой текстовый редактор или редактор, встроенный в компилятор. **Сохраняйте** свои файлы с расширением `.cpp`, `.cp` или `.c`. **Обращайтесь** к документации компилятора и компоновщика, чтобы быть уверенным в правильном компилировании и компоновке программы.	**Не используйте** текстовый процессор, который сохраняет форматированный текст. Если вам все-таки приходится обращаться к нему, сохраняйте файлы как текст ASCII.

Чтобы превратить исходный текст в программу, используется *компилятор*. Каким образом вызвать компилятор и как сообщить ему о местонахождении исходного текста программы, зависит от конкретного компилятора, поэтому вновь нужно заглянуть в документацию.

После завершения компиляции исходного кода создается объектный файл. Этот файл обычно имеет расширение `.obj`. Но это еще не выполняемая программа. Для превращения объектного файла в исполняемый нужно запустить программу компоновки.

Создание исполняемого файла с помощью компоновщика

Программы на языке C++ обычно создаются путем компоновки одного или нескольких объектных файлов (файлов `.obj`) с одной или несколькими библиотеками. *Библиотекой* называется коллекция компонуемых файлов, которые либо поставляются вместе с компилятором, либо приобретаются отдельно, либо создаются и компилируются самим программистом. Все компиляторы C++ поставляются с библиотекой функций (или процедур) и классов, которые можно включить в программу. *Функция* — это программный блок, который выполняет некоторые служебные действия, например складывает два числа или выводит информацию на экран. *Класс* можно рассматривать как коллекцию данных и связанных с ними функций. О функциях и классах речь впереди (см. занятия 5 и 6).

Итак, чтобы создать исполняемый файл, нужно выполнить перечисленные ниже действия.

1. Создать файл с исходным текстом программы, который будет иметь расширение `.cpp`.
2. Скомпилировать исходный код и получить объектный файл с расширением `.obj`.
3. Скомпоновать файл `.obj` с необходимыми библиотеками с целью создания исполняемого файла программы.

Цикл разработки

Если бы каждая программа заработала должным образом с первой попытки, можно было бы говорить о завершении цикла разработки: написание программы, компиляция исходного кода, компоновка программы и ее выполнение. К сожалению, почти все программы (тривиальные и не очень) содержат ошибки. Одни ошибки обнаружит компилятор, другие — компоновщик, а третьи проявятся только при запуске программы в работу.

Любая ошибка должна быть исправлена, и для этого нужно отредактировать исходный текст программы, перекомпилировать его и перекомпоновать, а затем снова выполнить. Этот цикл разработки представлен на рис. 1.1.

Первая программа на языке C++ — HELLO.cpp

Традиционно в книгах по программированию первые примеры программ начинаются с вывода на экран слов `Hello World` или какой-нибудь вариации на тему. В этой книге мы следовали устоявшимся традициям.

Введите первую программу с помощью текстового редактора, в точности повторяя все нюансы. Завершив ввод, сохраните файл, скомпилируйте его, скомпонуйте и выполните. Программа должна вывести на экран слова `Hello World`. Пока не стоит задумываться о том, как работает эта программа. Вы должны получить удовлетворение просто от того, что прошли полный цикл разработки. Все аспекты этой программы будут подробно рассмотрены на следующих занятиях.

ПРЕДУПРЕЖДЕНИЕ

В приведенном ниже листинге слева содержатся номера строк. Эти номера установлены лишь для ссылки в тексте книги на соответствующие строки программы. Их не нужно вводить в окно редактора. Например, в первой строке листинга 1.1 вы должны ввести:

```
#include <iostream.h>
```

Листинг 1.1. Файл `HELLO.cpp` — программа приветствия

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:  cout << "Hello World!\n";
6:  return 0;
7: }
```

Убедитесь в том, что введенный вами текст программы совпадает с содержимым приведенного здесь листинга. Обратите внимание на знаки препинания. Символ << в строке 5 является оператором перенаправления потока данных. Эти символы на большинстве клавиатур вводятся путем нажатия клавиши <Shift> и двойного нажатия клавиши с запятой. Строка 5 завершается точкой с запятой (;). Не пропустите этот символ завершения строки программного кода!

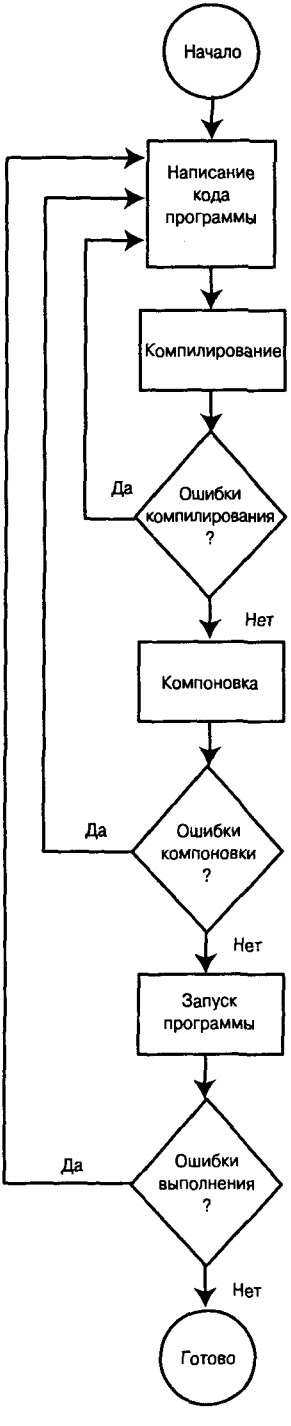


Рис. 1.1. Этапы разработки программы на языке C++

Кроме того, убедитесь, что вы корректно работаете со своим компилятором. Большинство компиляторов переходит к компоновке автоматически, но все-таки стоит свериться с документацией. Если вы получите какие-нибудь сообщения об ошибках, просмотрите внимательно текст своей программы и найдите отличия от варианта, приведенного в книге. Если вы увидите сообщение об ошибке со ссылкой на строку 1, уведомляющее о невозможности найти файл `iostream.h` (`cannot find file iostream.h`), обратитесь к документации за указаниями об установке пути для включаемых файлов или переменных окружения. Если вы получите сообщение об ошибке, уведомляющее об отсутствии прототипа для функции `main`, добавьте строку `int main();` сразу перед строкой 3. В этом случае вам придется добавлять эту строку до начала функции `main` в каждой программе, приведенной в этой книге. Большинство компиляторов не требует наличия прототипа для функции `main`, но вполне возможно, что именно вам достался компилятор из другой компании.

Один из возможных вариантов программы будет выглядеть следующим образом:

```
1: #include <iostream.h>
2: int main(); // большинство компиляторов не требует этой строки
3: int main()
4: {
5:   cout << " Hello World!\n";
6:   return 0;
7: }
```

Попробуйте выполнить программу `HELLO.exe`. Если все правильно, вы должны увидеть на экране приветствие:

```
Hello world!
```

Использование стандартных библиотек

Чтобы гарантировать, что все наши читатели, работающие со старыми компиляторами, не будут иметь проблем с программами из этой книги, мы используем старый стиль включения файлов:

```
#include <iostream.h>
```

а не заголовки новых стандартных библиотек:

```
#include <iostream>
```

Такой вариант включения должен работать на всех компиляторах, тем не менее он имеет ряд недостатков. Если вы предпочитаете использовать новые стандартные библиотеки, просто замените в своей программе строку 1 строкой

```
#include <iostream>
```

и добавьте строку

```
using namespace std;
```

сразу после списка включаемых файлов. Нюансы использования пространства имен подробно рассматриваются на занятии 17.

Будете вы использовать стандартные заголовочные файлы или нет, программы, приведенные в этой книге, должны работать без каких бы то ни было модификаций. Принципиальное отличие старых библиотек от новых стандартов заключается в использовании библиотеки `iostream` (см. занятие 16). Но даже эти изменения не должны оказать влияние на программы из этой книги ввиду их незначительности. Кроме того, они выходят за рамки обсуждения круга тем, предусмотренных для начинающих.

Трудно читать текст программы даже про себя, если не знаешь, как произносить специальные символы и ключевые слова. Советую читать первую строку так: "паунд инклуд (# — символ фунта) ай-оу-стрим-дот(или точка)-эйч". А строку 5 читайте как "си-аут-'Hello world!'".

Если увидели, то примите наши поздравления! Вы только что ввели, скомпилировали и запустили свою первую программу на языке C++. Конечно, она не поражает своей грандиозностью, но почти каждый профессиональный программист начинал именно с такой программы.

Осваиваем компилятор Visual C++ 6

Все программы в этой книге проверены на компиляторе Visual C++ 6.0 и должны прекрасно компилироваться, компоноваться и выполняться при использовании любого компилятора Microsoft Visual C++, по крайней мере начиная с версии 4.0 и выше. Теоретически, поскольку мы имеем дело с ANSI-совместимым текстом программ, все программы в этой книге должны компилироваться любым ANSI-совместимым компилятором любого производителя.

В идеале результаты выполнения программ должны совпадать с приведенными в этой книге, но на практике не всегда так бывает.

Чтобы наконец приступить к делу, ознакомьтесь в этом разделе с тем, как редактировать, компилировать, компоновать и выполнять программу, используя компилятор компании Microsoft. Если у вас другой компилятор, на каких-то этапах возможны некоторые отличия. Даже если вы используете компилятор Microsoft Visual C++ 6.0, все равно стоит свериться с документацией и уточнить все детали.

Построение проекта приветствия

Чтобы создать и протестировать программу приветствия, выполните ряд действий.

1. Запустите компилятор.
2. Выберите из меню File команду New.
3. Выберите опцию Win32 Console Application (Консольное приложение для Win32), введите имя проекта, например `Example 1`, и щелкните на кнопке OK.
4. Выберите из меню вариант An Empty Project (Пустой проект) и щелкните на кнопке OK.
5. Выберите в меню File команду New.
6. Выберите опцию C++ Source File (файл источника C++) и введите имя проекта `ex1`.
7. Введите текст программы, приведенный выше.
8. Выберите в меню Build команду Build Example1.exe.
9. Убедитесь в отсутствии ошибок компиляции.
10. Нажмите клавиши <Ctrl+F5> для выполнения программы.
11. Нажмите клавишу пробела для завершения программы.

Ошибки компиляции

Ошибки в процессе компиляции могут возникать по различным причинам. Обычно они являются результатом небрежного ввода и другого рода случайностей. Приличные компиляторы сообщат не только о том, что именно у вас не в порядке, они также укажут точное местоположение обнаруженной ошибки. Самые "продвинутые" компиляторы даже предложат вариант исправления ошибки!

В этом можно убедиться, специально сделав ошибку в нашей программе. Давайте удалим в программе HELLO.cpp закрывающую фигурную скобку в строке 7. Ваша программа теперь будет выглядеть так, как показано в листинге 1.2.

Перекомпилируйте программу, и вы увидите сообщение об ошибке, которое выглядит примерно следующим образом:

```
Hello.cpp, line 5: Compound statement missing terminating; in function main().
```

Листинг 1.2. Демонстрация ошибки компиляции

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:  cout << "Hello world!\n";
6:  return 0;
```

Либо вы можете увидеть такое сообщение об ошибке:

```
F:\ Mcp\ Tycpp21d\ Testing\ List0101.cpp(8) : fatal error C1004:
unexpected end of file found
Error executing cl.exe.
 }
```

В этом сообщении содержится информация о том, где гнездится проблема (указывается имя файла, номер строки и характер проблемы, хотя и в несколько зашифрованном виде). Обратите внимание на то, что в сообщении об ошибке указывается строка 5. Компилятор не уверен в вашем намерении вставить закрывающую фигурную скобку перед или после инструкции, содержащей объект cout. Иногда в сообщениях проблема обрисовывается только в общих чертах. Если бы компилятор мог точно идентифицировать каждую ошибку, то он бы тогда мог сам ее и исправить.

Резюме

Надеюсь, прочитав эту главу, вы получили хорошее представление об эволюции языка C++, а также о том, для решения каких проблем он предназначен. У вас не должно остаться сомнений по поводу того, что изучение C++ — правильный выбор для всякого, кто собирается программировать в ближайшие десять лет. В C++ предусмотрены средства объектно-ориентированного программирования, обеспечивающие эффективность языка системного уровня, благодаря чему C++ заслуженно выбирают в качестве языка разработки.

Сегодня вы научились вводить, компилировать, компоновать и выполнять свою первую программу на C++ и узнали, что представляет собой цикл разработки. Вы также получили небольшое представление об объектно-ориентированном программировании. Нам предстоит еще не раз коснуться этих тем в течение трех недель.

Вопросы и ответы

Что такое текстовый редактор?

Текстовый редактор создает и редактирует файлы, содержащие текст. Для написания текстов программ не требуется никаких атрибутов форматирования или специальных символов. Текстовые файлы с листингами программ не обладают такими свойствами, как автоматический перенос слов либо начертание букв полужирным шрифтом или курсивом и т.д.

Если мой компилятор имеет встроенный редактор, то обязан ли я использовать его?

Почти все компиляторы будут компилировать программы, созданные в любом текстовом редакторе. Однако преимущества использования встроенного текстового редактора состоит в том, что он может быстро переключаться между режимами редактирования и компиляции. Высокоорганизованные компиляторы включают полностью интегрированную среду разработки, позволяя программисту легко получать доступ к справочным файлам, редактировать, компилировать и сразу же исправлять ошибки компиляции и компоновки, не выходя из среды разработки.

Могу ли я игнорировать предупреждающие сообщения, поступающие от компилятора?

Среди программистов распространено мнение, что на предупреждающие сообщения компилятора можно не обращать внимания, но я придерживаюсь другого мнения. Возьмите за правило реагировать на предупреждения компилятора как на сообщения об ошибках. Компилятор C++ генерирует предупреждающие сообщения в тех случаях, когда, по его мнению, вы делаете то, что не входит в ваши намерения. Внимательно отнеситесь к этим предупреждениям и сделайте все, чтобы они исчезли.

Что означает время компиляции?

Это время работы вашего компилятора, в отличие от времени компоновки (когда работает компоновщик) или времени выполнения программы (когда выполняется программа). Эти термины придумали программисты, чтобы кратко обозначить временные периоды, в течение которых обычно и проявляются различные ошибки.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. В чем разница между интерпретатором и компилятором?
2. Как происходит компиляция исходного кода программы?
3. В чем состоит назначение компоновщика?
4. Какова обычная последовательность действий в цикле разработки?

Упражнения

1. Просмотрите следующую программу и попытайтесь понять, что она делает, не запуская ее на выполнение.

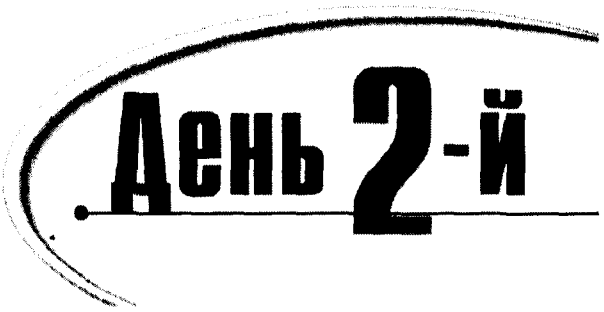
```
1: #include <iostream.h>
2: int main()
3: {
4:    int x = 5;
5:    int y = 7;
6:    cout << "\n";
7:    cout << x + y << " " << x * y;
8:    cout << "\n";
9:    return 0;
10: }
```

2. Введите программу из упражнения 1, а затем скомпилируйте и запустите ее. Что она делает? Так ли вы все это предполагали?

3. Введите следующую программу и скомпилируйте ее. Какие сообщения об ошибках вы получили?

```
1: include <iostream.h>
2: int main()
3: {
4:    cout << "Hello World\n";
5:    return 0;
6: }
```

4. Исправьте ошибку в программе из упражнения 3, а затем перекомпилируйте, скомпонуйте и выполните ее. Что делает эта программа?



Составные части программы на языке C++

Программы на языке C++ состоят из объектов, функций, переменных и других элементов. Большая часть этой книги посвящена подробному описанию каждого из них, но, для того чтобы получить представление о слаженной работе всех этих элементов, нужно рассмотреть какую-нибудь законченную рабочую программу. Сегодня вы узнаете:

- Из каких частей состоят программы на языке C++
- Как эти части взаимодействуют друг с другом
- Что такое функция и каково ее назначение

Простая программа на языке C++

Даже простенькая программа HELLO.CPP, приведенная на занятии 1, состоит из нескольких элементов, которые представляют для нас интерес. В этом разделе упомянутая программа рассматривается более подробно. В листинге 2.1 ради удобства обсуждения приведена оригинальная версия файла HELLO.CPP.

Листинг 2.1. Демонстрация частей программы C++ на примере программы HELLO.CPP

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:  cout << "Hello World!\ n";
6:  return 0;
7: }
```

РЕЗУЛЬТАТ

```
Hello World!
```

АНАЛИЗ В строке 1 выполняется включение файла `iostream.h` в текущий файл. Первым в программе стоит символ `#`, который служит сигналом для препроцессора. При каждом запуске компилятора запускается и препроцессор. Он читает исходный текст программы, находит строки, которые начинаются с символа фунта (`#`), и работает с этими строками до того, как начнется компиляция программы. Подробнее работа препроцессора рассматривается на занятии 21.

`Include` — это команда препроцессору, которую можно расшифровать следующим образом: "За именем команды следует имя файла. Нужно найти этот файл и вставить его содержимое прямо в это место программы". Угловые скобки, в которые заключено имя файла, означают, что этот файл нужно искать во всех папках, отведенных для хранения подобных файлов. Если ваш компилятор настроен корректно, то угловые скобки укажут препроцессору на то, что файл `iostream.h` следует искать в папке, содержащей все файлы с расширением `.h`, предназначенные для вашего компилятора. Файл `iostream.h` (*input-output-stream* — поток ввода-вывода) используется объектом `cout`, который обслуживает процесс вывода данных на экран. После выполнения строки 1 файл `iostream.h` будет включен в эту программу таким образом, как если бы вы собственноручно ввели сюда его содержимое. Препроцессор запускается перед компилятором и выполняет все строки, начинающиеся с символа (`#`), подготавливая код программы к компиляции.

Основной код программы начинается в строке 3 с вызова функции `main()`. Каждая программа на языке C++ содержит функцию `main()`. Функция — это блок программы, который выполняет одно или несколько действий. Обычно функции вызываются другими функциями, но `main()` — особая функция: она вызывается автоматически при запуске программы.

Функция `main()`, подобно всем другим функциям, должна объявить тип возвращаемого значения. В программе `HELLO.CPP` функция `main()` возвращает значение типа `int` (от слова *integer* — целый), а это значит, что по окончании работы эта функция возвратит операционной системе целочисленное значение. В данном случае будет возвращено целое значение `0`, как показано в строке 6. Возвращение значения в операционную систему не столь важно, и в общем-то это значение самой системой никак не используется, но стандарт языка C++ требует, чтобы функция `main()` была объявлена по всем правилам (как показано в этом листинге).

ПРИМЕЧАНИЕ

Некоторые компиляторы позволяют объявить функцию `main()` таким образом, чтобы она возвращала значение типа `void`. Этого больше нельзя делать в C++, поэтому вам следует избавляться от старых привычек. Позвольте функции `main()` возвращать значения типа `int` и ради этого поместите в последней строке этой функции выражение `return 0;`.

ПРИМЕЧАНИЕ

В некоторых операционных системах предусмотрена возможность проверки значения, возвращаемого программой. Удобно возвращать значение `0` как флаг нормального завершения функции.

Все функции начинаются открывающей фигурной скобкой ({) и оканчиваются закрывающей фигурной скобкой (}). Фигурные скобки функции `main()` помещены в строках 4 и 7. Все, что находится между открывающей и закрывающей фигурными скобками, считается телом функции.

Вся функциональность нашей простейшей программы заключена в строке 5.

Объект `cout` используется для вывода сообщений на экран. Об объектах пойдет речь на занятии 6, а объект `cout` и близкий ему объект `cin` будут подробно рассмотрены на занятии 16. Эти два объекта, `cin` и `cout`, используются в языке C++ для организации соответственно ввода данных (например, с клавиатуры) и их вывода (например, на экран).

Вот как используется объект `cout`: вводим слово `cout`, за которым ставим оператор перенаправления выходного потока `<<` (далее будем называть его оператором вывода). Все, что следует за этим оператором, будет выводиться на экран. Если вы хотите вывести на экран строку текста, не забудьте заключить ее в двойные кавычки ("), как показано в строке 5.

Строка текста — это набор печатаемых символов.

Два заключительных символа текстовой строки (`\n`) означают, что после слов *Hello world!* нужно выполнить переход на новую строку. Этот специальный код подробно объясняется при рассмотрении объекта `cout` на занятии 17.

Функция `main()` оканчивается в строке 7.

Кратко об объекте cout

На занятии 16 вы узнаете, как использовать объект `cout` для вывода данных на экран. А пока, не вдаваясь в детали использования объекта `cout`, скажем, что для вывода значения на экран нужно ввести слово `cout`, а за ним оператор вывода (`<<`), который состоит из двух символов "меньше" (`<`). Несмотря на то что вы вводите два символа, компилятор C++ воспринимает их как один оператор.

Листинг 2.2. Использование объекта `cout`

```
1: // Листинг 2.2. Использование объекта cout
2: #include <iostream.h>
3: int main()
4: {
5:    cout << "Hello there.\ n";
6:    cout << "Here is 5: " << 5 << "\ n";
7:    cout << "The manipulator endl writes a new line to the screen.";
8:    cout <<
9:           endl;
10:   cout << "Here is a very big number:\ t" << 70000 << endl;
11:   cout << "Here is the sum of 8 and 5:\ t" << 8+5 << endl;
12:   cout << "Here's a fraction:\ t\ t" << (float) 5/8 << endl;
13:   cout << "And a very very big number:\ t";
14:   cout << (double) 7000 * 7000 <<
15:          endl;
16:    cout << "Don't forget to replace Jesse Liberty with your name...\ n";
17:    cout << "Jesse Liberty is a C++ programmer!\ n";
18:    return 0;
19:    }
```

За символом вывода укажите выводимые данные. Использование объекта cout показано в листинге 2.2. Введите текст этой программы в точности так, как написано, за исключением одного: вместо имени Jesse Liberty подставьте свои имя и фамилию, лучше латинскими буквами.

РЕЗУЛЬТАТ

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:    70000
Here is the sum of 8 and 5:   13
Here's a fraction:            0.625
And a very very big number:   4.9e+07
Don't forget to replace Jesse Liberty with your name...
Jesse Liberty is a C++ programmer!
```

ПРИМЕЧАНИЕ

Некоторые компиляторы требуют, чтобы математические операции в случае использования после объекта cout заключались в круглые скобки. В этом случае строку 11 пришлось бы изменить следующим образом:

```
11:    cout << "Here is the sum of 8 and 5:\ t" << (8+5) << endl;
```

АНАЛИЗ

В строке 2 по команде `#include <iostream.h>` препроцессор вставляет содержимое файла iostream.h в исходный текст программы. Включать файл iostream.h необходимо, если в программе используется объект cout и связанные с ним функции-члены.

В строке 5 демонстрируется простейший вариант использования объекта cout: вывод строки символов. Символ \n — это специальный символ форматирования, который указывает объекту cout на необходимость вывода на экран символа новой строки (он произносится "слэш-эн" или просто разрыв строки).

В строке 6 объекту cout передаются три значения, и каждое из них отделяется оператором вывода. Первое значение представляет собой строку "Here is 5: ". Обратите внимание на наличие пробела после двоеточия: пробел является частью текстовой строки. Затем объекту cout с помощью оператора вывода передается значение 5, а за ним — символ разрыва строки (этот символ всегда должен быть заключен в двойные или в одинарные кавычки). При выполнении этого выражения на экране появится строка

```
Here is 5: 5
```

Поскольку после первого значения нет символа разрыва строки, следующее значение выводится сразу за предыдущим. Этот процесс называется конкатенацией двух значений.

В строке 7 на экран выводится информационное сообщение, после чего используется оператор endl. Этот оператор также выводит на экран символ разрыва строки. (Другое назначение оператора endl рассматриваются на занятии 16.)

ПРИМЕЧАНИЕ

Оператор endl расшифровывается как ***end line*** (конец строки) и читается как "энд-эл", а не "энд-один" (иногда букву *l* принимают за единицу).

В строке 10 используется еще один символ форматирования — \t, который вставляет символ табуляции, используемый обычно для выравнивания выводимой информации (строки 10–13). Строка 10 демонстрирует возможность вывода значений типа long int. В строке 11 показано, что объект cout может выводить результат математической операции. Объекту cout передается не значение, а целое математическое выражение 8+5, но на экран выводится число 13.

В строке 12 объект cout выводит результат другой математической операции — 5/8. Идентификатор (float) указывает объекту cout, что результат должен выводиться как дробное число. В строке 14 объекту cout передается выражение 7000 * 7000, а идентификатор (double) устанавливает вывод результата в экспоненциальном представлении. Использование идентификаторов double и float для установки типов значений рассматривается на занятии 3.

В строке 16 нужно вставить свое имя. Если при выполнении программы вы увидите свое имя на экране, шансы стать профессиональным программистом у вас существенно возрастут, хотя в этом и так нет никаких сомнений. Даже компьютер это знает!

Комментарии

Когда вы пишете программу, вам всегда ясно, что вы стараетесь сделать. Однако если через месяц вам придется вернуться к этой программе, то, как это ни удивительно, окажется, что вы почти совсем не помните, о чем идет речь, а о деталях и говорить не приходится.

Чтобы не казнить себя за пробелы в памяти и помочь другим понять вашу программу, используйте комментарии. Комментарии представляют собой текст, который игнорируется компилятором, но позволяет описать прямо в программе назначение отдельной строки или целого блока.

Виды комментариев

В языке C++ используется два вида комментариев: с двойным слешем (//) и сочетанием слеша и звездочки (/*). Комментарий с двойным слешем (его называют комментарием в стиле C++) велит компилятору игнорировать все, что следует за этими символами вплоть до конца текущей строки.

Комментарий со слешем и звездочкой (его называют комментарием в стиле C) велит компилятору игнорировать все, что следует за символами (/*) до того момента, пока не встретится символ завершения комментария: звездочка и слэш (*/). Каждой открывающей паре символов /* должна соответствовать закрывающая пара символов */.

Нетрудно догадаться, что комментарии в стиле C используются также и в языке C, но следует предупредить, что двойной слеш в языке C не воспринимается как символ комментария.

При программировании на C++ для выделения комментариев в основном используются символы двойного слеша, а комментарии в стиле C используются только для временного отключения больших блоков программы. Впрочем, двойной слеш часто используется и для временного отключения отдельных строк программного кода.

Использование комментариев

Раньше считалось хорошим тоном предварять блоки функций и саму программу комментариями, из которых должно было быть понятно, что делает эта функция и какое значение она возвращает.

Исходя из собственного опыта, могу сказать, что такие комментарии не всегда целесообразны. Комментарии в заголовке программы очень быстро устаревают, поскольку практически никто их не обновляет при обновлении текста программы. Функции должны иметь такие имена, чтобы у вас не оставалось ни тени сомнения в том, что они делают, в противном случае имя функции нужно изменить. Зачем использовать бессмысленные и труднопроизносимые имена, чтобы потом раскрывать их смысл с помощью комментариев?

Впрочем, одно другому не помеха. Лучше всего использовать понятные имена и дополнительно вносить краткие разъяснения с помощью комментариев.

Листинг 2.3 демонстрирует использование комментариев, доказывая, что они не влияют на выполнение программы и ее результаты.

Листинг 2.3. Демонстрация комментариев на примере программы HELLO.CPP

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:   /* это комментарий,
6:   который продолжается до тех пор, пока не
7:   встретится символ конца комментария в виде звездочки и слэша */
8:   cout << "Hello world!\ n";
9:   // Этот комментарий оканчивается в конце строки
10:  cout << "That comment ended!\ n";
11:
12:  // после двойного слеша может не быть никакого текста,
13:  /* как, впрочем, и между этими символами */
14:  return 0;
15: }
```

РЕЗУЛЬТАТ

```
Hello world!
That comment ended!
```

АНАЛИЗ Комментарии в строках 5—7 полностью игнорируются компилятором, как и комментарии в строках 9, 12 и 13. Комментарий в строке 9 завершается в конце этой строки, но для завершения комментариев, начавшихся в строках 5 и 13, требуется символ окончания комментария (*/).

Напоследок предупреждение: осторожнее с комментариями!

В комментариях, разъясняющих очевидные места, проку немного. Но они могут даже вводить в заблуждение, если при изменения текста программы вы забудете их скорректировать. Однако очевидность — понятие относительное. То, что

очевидно для одного человека, может быть непонятно другому. Всегда старайтесь разумно комментировать свои действия и не забывайте обновлять комментарии при обновлении программы.

И последнее, комментарии должны разъяснять, не что это за операторы, а для чего они тут используются.

Функции

Вы уже познакомились с функцией `main()`, правда, это необычная, единственная в своем роде функция. Чтобы приносить пользу, функция должна быть вызвана во время сеанса работы программы. Функция `main()` вызывается не программой, а операционной системой.

Программа выполняется по строкам в порядке их расположения в исходном тексте до тех пор, пока не встретится вызов какой-нибудь функции. Затем управление передается строкам этой функции. После выполнения функции управление возвращается той строке программы, которая следует сразу за вызовом функции.

Есть прекрасная аналогия для работы программы с функцией. Например, если во время рисования у вас ломается карандаш, вы прекращаете рисовать и затачиваете его. После этого вы возвращаетесь к тому месту рисунка, где сломался карандаш. Когда программа нуждается в выполнении некоторой сервисной операции, вызывается функция, ответственная за выполнение этой операции, после чего программа продолжает свою работу с того места, где была вызвана функция. Эта идея демонстрируется в листинге 2.4.

Листинг 2.4. Пример вызова функции

```
1:  #include <iostream.h>
2:
3:  // Функция DemonstrationFunction
4:  // выводит на экран информативное сообщение
5:  void DemonstrationFunction()
6:  {
7:    cout << "In DemonstrationFunction\ n";
8:  }
9:
10: // Функция main выводит сообщение, затем
11: // вызывает функцию DemonstrationFunction и
12: // выводит на экран второе сообщение.
13: int main()
14: {
15:   cout << "In main\ n" ;
16:   DemonstrationFunction();
17:   cout << " Back in main\ n";
18:   return 0;
19: }
```

Результат

```
In main
In DemonstrationFunction
Back in main
```

АНАЛИЗ

В строках 5–8 определяется функция DemonstrationFunction(). Она выводит на экран сообщение и возвращает управление программе.

Функция main() начинается в строке 13, и в строке 15 выводится на экран сообщение, уведомляющее о том, что сейчас управление программой находится в функции main(). После вывода этого сообщения в строке 16 вызывается функция DemonstrationFunction(). В результате этого вызова выполняются команды, содержащиеся в функции DemonstrationFunction(). В данном случае вся функция состоит из одной команды, содержащейся в строке 7, которая выводит другое сообщение. По завершении выполнения функции DemonstrationFunction() (строка 8) управление программой возвращается туда, откуда эта функция была вызвана. В данном случае выполнение программы продолжается со строки 17, в которой функция main() выводит на экран заключительное сообщение.

Использование функций

Функции возвращают либо некоторое реальное значение, либо значение типа void, т.е. ничего не возвращают. Функцию, которая складывает два целых числа и возвращает значение суммы, следует определить как возвращающую целочисленное значение. Функции, которая только выводит сообщение, возвращать нечего, поэтому для нее задается тип возврата void.

Функции состоят из заголовка и тела. Заголовок содержит установки типа возвращаемого значения, имени и параметров функции. Параметры позволяют передавать в функцию значения. Следовательно, если функция предназначена для сложения двух чисел, то эти числа необходимо передать в функцию как параметры. Вот как будет выглядеть заголовок такой функции:

```
int Sum(int a, int b)
```

Параметр — это объявление типа данных значения, передаваемого в функцию. Реальное значение, передаваемое при вызове функции, называется *аргументом*. Многие программисты используют эти два понятия как синонимы. Другие считают смешение этих терминов признаком непрофессионализма. Возможно, это и так, но в данной книге эти термины взаимозаменяемы.

Тело функции начинается открывающей фигурной скобкой и содержит ряд строк (хотя тело функции может быть даже нулевым), за которыми следует закрывающая фигурная скобка. Назначение функции определяется содержащимися в ней строками программного кода. Функция может возвращать значение в программу с помощью оператора возврата (return). Этот оператор также означает выход из функции. Если не поместить в функцию оператор возврата, то по завершении функции автоматически возвращается значение типа void. Значение, возвращаемое функцией, должно иметь тип, объявленный в заголовке функции.

Более подробно функции рассматриваются на занятии 5; типы значений, возвращаемых функциями, — на занятии 3. Информация, представленная на этом занятии, является хотя и обзорной, но вполне достаточной для усвоения последующего материала, поскольку функции будут использоваться практически во всех программах, представленных в этой книге.

В листинге 2.5 демонстрируется функция, которая принимает два целочисленных параметра и возвращает целочисленное значение. Не беспокойтесь пока насчет синтаксиса или особенностей работы с целыми значениями (например, `int x`): эта тема подробно раскрывается на занятии 3.

Листинг 2.5. Пример использования простой функции (FUNC.CPP)

```
1:  #include <iostream.h>
2:  int Add (int x, int y)
3:  {
4:
5:    cout << "In Add(), received " << x << " and " << y << "\ n";
6:    return (x+y);
7:  }
8:
9:  int main()
10:  {
11:    cout << "I'm in main()!\ n";
12:    int a, b, c;
13:    cout << "Enter two numbers: ";
14:    cin >> a;
15:    cin >> b;
16:    cout << "\ nCalling Add()\ n";
17:    c=Add(a,b);
18:    cout << "\ nBack in main().\ n";
19:    cout << "c was set to " << c;
20:    cout << "\ nExiting...\ n\ n";
21:    return 0;
22:  }
```

РЕЗУЛЬТАТ

```
I'm in main()!
Enter two numbers: 3 5

Calling Add()
In Add(), received 3 and 5

Back in main().
c was set to 8
Exiting...
```

АНАЛИЗ Функция `Add()` определена в строке 2. Она принимает два целочисленных параметра и возвращает целочисленное значение. Сама же программа начинается в строке 9, выводя на экран первое сообщение. Затем пользователю предлагается ввести два числа (строки 13–15). Пользователь вводит числа, разделяя их пробелом, а затем нажимает клавишу <Enter>. В строке 17 функция `main()` передает функции `Add()` в качестве аргументов два числа, введенные пользователем.

Управление программой переходит к функции `Add()`, которая начинается в строке 2. Параметры `a` и `b` выводятся на экран, а затем складываются. Результат функции возвращается в строке 6, и на этом функция завершает свою работу.

Резюме

Сложность изучения такого предмета, как программирование, состоит в следующем: большая часть изучаемого вами материала во многом зависит от того, что вам еще только предстоит изучить. На этом занятии вы познакомились с основными составляющими частями простой программы на языке C++. Кроме того, вы получили представление о цикле разработки и узнали несколько важных терминов.

Вопросы и ответы

Какую роль выполняет директива `#include`?

Это команда для препроцессора, который автоматически запускается при вызове компилятора. Данная директива служит для введения содержимого файла, имя которого стоит после директивы, в исходный текст программы.

В чем разница между символами комментариев // и /*?

Комментарии, выделенные двойным слешем (//), распространяются до конца строки. Комментарии, начинающиеся слешем со звездочкой (/*), продолжаются до тех пор, пока не встретится символ завершения комментария (*/). Помните, что даже конец функции не завершит действие комментария, начавшегося с пары символов (/*). Если вы забудете установить завершение комментария (*/), то получите сообщение об ошибке во время компиляции.

В чем разница между хорошими и плохими комментариями?

Хороший комментарий сообщит читателю, почему здесь используются именно эти операторы, или объяснит назначение данного блока программы. Плохой комментарий констатирует то, что делается в данной строке программы. Программа в идеале должна писаться так, чтобы имена переменных и функций говорили сами за себя, а логика выражений была проста и понятна без особых комментариев.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. В чем разница между компилятором и препроцессором?
2. В чем состоит особенность функции `main()`?
3. Какие два типа комментариев вы знаете и чем они отличаются друг от друга?
4. Могут ли комментарии быть вложенными?
5. Могут ли комментарии занимать несколько строк?

Упражнения

1. Напишите программу, которая выводит на экран сообщение `I love C++`.
2. Напишите самую маленькую программу, которую можно скомпилировать, скомпоновать и выполнить.
3. **Жучки:** введите эту программу и скомпилируйте ее. Почему она дает сбой? Как ее можно исправить?

```
1: #include <iostream.h>
2: int main()
3: {
4:   cout << Is there a bug here?";
5:   return 0;
6: }
```

4. Исправьте ошибку в упражнении 3, после чего перекомпилируйте ее, скомпонуйте и запустите на выполнение.

День 3-й

Переменные и константы

Программы должны обладать способностью хранить используемые данные. Для представления и манипуляции этими данными используются переменные и константы. Сегодня вы узнаете:

- Как объявлять и определять переменные и константы
- Как присваивать значения переменным и использовать их в программе
- Как выводить значения переменных на экран

Что такое переменная

В языке C++ переменные используются для хранения информации. Переменную можно представить себе как ячейку в памяти компьютера, в которой может храниться некоторое значение, доступное для использования в программе.

Память компьютера можно рассматривать как ряд ячеек. Все ячейки последовательно пронумерованы. Эти номера называют адресами памяти. Переменная занимает одну или несколько ячеек, в которых можно хранить некоторое значение.

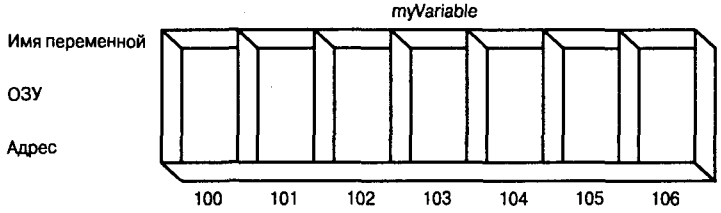


Рис. 3.1. Сохранение переменной в памяти компьютера

Имя переменной (например, `MyVariable`) можно представить себе как надпись на ячейке памяти, по которой, не зная настоящего адреса памяти, можно ее найти. На рис. 3.1 схематически представлена эта идея. Согласно этому рисунку, переменная `MyVariable` начинается с ячейки с адресом 103. В зависимости от своего размера, переменная `MyVariable` может занимать одну или несколько ячеек памяти.

В ОЗУ обеспечивается произвольный доступ к ячейкам памяти. Запускаемая программа загружается в ОЗУ с дискового файла. Все переменные также хранятся в ОЗУ. Когда программисты говорят о памяти, они обычно имеют в виду ОЗУ.

Резервирование памяти

При определении переменной в языке C++ необходимо предоставить компилятору информацию о ее типе, например `int`, `chart` или другого типа. Благодаря этой информации компилятору будет известно, сколько места нужно зарезервировать для нее и какого рода значение будут хранить в этой переменной.

Каждая ячейка имеет размер в один байт. Если для переменной указанного типа требуется четыре байта, то для нее будет выделено четыре ячейки, т.е. именно по типу переменной (например, `int`) компилятор судит о том, какой объем памяти (сколько ячеек) нужно зарезервировать для этой переменной.

Поскольку для представления значений в компьютерах используются биты и байты и память измеряется тоже в байтах, важно хорошо разбираться в этих понятиях. Более полно эта тема рассматривается в приложении В.

Размер целых

Для переменных одних и тех же типов на компьютерах разных марок может выделяться разный объем памяти, в то же время в пределах одного компьютера две переменные одинакового типа всегда будут иметь постоянный размер.

Переменная типа `char` (используемая для хранения символов) чаще всего имеет размер в один байт.

Не прекращаются споры о произношении имени типа `char`. Одни произносят его как "кар", другие — как "чар". Поскольку это сокращение слова character, то первый вариант правильнее, но вы вольны произносить его так, как вам удобно.

В большинстве компьютеров для типа `short int` (короткий целый) обычно отводится два байта, для типа `long int` (длинный целый) — четыре байта, а для типа `int` (без ключевого слова `short` или `long`) может быть отведено два или четыре байта. Размер целого значения определяется системой компьютера (16- или 32-разрядная) и используемым компилятором. На современных 32-разрядных компьютерах, использующих последние версии компиляторов (например, Visual C++ 4 или более поздние), целые переменные имеют размер в *четыре* байта. Эта книга ориентирована на использование 4-байтовых целых, хотя у вас может быть другой вариант. Программа, представленная в листинге 3.1, поможет определить точный размер этих типов на вашем компьютере.

Под символом подразумевается одиночная буква, цифра или знак, занимающий только один байт памяти.

Листинг 3.1. Определение размеров переменных разных типов на вашем компьютере

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:   cout << "The size of an int is:\t\t" << sizeof(int) << " bytes.\n";
6:   cout << " The size of a short int is:\t\t" << sizeof(short) << " bytes.\n";
7:   cout << " The size of a long int is:\t\t" << sizeof(long) << " bytes.\n";
8:   cout << " The size of a char is:\t\t" << sizeof(char) << " bytes.\n";
9:   cout << " The size of a float is:\t\t" << sizeof(float) << " bytes.\n";
10:  cout << " The size of a double is:\t\t" << sizeof(double) << " bytes.\n";
11:  cout << " The size of a bool is:\t\t" << sizeof(bool) << " bytes.\n";
12:
13:  return 0;
14: };
```

Вывод

```
The size of an int is:          4 bytes.
The size of a short int is:     2 bytes.
The size of a long int is:      4 bytes.
The size of a char is:          1 bytes.
The size of a float is:         4 bytes.
The size of a double is:        4 bytes.
The size of a bool is:          1 bytes.
```

ПРИМЕЧАНИЕ

На вашем компьютере размеры переменных разных типов могут быть другими.

Анализ Большинство операторов листинга 3.1 вам знакомо. Возможно, новым для вас будет использование функции `sizeof()` в строках 5–10. Результат выполнения функции `sizeof()` зависит от компилятора и компьютера, а ее назначение состоит в определении размеров объектов, переданных в качестве параметра. Например, в строке 5 функции `sizeof()` передается ключевое слово `int`. Функция возвращает размер в байтах переменной типа `int` на данном компьютере. В нашем примере для типов `int` и `long int` возвращается значение четыре байта.

Знаковые и беззнаковые типы

Целочисленные переменные, используемые в программах, могут быть знаковыми и беззнаковыми. Иногда бывает полезно установить для переменной использование только положительных чисел. Целочисленные типы (`short` и `long`) без ключевого слова `unsigned` считаются знаковыми. Знаковые целые могут быть отрицательными или положительными. Беззнаковые числа всегда положительные.

Поскольку как для знаковых, так и для беззнаковых целых отводится одно и то же число байтов, то максимальное число, которое можно хранить в беззнаковом целом,

вдвое превышает максимальное положительное число, которое можно хранить в знаковом целом. С помощью типа `unsigned short int` можно обрабатывать числа в диапазоне 0—65 535. Половина чисел, представляемых знаковым коротким целым типом, отрицательные, следовательно, с помощью этого типа можно представить числа только в диапазоне - 32 768—32 767. Если в этом вопросе вам что-то неясно, прочитайте приложение В.

Базовые типы переменных

В языке C++ предусмотрены и другие типы переменных. Они делятся на целочисленные (которые рассматривались до сих пор), вещественные (с плавающей точкой) и символьные.

Вещественные переменные содержат значения, которые могут выражаться в виде дробей. Символьные переменные занимают один байт и используются для хранения 256 символов и знаков ASCII, а также расширенных наборов символов ASCII.

Под символами ASCII понимают стандартный набор знаков, используемых в компьютерах. ASCII — это American Standard Code for Information Interchange (Американский стандартный код для обмена информацией). Почти все компьютерные операционные системы поддерживают код ASCII, хотя многие также поддерживают и другие национальные наборы символов.

Базовые типы переменных, используемые в программах C++, представлены в табл. 3.1. В ней также приведены обычные размеры переменных указанных типов и предельные значения, которые могут храниться в этих переменных. Вы можете сверить результаты работы программы, представленной в листинге 3.1, с содержимым табл. 3.1.

Таблица 3.1. Типы переменных

Тип	*Размер, байт*	*Значения*
`bool`	1	`true` или `false`
`unsigned short int`	2	0–65 535
`short int`	2	-32 768–32 767
`unsigned long int`	4	0–4 294 967 295
`long int`	4	-2 147 483 648–2 147 483 647
`int` (16 разрядов)	2	-32 768–32 767
`int` (32 разряда)	4	-2 147 483 648–2 147 483 647
`unsigned int` (16 разрядов)	2	0–65 535
`unsigned int` (32 разряда)	4	0–4 294 967 295
`char`	1	256 значений символов
`float`	4	1,2e-38–3,4e38
`double`	8	2,2e-308–1,8e308

В зависимости от версии компилятора и марки компьютера, размеры переменных могут отличаться от приведенных в табл. 3.1. Если результаты, полученные на вашем компьютере, совпадают с теми, что приведены после листинга 3.1, значит, табл. 3.1 применима к вашему компьютеру. В противном случае вам следует обратиться к документации, прилагаемой к компилятору, чтобы получить информацию о значениях, которые могут хранить переменные разных типов в вашей компьютерной системе.

Определение переменной

Чтобы создать или определить переменную, нужно указать ее тип, за которым (после одного или нескольких пробелов) должно следовать ее имя, завершающееся точкой с запятой. Для имени переменной можно использовать практически любую комбинацию букв, но оно не должно содержать пробелов, например: `x`, `J23qrsnf` и `myAge`. Хорошими считаются имена, позволяющие судить о назначении переменных, ведь удачно подобранное имя способно облегчить понимание работы программы в целом. В следующем выражении определяется целочисленная переменная с именем `myAge`:

```
int myAge;
```

При объявлении переменной для нее выделяется (резервируется) память. Резервирование памяти не очищает ячейки от значений, которые ранее в них хранились, поэтому если за объявлением переменной не следует ее инициализация, то текущее значение этой переменной будет непредсказуемым, а не нулевым, как думают многие. Далее вы узнаете, как инициализировать переменную (другими словами, присвоить ей новое значение).

Уважающие себя программисты стремятся избегать таких нечитабельных имен переменных, как `J23qrsnf`, а однобуквенные имена (например, `x` или `i`) используют только для временных переменных, таких как счетчики циклов. Старайтесь использовать как можно более информативные имена, например `myAge` или `howMany`. Такие имена даже три недели спустя помогут вам вспомнить, что вы имели в виду, когда писали те или иные программные строки.

Поставьте следующий эксперимент. Опираясь лишь на первые пять строк программы, попробуйте догадаться, для чего предназначены объявленные ниже переменные.

Пример 1:

```
int main()
{
    unsigned short x;
    unsigned short y;
    unsigned short z;
    z = x * y;
    return 0;
}
```

Пример 2:

```
int main ()
{
     unsigned short Width;
     unsigned short Length;
     unsigned short Area;
     Area = Width * Length;
     return 0;
}
```

Если вы скомпилируете эту программу, компилятор выведет предупреждение о том, что эти переменные не инициализированы. Чуть ниже вы увидите, как решается эта проблема.

Очевидно, что о назначении переменных во втором примере догадаться легче, и неудобство, связанное с необходимостью вводить более длинные имена переменных, впоследствии окупится с лихвой, потому что вам не придется ломать голову, для чего предназначена та или иная переменная.

Чувствительность к регистру букв

Язык C++ чувствителен к регистру букв. Другими словами, прописные и строчные буквы считаются разными буквами. Переменные с именами `age`, `Age` и `AGE` рассматриваются как три различные переменные.

Некоторые компиляторы позволяют отключать чувствительность к регистру букв. Лучше этого не делать, ведь ваши программы не смогут работать с другими компиляторами и другие программисты будут введены в заблуждение такой программой.

Существуют различные соглашения по поводу принципов подбора имен переменным. Хотя не так уж важно, каких принципов будете придерживаться вы, желательно оставаться верными им по крайней мере на протяжении работы над одним проектом.

Многие программисты предпочитают записывать имена переменных только строчными буквами. Если для имени требуется два слова (например, `my car`), то в соответствии с самыми популярными соглашениями используются варианты `my_car` или `myCar`. Последняя форма записи называется "верблюжьим представлением", поскольку одна прописная буква посередине слова напоминает горб верблюда.

Одни считают, что имена переменных с символом подчеркивания внутри слова (`my_car`) читаются легче. Другим не нравится этот подход, потому что он якобы вызывает трудности при вводе. В этой книге отдельные слова в составных именах переменных начинаются с прописной буквы: `myCar`, `theduickBrownFox` и т.д. Но это, конечно, ни к чему вас не обязывает, и вы можете использовать другие подходы при выборе имен.

ПРИМЕЧАНИЕ

Многие профессиональные программисты применяют так называемый *венгерский стиль записи переменных*. Идея состоит в том, что каждая переменная должна иметь префикс, указывающий на ее тип. Так, имена целочисленных переменных (типа `int`) должны начинаться со строчной буквы *i*, длинные целые (типа `long int`) — со строчной буквы *l*. Соответствующими префиксами должны быть помечены константы, глобальные переменные, указатели и другие объекты. Однако это имеет более важное значение в программировании на языке C, чем на C++, поскольку последний поддерживает создание нестандартных типов, или типов, определенных пользователем (подробнее об этом см. *занятие 6*), а также потому, что в языке C++ установлен более строгий контроль за типами данных.

Ключевые слова

Некоторые слова изначально зарезервированы в языке C++ и поэтому их нельзя использовать в качестве имен переменных. Такие слова называются ключевыми и используются компилятором для управления программой. В их число входят `if`, `while`, `for` и `main`. В технической документации компилятора должен быть полный список всех зарезервированных слов. Типичный набор ключевых слов языка C++ приведен в приложении Б.

Рекомендуется

Указывайте тип переменной перед именем при ее определении.

Используйте для переменных информативные имена.

Помните, что в языке C++ различаются прописные и строчные буквы.

Уточните, сколько байтов занимает в памяти каждый тип переменной на вашем компьютере и какие значения могут храниться в переменных этого типа.

Не рекомендуется

Не используйте ключевые слова в качестве имен переменных.

Не присваивайте беззнаковым переменным отрицательные числа.

Создание нескольких переменных одного типа

В языке C++ предусмотрена возможность создания в строке программы сразу нескольких переменных одного типа. Для этого следует указать тип, за которым перечисляются имена переменных, разделенные запятыми. Например:

```
unsigned int myAge, myWeight; // две переменные типа unsigned int
long int area, width, length; // три переменные типа long int
```

В данном примере обе переменные, `myAge` и `myWeight`, объявлены как беззнаковые целочисленные. Во второй строке объявляются три переменные с именами `area`, `width` и `length`. Всем этим переменным присваивается один и тот же тип (`long`), поэтому в одной строке определения переменных нельзя смешивать разные типы.

Присваивание значений переменным

С этой целью используется оператор присваивания (=). Так, чтобы присвоить число 5 переменной Width, запишите следующее:

```
unsigned short Width;
Width = 5;
```

Тип long — это сокращенное название типа long int, а short — сокращенное название типа short int.

Эти две строки можно объединить в одну и инициализировать переменную Width в *процессе определения*:

```
unsigned short Width = 5;
```

Инициализация напоминает присваивание, особенно в случае инициализации целочисленных переменных. Ниже, при рассмотрении констант, вы узнаете, что некоторые значения обязательно должны быть инициализированы, поскольку по отношению к ним нельзя выполнять операцию присваивания. Существенное отличие инициализации от присваивания состоит в том, что она происходит в момент создания переменной.

Подобно тому как можно определять сразу несколько переменных, можно и инициализировать сразу несколько переменных при их создании. Например:

```
// создаем две переменных типа long и инициализируем их
long width = 5, length = 7;
```

В этом примере переменная width типа long int была инициализирована значением 5, а переменная length того же типа — значением 7. При определении нескольких переменных в одной строке, инициализировать можно только некоторые из них:

```
int myAge = 39, yourAge, hisAge = 40;
```

В этом примере создаются три переменных типа int, а инициализируются только первая и третья.

В листинге 3.2 показана программа, полностью готовая к компиляции. В ней вычисляется площадь прямоугольника, после чего результат выводится на экран.

Листинг 3.2. Демонстрация использования переменных

```
1: // Демонстрация использования переменных
2: #include <iostream.h>
3:
4: int main()
5: {
6:    unsigned short int Width = 5, Length;
7:    Length = 10;
8:
9:    // создаем переменную Area типа unsigned short и инициализируем ее
10:   // результатом умножения значений переменных Width на Length
```

```
11:  unsigned short int Area = (Width * Length);
12:
13:  cout << "Width:" << Width << "\n";
14:  cout << "Length: " << Length << endl;
15:  cout << "Area: " << Area << endl;
16:  return 0;
17: }
```

РЕЗУЛЬТАТ

```
Width:5
Length: 10
Area: 50
```

АНАЛИЗ В строке 2 содержится директива препроцессора `include`, включающая библиотеку `iostream`, которая обеспечивает работоспособность объекта вывода `cout`. Собственно, программа начинает свою работу в строке 4.

В строке 6 переменная `Width` определяется для хранения значения типа `unsigned short int`, и тут же выполняется инициализация этой переменной числом `5`. В этой же строке определяется еще одна переменная `Length` такого же типа, но без инициализации. В строке 7 переменной `Length` присваивается значение `10`.

В строке 11 определяется переменная `Area` типа `unsigned short int`, которая тут же инициализируется значением, полученным в результате умножения значений переменных `Width` и `Length`. В строках 13–15 значения всех переменных программы выводятся на экран. Обратите внимание на то, что для разрывов строк используется специальный оператор `endl`.

Ключевое слово typedef

Порой утомительно и скучно многократно повторять запись таких ключевых слов, как `unsigned short int`. (Кроме того, в этих трех словах немудрено наделать еще и кучу ошибок.) В языке C++ предусмотрена возможность создания псевдонима для этой фразы путем использования ключевого слова `typedef`, которое означает определение типа.

При создании псевдонима важно отличать его от создания нового типа (об этом пойдет речь на занятии 6). Чтобы создать псевдоним типа данных, сначала записывается ключевое слово `typedef`, за которым следует существующий тип, а за ним новое имя с символом точки с запятой. Например, при выполнении строки

```
typedef unsigned short int USHORT;
```

создается новое имя `USHORT`, которое можно использовать везде, где нужно определить переменную типа `unsigned short int`. Листинг 3.3 переделан из листинга 3.2 с использованием псевдонима `USHORT` вместо слов *unsigned short int*.

Листинг 3.3. Пример определения типа с помощью `typedef`

```
1: // *****************
2: // Пример определения типа с помощью typedef
3: #include <iostream.h>
4:
```

```
5: typedef unsigned short int USHORT; //определение псевдонима
6:
7: int main()
8: {
9:    USHORT Width = 5;
10:   USHORT Length;
11:   Length = 10;
12:   USHORT Area = Width * Length;
13:   cout << "Width:" << Width << "\n";
14:   cout << "Length: " << Length << endl;
15:   cout << "Area: " << Area << endl;
16:   return 0;
17: }
```

```
Width:5
Length: 10
Area: 50
```

В строке 5 идентификатор `USHORT` с помощью ключевого слова `typedef` определен как псевдоним типа `unsigned short int`. В остальном эта программа аналогична предыдущей, представленной в листинге 3.2, да и результаты работы обеих программ совпадают.

В каких случаях следует использовать типы short и long

Начинающим программистам часто бывает трудно принять решение о том, когда объявлять переменную с использованием типа `long`, а когда — с использованием типа `short`. Правило довольно простое: если существует хоть малейший шанс, что ваше значение будет слишком большим для предполагаемого типа, используйте тип с большим размером.

Приведенные в табл. 3.1 переменные типа `unsigned short int`, как правило, имеют размер, равный двум байтам, и могут хранить значение, не превышающее 65 535. Знаковые короткие целые делят свой диапазон между положительными и отрицательными числами, поэтому их максимальное значение вдвое меньше, чем у беззнакового короткого целого.

Хотя переменные типа `unsigned long int` могут хранить очень большое число (4 294 967 295), оно все-таки конечно. Если вам нужно работать с еще большими числами, придется перейти к использованию типов `float` или `double`, но при этом вы несколько проиграете в точности. Переменные типа `float` и `double` могут хранить чрезвычайно большие числа, но на большинстве компьютеров значимыми остаются только первые 7 или 19 цифр, т.е. после указанного количества цифр число округляется.

Переменные с меньшим размером используют меньший объем памяти. В наши дни память становится все дешевле, а жизнь не так уж длинна, чтобы тратить ее на экономию памяти. Поэтому отдайте предпочтение типу `int`, который на большинстве компьютеров имеет размер в четыре байта.

Переполнение беззнаковых целых

Что случится, если при использовании беззнаковых длинных целых превысить их предельный максимум?

Когда беззнаковое целое достигает своего максимального значения, при очередном инкременте оно сбрасывается в нуль и отсчет начинается сначала, как в автомобильном одометре. В листинге 3.4 показано, что произойдет при попытке поместить слишком большое число в переменную типа short.

Листинг 3.4. Пример переполнения беззнаковой целой переменной

```
1: #include <iostream.h>
2: int main()
3: {
4:    unsigned short int smallNumber;
5:    smallNumber = 65535;
6:    cout << "small number:" << smallNumber << endl;
7:    smallNumber++;
8:    cout << "small number:" << smallNumber << endl;
9:    smallNumber++;
10:   cout << "small number:" << smallNumber << endl;
11:   return 0;
12: }
```

РЕЗУЛЬТАТ

```
small number:65535
small number:0
small number:1
```

АНАЛИЗ

В строке 4 объявляется переменная smallNumber типа unsigned short int, которая на моем компьютере является двухбайтовой, способной хранить значение между 0 и 65 535. В строке 5 переменной smallNumber присваивается максимальное значение, которое в строке 6 выводится на экран.

В строке 7 переменная smallNumber увеличивается на 1. Приращение осуществляется с помощью оператора инкремента, имеющего вид двух символов плюс (++). Следовательно, значение переменной smallNumber должно стать 65 536. Однако переменная типа unsigned short int не может хранить число, большее 65 535, поэтому ее значение сбрасывается в 0, который и выводится в строке 8.

В строке 9 переменная smallNumber вновь увеличивается на единицу, после чего ее новое значение выводится на экран.

Переполнение знаковых целочисленных значений

Знаковые целые отличаются от беззнаковых тем, что половина этих значений всего диапазона — отрицательные числа. При выходе за пределы максимального положительного значения переменная принимает минимальное отрицательное значение. В листинге 3.5 показано, что происходит, если добавить единицу к максимальному положительному числу, хранящемуся в переменной типа short int.

Листинг 3.5. Пример переполнения знаковой целой переменной

```
1: #include <iostream.h>
2: int main()
3: {
4:    short int smallNumber;
5:    smallNumber = 32767;
6:    cout << "small number:" << smallNumber << endl;
7:    smallNumber++;
8:    cout << "small number:" << smallNumber << endl;
9:    smallNumber++;
10:   cout << "small number:" << smallNumber << endl;
11:   return 0;
12: }
```

РЕЗУЛЬТАТ

```
small number:32767
small number:-32768
small number:-32767
```

АНАЛИЗ В строке 4 переменная `smallNumber` объявляется на этот раз короткой целой (`short int`) со знаком (если в объявлении переменной ключевое слово `unsigned` отсутствует, т.е. эта переменная явно не объявляется беззнаковой, то подразумевается ее использование со знаком). В остальном эта программа выполняет те же действия, что и предыдущая, но на экран выводятся совсем другие результаты. Чтобы до конца понять, почему получены именно такие результаты, нужно знать, как представляются числа со знаком в двухбайтовом целом значении.

Этот пример показывает, что в случае приращения максимального положительного целого числа со знаком будет получено не нулевое значение (как в случае с беззнаковыми целыми), а минимальное отрицательное число.

Символы

Символьные переменные (типа `char`) обычно занимают один байт, этого достаточно для хранения 256 значений печатаемых символов (см. приложение В). Значение типа `char` можно интерпретировать как число в диапазоне 0–255, или символ ASCII. Набор символов ASCII и его эквивалент ISO (International Standards Organization — Международная организация по стандартизации) представляют собой способ кодировки всех букв, цифр и знаков препинания.

Например, в коде ASCII английской строчной букве "a" присвоено значение 97. Всем прописным и строчным буквам, всем цифрам и знакам препинания присвоены значения от 1 до 128. Дополнительные 128 знаков и символов зарезервированы для расширения возможностей компьютера, хотя расширенный набор символов IBM стал уже чем-то вроде стандарта.

ПРИМЕЧАНИЕ ASCII обычно произносится как "аскей".

Компьютеры не имеют ни малейшего понятия ни о каких буквах, знаках препинания или предложениях. Все они понимают только числа. В действительности же они оценивают некоторые электрические параметры в определенных точках своих схем. Если значение оцениваемого параметра выше некоторой оговоренной величины, оно представляется внутренне как 1, если нет — как 0. Путем группирования нулей и единиц компьютер способен генерировать кодовые комбинации, которые можно интерпретировать как числа, а те, в свою очередь, можно присвоить буквам и знакам препинания.

Символы и числа

Если разместить какой-нибудь символ, например "a", в переменной типа `char`, то в действительности она будет хранить число, лежащее в диапазоне между 0 и 255. Однако компилятор знает, как переходить от символов к их цифровым эквивалентам в ASCII и обратно.

Взаимосвязь между числом и буквой произвольна, поскольку нет никакой весомой причины для присваивания строчной букве "a" именно значения 97. Если все составляющие компьютера (ваша клавиатура, компилятор и экран) с этим "согласны", никаких проблем не возникнет. Однако важно понимать, что между значением 5 и символом "5" большая разница. Символу "5" в действительности соответствует значение 53, так же как букве "a" соответствует число 97.

Листинг 3.6. Вывод на экран символов по их значениям

```
1: #include <iostream.h>
2: int main()
3: {
4:   for (int i = 32; i<128; i++)
5:     cout << (char) i;
6:   return 0;
7: }
```

```
!"#$%'()*+,./0123456789:;<>?@ABCDEFGHIJKLMNOP
_QRSTUVWXYZ[\]^`abcdefghijklmnopqrstuvwxyz{|{~
```

Эта простая программа выводит символы, значения которых лежат в диапазоне 32—127.

Специальные символы

Компилятор C++ распознает некоторые специальные символы, предназначенные для форматирования текста. (Самые распространенные из них представлены в табл. 3.2.) Чтобы вставить эти символы в программу, используется обратный слеш (называемый символом начала управляющей последовательности), указывающий, что следующий за ним символы является управляющими. Следовательно, чтобы вставить в программу символ табуляции, нужно ввести одиночную кавычку, обратный слеш, букву t и снова одиночную кавычку:

```
char tabCharacter = '\t';
```

В этом примере объявляется переменная типа char (с именем tabCharacter), которая тут же инициализируется символьным значением \t, распознаваемым как символ табуляции. Специальные символы форматирования используются при выводе информации на экран, в файл или на другое устройство вывода (например, принтер).

Символ начала управляющей последовательности изменяет значение символа, который следует за ним. Например, обычно символ *n* означает букву *n*, но когда перед ней стоит символ начала управляющей последовательности (\), то он превращается в символ разрыва строки.

Таблица 3.2. Управляющие символы

Символ	*Значение*
\n	Разрыв строки
\t	Табуляция
\b	Возврат на одну позицию
\"	Двойная кавычка
\'	Одиночная кавычка
\?	Вопросительный знак
\\	Обратный слеш

Константы

Подобно переменным, константы представляют собой ячейки памяти, предназначенные для хранения данных. Но, в отличие от переменных, константы не изменяются (о чем говорит само название — *константа*). Создаваемую константу нужно инициализировать, поскольку позже ей нельзя присвоить новое значение.

В языке C++ предусмотрено два типа констант: литеральные и символьные.

Литеральные константы

Литеральная константа — это значение, непосредственно вводимое в самой программе. Например, в выражении

```
int myAge = 39;
```

myAge является переменной типа int, а число 39 — литеральной константой. Нельзя присвоить никакое значение константе 39.

Символьные константы

Символьная константа — это константа, представленная именем (точно так же, как именем представляется любая переменная). Однако, в отличие от переменной, значение инициализированной константы изменить нельзя.

Если в вашей программе есть одна целочисленная переменная с именем students, а другая — с именем classes, вы могли бы вычислить общее количество учеников школы при условии, что вам известно, сколько классов в школе и сколько учеников в каждом классе (допустим, каждый класс состоит из 15 учеников):

```
students = classes * 15;
```

Символ * означает умножение.

В этом примере число 15 является литеральной константой. Но если эту литеральную константу заменить символьной, то вашу программу будет легче читать и изменять в будущем:

```
students = classes * studentsPerClass
```

Если впоследствии потребуется изменить количество учеников в каждом классе, вы сможете сделать это единожды в той строке программы, где определяется константа studentsPerClass, и вам не придется вносить изменения во все строки программы, где используется это значение.

В языке C++ существует два способа объявления символьной константы. Традиционный и ныне уже устаревший способ состоит в использовании директивы препроцессора #define.

Определение констант с помощью директивы #define

Для определения константы традиционным способом введите следующее выражение:

```
#define studentsPerClass 15
```

Обратите внимание на то, что константа studentsPerClass не имеет никакого конкретного типа (int, char и т.д.). Директива #define выполняет простую текстовую подстановку. Каждый раз, когда препроцессор встречает слово studentsPerClass, он заменяет его литералом 15.

Поскольку препроцессор запускается перед компилятором, последний никогда не увидит константу, а будет видеть только число 15.

Определение констант с помощью ключевого слова const

Хотя директива #define и справляется со своими обязанностями, в языке C++ существует новый, более удобный способ определения констант:

```
const unsigned short int studentsPerClass = 15;
```

В этом примере также объявляется символическая константа с именем studentsPerClass, но на сей раз для этой константы задается тип unsigned short int. Этот способ имеет несколько преимуществ, облегчая дальнейшую поддержку вашей программы и предотвращая появление некоторых ошибок. Самое важное отличие этой константы от предыдущей (объявленной с помощью директивы #define) состоит в том, что она имеет тип и компилятор может проследить за ее использованием только по назначению (т.е. в соответствии с объявленным типом).

Во время работы программы константы изменять нельзя. Если же возникла необходимость в изменении, например, константы studentsPerClass, вам нужно изменить соответствующее объявление в программе и перекомпилировать ее.

Рекомендуется	Не рекомендуется
Следите, чтобы значения переменных не превышали допустимый предел. **Присваивайте** переменным осмысленные имена, отражающие их назначение. **Используйте** типы `short` и `long`, чтобы более эффективно управлять памятью компьютера.	**Не используйте** в качестве имен переменных ключевые слова.

Константы перечислений

Перечисления позволяют создавать новые типы данных, а затем определять переменные этих типов, значения которых ограничены набором константных значений. Например, можно объявить `COLOR` как перечисление и определить для него пять значений: `RED`, `BLUE`, `GREEN`, `WHITE` и `BLACK`.

Для создания перечисления используется ключевое слово `enum`, за которым следуют: имя типа, открывающая фигурная скобка, список константных значений, разделенных запятыми, закрывающая фигурная скобка и точка с запятой. Например:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

Это выражение выполняет две задачи.

1. Создается перечисление с именем `COLOR`, являющееся новым типом.
2. Определяются символьные константы: `RED` со значением `0`; `BLUE` со значением `1`; `GREEN` со значением `2` и т.д.

Каждой константе перечисления соответствует определенное целочисленное значение. По умолчанию первая константа инициализируется значением `0`, а каждая следующая — значением, на единицу большим предыдущего. Однако любую константу можно инициализировать произвольным значением, и в этом случае явно неинициализированные константы продолжат возрастающую последовательность, взяв за точку отсчета значение, стоящее перед ними. Таким образом, если записать

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

то константа `red` будет иметь значение `100`; константа `blue` — `101`; константа `GREEN` — `500`; константа `WHITE` — `501`; константа `BLACK` — `700`.

Теперь можно определить переменные типа `COLOR`, но каждой из них можно присвоить только одно из перечислимых значений (в данном случае `RED`, `BLUE`, `GREEN`, `WHITE` или `BLACK` либо `100`, `101`, `500`, `501` или `700`). Переменной `COLOR` можно присвоить любое значение цвета. На самом деле этой переменной можно присвоить любое целое значение, даже если оно не соответствует ни одному из разрешенных цветов, но в этом случае приличный компилятор должен возмутиться и показать предупреждающее сообщение. Важно понимать, что переменные перечисления на самом деле имеют тип `unsigned int` и целочисленным переменным присваиваются заданные константы перечисления. Однако иногда при работе с цветами, днями недели или другими подобными наборами значений неплохо иметь возможность называть эти значения по имени. В листинге 3.7 представлена программа, в которой используется тип перечисления.

Листинг 3.7. Использование перечисления

```
1: #include <iostream.h>
2: int main()
3: {
4:    enum Days { Sunday, Monday, Tuesday,
5:      Wednesday, Thursday, Friday, Saturday };
6:    int choice;
7:    cout << "Enter a day (0-6): ";
8:    cin << choice;
9:    if (choice = Sunday || choice == Saturday)
10:     cout << "\nYou're already off on weekends!\n";
11:   else
12:     cout << "\nOkay, I'll put in the vacation day.\n";
13:   return 0;
14: }
```

РЕЗУЛЬТАТ

```
Enter a day (0-6): 6
You're already off on weekends!
```

АНАЛИЗ

В строке 4 определяется перечисление DAYS с семью константными значениями. Все они образуют возрастающую последовательность чисел, начиная с нуля; таким образом, значение вторника (Tuesday) равно 2.

Пользователю предлагается ввести значение между 0 и 6. Он не может ввести слово Sunday, поскольку в программе не предусмотрен перевод символов в значение перечисления. Но можно проверить введенное пользователем значение, сравнив его с константными значениями перечисления, как показано в строке 9. Использование перечисления облегчает анализ программы. Того же эффекта можно добиться, используя константы целочисленного типа, как показано в листинге 3.8.

ПРИМЕЧАНИЕ

Для этой и всех небольших программ в данной книге я намеренно не предусматривал включения ряда выражений, которые обычно пишутся для обработки ситуаций, связанных с приемом от пользователя неверных данных. Например, в этой программе отсутствует проверка вводимых чисел, которая должна обязательно присутствовать в настоящей программе и предназначена для предупреждения ввода неразрешенных чисел. Это было сделано намеренно, для того чтобы сэкономить место в книге и сосредоточить ваше внимание только на рассматриваемой в данном разделе теме.

Листинг 3.8. Та же программа, но с использованием констант целочисленного типа

```
1: #include <iostream.h>
2: int main()
3: {
4:    const int Sunday = 0;
5:    const int Monday = 1;
6:    const int Tuesday = 2;
7:    const int Wednesday = 3;
```

```
8:   const int Thursday = 4;
9:   const int Friday = 5;
10:  const int Saturday = 6;
11:
12:  int choice;
13:  cout << "Enter a day (0-6): ";
14:  cin << choice;
15:
16:  if (choice = Sunday || choice == Saturday)
17:       cout << "\nYou're already off on weekends!\n";
18:  else
19:       cout << "\nOkay, I'll put in the vacation day.\n";
20:
21:  return 0;
22: }
```

РЕЗУЛЬТАТ

```
Enter a day (0-6): 6
You're already off on weekends!
```

АНАЛИЗ Результаты работы этой программы идентичны результатам программы из листинга 3.7. Но в этом варианте все константы (`Sunday`, `Monday` и пр.) определены в явном виде и отсутствует тип перечисления `Days`. Обратите внимание, что программа с перечислением короче и логичнее.

Резюме

На этом занятии рассматривались числовые и символьные переменные и константы, которые в C++ используются для хранения данных во время выполнения программы. Числовые переменные могут быть либо целыми (`char`, `short` и `long int`), либо вещественными (`float` и `double`). Кроме того, они могут быть знаковыми и беззнаковыми (`unsigned`). Хотя на различных компьютерах все эти типы могут иметь разные размеры, но на одном компьютере переменные одного и того же типа всегда имеют постоянный размер.

Переменную нужно объявить до ее использования. При работе с данными необходимо следить, чтобы тип данных соответствовал объявленному типу переменной. Если, например, поместить слишком большое число в целую переменную, возникнет переполнение, которое приведет к неверному результату.

Кроме того, вы познакомились с литеральными, символьными и перечислимыми константами, а также с двумя способами объявления символьных констант: с помощью директивы `#define` и ключевого слова `const`.

Вопросы и ответы

Если переменные типа `short int` могут преподнести сюрприз в виде переполнения памяти, то почему бы не использовать во всех случаях переменные типа `long int`?

Как короткие (`short`), так и длинные (`long`) целочисленные переменные могут переполняться, но для того чтобы это произошло с типом `long int`, нужно уж слишком большое число. Например, переменная типа `unsigned short int` переполнится после

числа 65 535, в то время как переменная типа `unsigned long int` — только после числа 4 294 967 295. Однако на многих компьютерах каждое объявление длинного целого значения занимает вдвое больше памяти по сравнению с объявлением короткого целого (четыре байта против двух), и программа, в которой объявлено 100 таких переменных займет лишних 200 байт ОЗУ. Честно говоря, память сейчас перестала быть проблемой, поскольку большинство персональных компьютеров оснащено многими тысячами (если не миллионами) байтов памяти.

Что случится, если присвоить число с десятичной точкой целочисленной переменной, а не переменной типа `float`, как в следующей программной строке?

```
int aNumber = 5.4;
```

Хороший компилятор выдаст по этому поводу предупреждение, но такое присваивание совершенно правомочно. Присваиваемое число будет в таком случае усечено до целого. Следовательно, если присвоить число `5,4` целочисленной переменной, эта переменная получит значение `5`. Однако налицо потеря информации, и если затем хранящееся в этой целой переменной значение вы попытаетесь присвоить переменной типа `float`, то вещественной переменной придется "довольствоваться" лишь значением `5`.

Почему вместо литеральных констант лучше использовать символьные?

Если некоторое значение используется во многих местах программы, то применение символьной константы позволяет изменять все значения путем изменения одного лишь определения этой константы. Кроме того, смысл символьной константы проще понять по ее названию. Ведь иногда трудно разобраться, почему некоторое значение умножается на число 360, а если это число будет заменено символьной константой `degreesInACircle` (*градусов в окружности*), то сразу становится ясно, о чем идет речь.

Что произойдет, если присвоить отрицательное число беззнаковой переменной?

Рассмотрите следующую строку программы:

```
unsigned int aPositiveNumber = -1;
```

Хороший компилятор отреагирует на это предупреждением, но такое присвоение вполне законно. Отрицательное число будет представлено в беззнаковом побитовом виде. Так, побитовое представление числа `-1` выглядит как `11111111 11111111` (`0xFF` в шестнадцатеричном формате) и соответствует числу `65 535`. Если вам что-то непонятно, обратитесь к *приложению В*.

Можно ли работать с языком C++, не имея представления о побитовом выражении числовых значений и арифметике двоичных и шестнадцатеричных чисел?

Можно, но эффективность работы снизится. Мощь языка C++ состоит не в том, чтобы уберечь вас от ненужных деталей работы компьютера, а в том, чтобы заставить компьютер работать с максимальной отдачей. В этом основное отличие C++ от других языков программирования. Программисты, которые не имеют представления, как работать с двоичными значениями, часто обескуражены получаемыми результатами.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. В чем разница между целочисленной и вещественной (с плавающей точкой) переменными?
2. Каково различие между типами `unsigned short int` и `long int`?
3. Каковы преимущества использования символьной константы вместо литерала?
4. Каковы преимущества использования ключевого слова `const` вместо директивы `#define`?
5. Как влияет на работу программы "хорошее" или "плохое" имя переменной?
6. Если перечисление (`enum`) заданно таким образом, то каково значение его члена `Blue`?

```
enum COLOR { WHITE, BLACK = 100, RED, BLUE, GREEN = 300 };
```

7. Какие из следующих имен переменных можно считать хорошими, плохими или вообще недопустимыми?

   а) `Age`

   б) `!ex`

   в) `R79J`

   г) `TotalIncome`

   д) `_Invalid`

Упражнения

1. Какой тип переменной был бы правильным для хранения следующей информации?
   - Ваш возраст.
   - Площадь вашего заднего двора.
   - Количество звезд в галактике.
   - Средний уровень выпадения осадков за январь.
2. Создайте подходящие имена переменных для хранения этой информации.
3. Объявите константу для числа pi, равного 3.14159.
4. Объявите переменную типа `float` и инициализируйте ее, используя константу `pi`.

Выражения и операторы

Программа представляет собой набор команд, выполняемых в определенной последовательности. Современные программы сильны тем, что выполняются не последовательно, команда за командой от начала до конца программы, а по блокам. Каждый блок программы запускается в зависимости от выполнения заданного условия. Сегодня вы узнаете:

- Что такое операторы
- Что такое блоки
- Что такое выражения
- Как реализовать ветвление программы на основе результата выполнения заданного логического условия
- Что такое ИСТИННО и ЛОЖНО с точки зрения программиста на C++

Выражения

В языке C++ выражения управляют последовательностью выполнения других выражений, возвращают результаты вычислений или ничего не делают (нулевые выражения). Все выражения в C++ оканчиваются точкой с запятой. Нулевое выражение представляет собой просто точку с запятой. Наиболее простой пример выражения — это операция присвоения значения:

```
x = a + b;
```

В отличие от алгебры, это выражение не означает, что `x` равняется `a+b`. Данное выражение следует понимать так: присвоим результат суммирования значений переменных `a` и `b` переменной `x`, или присвоим переменной `x` значение `a+b`. Несмотря на то что в этом выражении выполняется сразу два действия — вычисление суммы и присвоение значения, после выражения устанавливается только один символ точки с запятой. Оператор (`=`) присваивает результаты операций, выполняемых над операндами, расположенными справа от знака равенства, операнду, находящемуся слева от него.

Символы пробелов

Символы пробелов, к которым относятся не только пробелы, но и символы табуляции и разрыва строки, в выражениях обычно игнорируются. Рассмотренное выше выражение можно записать так:

```
x=a+b;
```

или так:

```
x           =a
+      b    ;
```

Хотя последний вариант абсолютно правомочен, выглядит он довольно глупо. Символы пробелов можно использовать для улучшения читабельности программы, что облегчит работу с ней. Но при неумелом использовании эти же пробелы могут совершенно запутать программный код. Создатели C++ предоставили много различных возможностей, а уж насколько эффективно они будут использоваться, зависит от вас.

Символы пробелов не отображаются на экране и при печати — видны только различные отступы и промежутки между элементами текста.

Блоки и комплексные выражения

Иногда для облегчения восприятия программы логически взаимосвязанные выражения удобно объединять в комплексы, называемые блоками. Блок начинается открывающей фигурной скобкой ({) и оканчивается закрывающей фигурной скобкой (}). Хотя каждое выражение в блоке должно оканчиваться точкой с запятой, после символов открытия и закрытия блока точки с запятой не ставятся, как в следующем примере:

```
{
   temp = a;
   a = b;
   b = temp;
}
```

Этот блок выполняется как одно выражение, осуществляющее обмен значениями между переменными `a` и `b`.

Рекомендуется

Не забывайте о закрывающей фигурной скобке каждый раз, когда используется открывающая фигурная скобка.

Завершайте выражения в программе символом точки с запятой.

Не рекомендуется

Используйте разумно символы пробелов, чтобы сделать свою программу более понятной, а не наоборот.

Операции

Все, в результате чего появляется некоторое значение, в языке C++ называется *операцией*. Об операциях говорят, что они *возвращают* значение. Так, операция `3+2;` возвращает значение `5`. Все операции являются вместе с тем и выражениями.

Возможно, вы будете удивлены тому, что многие программные блоки рассматриваются как выражения. Приведем лишь три примера:

```
3.2                 // возвращает значение 3.2
PI                  // вещественная константа, которая возвращает значение 3.14
SecondsPerMinute    // целочисленная константа, которая возвращает 60
```

Предполагая, что PI — константа, равная 3.14, а SecondsPerMinute — константа, равная 60, *можно утверждать, что все три выражения являются операциями.*

Выражение

```
x = a + b;
```

не только складывает значения переменных a и b, но и присваивает результат переменной x, т.е. возвращает результат суммирования переменной x. Поэтому данное выражение вполне можно назвать операцией. Операции всегда располагаются справа от оператора присваивания:

```
y = x = a + b;
```

Данное выражение выполняет представленную ниже последовательность действий.

Прибавляем a к b.

Присваиваем результат выражения a + b переменной x.

Присваиваем результат выражения присваивания x = a + b переменной y.

Если переменные a, b, x и y являются целыми и если a имеет значение 2, а b — значение 5, то переменным x и y будет присвоено значение 7.

Пример выполнения некоторых выражений представлен в листинге 4.1.

Листинг 4.1. Выполнение сложных операций

```
1:  #include <iostream.h>
2:  int main()
3:  {
4:    int a=0, b=0, x=0, y=35;
5:    cout << "a: " << a << " b: " << b;
6:    cout << " x: " << x << " y: " << y << endl;
7:    a = 9;
8:    b = 7;
9:    y = x = a+b;
10:   cout << "a: " << a << " b: " << b;
11:   cout << " x: " << x << " y: " << y << endl;
12:   return 0;
13: }
```

Результат

```
a: 0 b: 0 x: 0 y: 35
a: 9 b: 7 x: 16 y: 16
```

АНАЛИЗ В строке 4 объявляются и инициализируются четыре переменные. Их значения выводятся в строках 5 и 6. В строке 7 переменной a присваивается значение 9. В строке 8 переменной b присваивается значение 7. В строке 9 значения переменных a и b суммируются, а результат присваивается переменной x. Результат операции x = a+b, в свою очередь, присваивается переменной y.

Операторы

Оператор — это литерал, который заставляет компилятор выполнять некоторое действие. Операторы воздействуют на операнды. Операндами в C++ могут быть как отдельные литералы, так и целые выражения. Язык C++ располагает двумя видами операторов:

- операторы присваивания;
- математические операторы.

Оператор присваивания

Оператор присваивания (=) позволяет заменить значение операнда, расположенного с левой стороны от знака равенства, значением, вычисляемым с правой стороны от него. Так, выражение

```
x = a + b;
```

присваивает операнду x значение, которое является результатом сложения значений переменных a и b.

Операнд, который может находиться слева от оператора присваивания, называется адресным операндом, или l-значением (от англ. слова *left*, т.е. *левый*). Операнд, который может находиться справа от оператора присваивания, называется операционным операндом, или r-значением (от англ. слова *right*, т.е. *правый*).

Константы могут быть только r-значениями и никогда не бывают адресными операндами, поскольку в ходе выполнения программы значения констант изменять нельзя. Так, можно записать:

```
x = 35;      // правильно
```

Но нельзя записать:

```
35 = x;      // ошибка!
```

Повторим: l-значение — это операнд, который может стоять в левой части выражения присваивания, а r-значение — операнд, который может стоять в правой части этого выражения. Обратите внимание, что все l-значения могут быть r-значениями, но не все r-значения могут быть l-значениями. Примером r-значения, которое не может быть l-значением, служит литеральная константа. Так, можно записать: x = 5;, но нельзя записать: 5 = x; (x может быть l- или r-значением, а 5 может быть только r-значением).

Математические операторы

В C++ используется пять математических операторов: сложения (+), вычитания (-), умножения (*), целочисленного деления (/) и деления по модулю (%).

В операциях сложения и вычитания разобраться несложно: они возвращают сумму и разность двух операндов. Хотя следует отметить, что вычитание беззнаковых целых

может привести к удивительным результатам, если полученная разность окажется отрицательным числом. Вы уже видели нечто подобное на прошлом занятии при описании переполнения переменных. В листинге 4.2 демонстрируется ситуация, когда из малого беззнакового числа вычитается большое беззнаковое число.

Листинг 4.2. Пример вычитания с переполнением целого числа

```
1: // Листинг 4.2. Пример вычитания с
2: // переполнением целого числа
3: #include <iostream.h>
4:
5: int main()
6: {
7:    unsigned int difference;
8:    unsigned int bigNumber = 100;
9:    unsigned int smallNumber = 50;
10:   difference = bigNumber - smallNumber;
11:   cout << "Difference is: " << difference;
12:   difference = smallNumber - bigNumber;
13:   cout << "\nNow difference is: " << difference <<endl;
14:   return 0;
15: }
```

РЕЗУЛЬТАТ

```
Difference is: 50
Now difference is: 4294967246
```

АНАЛИЗ Оператор вычитания используется в строке 10, а результат выводится на экран в строке 11, в данном случае вполне ожидаемый. В строке 12 вновь вызывается оператор вычитания, но на этот раз большое беззнаковое число вычитается из малого беззнакового числа. Результат должен быть отрицательным, но поскольку он вычисляется (и выводится) как беззнаковое число, происходит переполнение, о чем говорилось на прошлом занятии. Эта тема подробно рассматривается в приложении А.

Целочисленное деление и деление по модулю

Целочисленное деление несколько отличается от обычного. Целочисленное деление — это *то же самое* деление, которое вы изучали, когда ходили в первый класс. При делении числа 21 на число 4 (21/4) в случае целочисленного деления в ответе получается 5 и остаток 1.

Чтобы получить остаток, нужно число 21 разделить по модулю 4 (`21 % 4`), в результате получим остаток 1.

Операция деления по модулю иногда оказывается весьма полезной, например, если вы захотите вывести из ряда чисел каждое десятое значение. Любое число, результат деления которого по модулю 10 равен нулю, является кратным десяти, т.е. делится на 10 без остатка. Так, результат выражения 1 % 10 равен 1; 2 % 10 равен 2 и т.д.; а 10 % 10 равен 0. Результат от деления 11 % 10 снова равен1; 12 % 10 снова равен 2; и так можно продолжать до следующего числа, кратного 10, которым окажется 20. Мы воспользуемся этим методом при рассмотрении циклов на занятии 7.

Вопросы и ответы

При делении 5 на 3 я получаю в ответе 1. В чем моя ошибка?

При делении одного целого числа на другое в качестве результата вы также получите целое число. Следовательно, 5/3 равно 1.

Для получения дробного результата нужно использовать вещественные числа.

Выражение 5,0 / 3,0 даст дробный ответ: 1,66667.

Если ваш метод принимает в качестве параметров целочисленные значения, нужно *привести* их к типу `float`.

Выполняя операцию приведения типа переменной, вы заставляете компилятор изменить ее тип. При этом вы как будто говорите своему компилятору: "Я знаю, что делаю". Было бы неплохо, если бы это оказалось правдой, поскольку компилятор как бы отвечает вам: "Как скажете, босс: вся ответственность ложится на вас".

В данном случае мы хотим сказать компилятору: "Я понимаю, что ты считаешь это значение целым, но *я знаю, что делаю*: это действительно вещественное значение".

Для приведения типа существует два способа. Можно использовать приведение типа в старом стиле C или новый улучшенный оператор ANSI `static_cast`. Оба варианта демонстрируются в листинге 4.3.

Листинг 4.3. Приведение переменной к типу `float`

```
1: #include <iostream.h>
2:
3: void intDiv(int x, int y)
4: {
5:    int z = x / y;
6:    cout << "z: " << z << endl;
7: }
8:
9: void floatDiv(int x, int y)
10:{
11:   float a = (float)x;                  // старый стиль
12:   float b = static_cast<float>(y);     // современный стиль
13:   float c = a / b;
14:
15:   cout << "c: " << c << endl;
16:}
17:
18: int main()
19: {
20:   int x = 5, y = 3;
21:   intDiv(x,y);
22:   floatDiv(x,y);
23:   return 0;
24: }
```

```
z: 1
c: 1.66667
```

В строке 20 объявляются две целочисленные переменные. В строке 21 они как параметры пере даются функции `intDiv`, а в строке 22 — функции `floatDiv`. Вторая функция начинается со строки 9. В строках 11 и 12 целые значения приводятся к вещественному типу и присваиваются переменным типа `float`. Результат деления присваивается третьей переменной типа `float` в строке 13 и выводится на экран в строке 15.

Совместное использование математических операторов с операторами присваивания

Нет ничего необычного в том, чтобы к переменной прибавить некоторое значение, а затем присвоить результат той же переменной. Если у вас есть переменная `myAge` и вы хотите увеличить ее значение на два, можно записать следующее:

```
int myAge = 5;
int temp;
temp = myAge + 2;      // складываем 5 + 2 и результат помещаем в temp
myAge = temp;          // значение возраста снова помещаем в myAge
```

Однако этот метод грешит излишествами. В языке C++ можно поместить одну и ту же переменную по обе стороны оператора присваивания, и тогда предыдущий блок сведется лишь к одному выражению:

```
myAge = myAge + 2;
```

В алгебре это выражение рассматривалось бы как бессмысленное, но в языке C++ оно читается следующим образом: добавить два к значению переменной `myAge` и присвоить результат переменной `myAge`.

Существует еще более простой вариант предыдущей записи, хотя его труднее читать:

```
myAge += 2;
```

Этот оператор присваивания с суммой (`+=`) добавляет r-значение к l-значению, а затем снова записывает результат в l-значение. Если бы до начала выполнения выражения переменная `myAge` имела значение 4, то после ее выполнения значение переменной `myAge` стало бы равным 6.

Помимо оператора присваивания с суммой существуют также оператор присваивания с вычитанием (`-=`), делением (`/=`), умножением (`*=`) и делением по модулю (`%=`).

Инкремент и декремент

Очень часто в программах к переменным добавляется (или вычитается) единица. В языке C++ увеличение значения на 1 называется *инкрементом*, а уменьшение на 1 — *декрементом*. Для этих действий предусмотрены специальные операторы.

Оператор инкремента (++) увеличивает значение переменной на 1, а оператор декремента (--) уменьшает его на 1. Так, если у вас есть переменная C и вы хотите прирастить ее на единицу, используйте следующее выражение:

```
C++;          // Увеличение значения C на единицу
```

Это же выражение можно было бы записать следующим образом:

```
C = C + 1;
```

что, в свою очередь, равносильно выражению.

```
C += 1;
```

Префикс и постфикс

Как оператор инкремента, так и оператор декремента работает в двух вариантах: префиксном и постфиксном. Префиксный вариант записывается перед именем переменной (++myAge), а постфиксный — после него (myAge++).

В простом выражении вариант использования не имеет большого значения, но в сложном при выполнении приращения одной переменной с последующим присваиванием результата другой переменной это весьма существенно. Префиксный оператор вычисляется до присваивания, а постфиксный — после.

Семантика префиксного оператора следующая: инкрементируем значение, а затем считываем его. Семантика постфиксного оператора иная: считываем значение, а затем декрементируем оригинал.

На первый взгляд это может выглядеть несколько запутанным, но примеры легко проясняют механизм действия этих операторов. Если x — целочисленная переменная, значение которой равно 5, и, зная это, вы записали

```
int a = ++x;
```

то тем самым велели компилятору инкрементировать переменную x (сделав ее равной 6), а затем присвоить это значение переменной a. Следовательно, значение переменной a теперь равно 6 и значение переменной x тоже равно 6.

Если же, после этого вы записали

```
int b = x++;
```

то тем самым велели компилятору присвоить переменной b текущее значение переменной x (6), а затем вернуться назад к переменной x и инкрементировать ее. В этом случае значение переменной b равно 6, но значение переменной x уже равно 7. В листинге 4.4 продемонстрировано использование обоих типов операторов инкремента и декремента.

Листинг 4.4. Примеры использования префиксных и постфиксных операторов

```
1: // Листинг 4.4. Демонстрирует использование
2: // префиксных и постфиксных операторов
3: // инкремента и декремента
4: #include <iostream.h>
5: int main()
6: {
7:    int myAge = 39;    // инициализируем две целочисленные переменные
```

```
8:    int yourAge = 39;
9:    cout << "I am: " << myAge << " years old.\ n";
10:   cout << "You are: " << yourAge << " years old\ n";
11:   myAge++;       // постфиксный инкремент
12:   ++yourAge;     // префиксный инкремент
13:   cout << "One year passes...\ n";
14:   cout << "I am: " << myAge << " years old.\ n";
15:   cout << "You are: " << yourAge << " years old\ n";
16:   cout << "Another year passes\ n";
17:   cout << "I am: " << myAge++ << " years old.\ n";
18:   cout << "You are: " << ++yourAge << " years old\ n";
19:   cout << "Let's print it again.\ n";
20:   cout << "I am: " << myAge << " years old.\ n";
21:   cout << "You are: " << yourAge << " years old\ n";
22:   return 0;
23: }
```

РЕЗУЛЬТАТ

```
I am     39 years old
You are  39 years old
One year passes
I am     40 years old
You are  40 years old
Another year passes
I am     40 years old
You are  41 years old
Let's print it again
I am     41 years old
You are  41 years old
```

АНАЛИЗ

В строках 7 и 8 объявляются две целочисленные переменные и каждая из них инициализируется значением 39. Значения этих переменных выводятся в строках 9 и 10.

В строке 11 инкрементируется переменная myAge с помощью постфиксного оператора инкремента, а в строке 12 инкрементируется переменная yourAge с помощью префиксного оператора инкремента. Результаты этих операций выводятся в строках 14 и 15; как видите, они идентичны (обоим участникам нашего эксперимента по 40 лет).

В строке 17 инкрементируется переменная myAge (также с помощью постфиксного оператора инкремента), являясь при этом частью выражения вывода на экран. Поскольку здесь используется постфиксная форма оператора, то инкремент выполняется после операции вывода, поэтому снова было выведено значение 40. Затем (для сравнения с постфиксным вариантом) в строке 18 инкрементируется переменная yourAge с использованием префиксного оператора инкремента. Эта операция выполняется перед выводом на экран, поэтому отображаемое значение равно числу 41.

Наконец, в строках 20 и 21 эти же значения выводятся снова. Поскольку приращения больше не выполнялись, значение переменной myAge сейчас равно 41, как и значение переменной yourAge (все правильно: стареем мы все с одинаковой скоростью!).

Приоритеты операторов

Какое действие — сложение или умножение — выполняется первым в сложном выражении, например в таком, как это:

```
x = 5 + 3 * 8;
```

Если первым выполняется сложение, то ответ равен 8 * 8, или 64. Если же первым выполняется умножение, то ответ равен 5 + 24, или 29.

Каждый оператор имеет значение приоритета (полный список этих значений приведен в приложении А). Умножение имеет более высокий приоритет, чем сложение, поэтому значение этого "спорного" выражения равно 29.

Если два математических оператора имеют один и тот же приоритет, то они выполняются в порядке следования слева направо. Значит, в выражении

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

сначала вычисляется умножение, причем слева направо: `8*9 = 72` и `6*4 = 24`. Теперь то же выражение выглядит проще:

```
x = 5 + 3 + 72 + 24;
```

Затем выполняем сложение, тоже слева направо: `5 + 3 = 8; 8 + 72 = 80; 80 + 24 = 104`.

Однако будьте осторожны — не все операторы придерживаются этого порядка выполнения. Например, операторы присваивания вычисляются справа налево! Но что же делать, если установленный порядок приоритетов не отвечает вашим намерениям? Рассмотрим выражение:

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

Предположим, что в этом выражении вы не хотите умножать значение переменной `NumMinutesToType` на число `60`, а затем складывать результат со значением переменной `NumMinutesToThink`. Вам нужно сначала сложить значения двух переменных, чтобы получить общее число минут, а затем умножить это число на `60`, получив тем самым общее количество секунд.

В этом случае для изменения порядка выполнения действий, предписанного приоритетом операторов, нужно использовать круглые скобки. Элементы, заключенные в круглые скобки, имеют более высокий приоритет, чем любые другие математические операторы. Поэтому для реализации ваших намерений приведенное выше выражение нужно представить в таком виде:

```
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60
```

Вложение круглых скобок

При создании сложных выражений может возникнуть необходимость вложить круглые скобки друг в друга. Например, вам нужно вычислить общее число секунд, затем общее число включенных в рассмотрение людей, а уж потом перемножить эти числа:

```
TotalPersonSeconds = ( ( (NumMinutesToThink + NumMinutesToType) * 60) *
(PeopleInTheOffice + PeopleOnVacation) )
```

Это сложное выражение читается изнутри. Сначала значение переменной NumMinutesToThink складывается со значением переменной NumMinutesToType, поскольку они заключены во внутренние круглые скобки. Затем полученная сумма умножается на 60. После этого значение переменной PeopleInTheOffice прибавляется к значению переменной PeopleOnVacation. Наконец, вычисленное общее количество людей умножается на общее число секунд.

Этот пример затрагивает близкую и не менее важную тему. Приведенное выше выражение легко вычисляется компьютером, но нельзя сказать, что человеку его так же легко читать, понимать и модифицировать. Вот как можно переписать это выражение с помощью временных целочисленных переменных:

```
TotalMinutes = NumMinutesToThink + NumMinutesToType;
TotalSeconds = TotalMinutes * 60;
TotalPeople = PeopleInTheOffice + PeopleOnVacation;
TotalPersonSeconds = TotalPeople * TotalSeconds;
```

Для записи этого варианта требуется больше времени и много временных переменных, но он гораздо легче для понимания. Осталось лишь добавить комментарии, разъясняющие назначение этого программного кода, и заменить число 60 символьной константой. И тогда можно считать этот программный фрагмент практически идеальным для чтения и дальнейшей эксплуатации.

Рекомендуется

Помните, что выражения оперируют значениями.

Используйте префиксный оператор (++переменная) для инкремента или декремента переменной перед ее использованием в выражении.

Используйте постфиксный оператор (переменная++) для инкремента или декремента переменной после ее использования в выражении.

Используйте круглые скобки для изменения порядка выполнения операторов, обусловленного их приоритетами.

Не рекомендуется

Не используйте слишком много вложенных круглых скобок, поскольку такие выражения становятся трудными для понимания.

Что такое ИСТИННО

В предыдущих версиях языка C++ результаты логических выражений представлялись целочисленными значениями, но в новом стандарте ANSI введен новый тип — bool, имеющий только два возможных значения: true или false.

Любое выражение может быть рассмотрено с точки зрения его истинности или ложности. Математические выражения, возвращающие нуль, можно использовать для присвоения значения false логической переменной, а любой другой результат будет означать true.

Многие компиляторы и раньше были сориентированы на тип `bool`, который внутренне представлялся с помощью типа `long int` и поэтому имел размер, равный четырем байтам. Ныне ANSI-совместимые компиляторы часто обеспечивают однобайтовый тип `bool`.

Операторы отношений

Такие операторы используются для выяснения равенства или неравенства двух значений. Выражения сравнения всегда возвращают значения `true` (истина) или `false` (ложь). Операторы отношения представлены в табл. 4.1.

В новом стандарте ANSI предусмотрен новый тип `bool`, и все операторы отношений теперь возвращают значение типа `bool` — `true` и `false`. В предыдущих версиях языка C++ эти операторы возвращали `0` в качестве `false` или любое ненулевое значение (обычно `1`) в качестве `true`.

Если одна целочисленная переменная `myAge` содержит значение `39`, а другая целочисленная переменная `yourAge` — значение `40`, то, используя оператор равенства (`==`), можно узнать, равны ли эти переменные:

```
myAge == yourAge; // совпадает ли значение переменной myAge со значением переменной yourAge?
```

Это выражение возвращает `0`, или `false` (ложь), поскольку сравниваемые переменные не равны. Выражение

```
myAge > yourAge; // значение переменной myAge больше значения переменной yourAge?
```

также возвратит `0` (или `false`).

Многие начинающие программировать на языке C++ путают оператор присваивания (`=`) с оператором равенства (`==`). Случайное использование не того оператора может привести к такой ошибке, которую трудно обнаружить.

Всего в языке C++ используется шесть операторов отношений: равно (`==`), меньше (`<`), больше (`>`), меньше или равно (`<=`), больше или равно (`>=`) и не равно (`!=`). В табл. 4.1 не только перечислены все операторы отношений, но и приведены примеры их использования.

Рекомендуется

Помните, что операторы отношений возвращают значение `true` или `false`.

Не рекомендуется

Не путайте оператор присваивания (`=`) с оператором равенства (`==`). Это одна из самых распространенных ошибок программирования на языке C++ — будьте начеку!

Таблица 4.1. Операторы отношений

Имя	Оператор	Пример	Возвращаемое значение
Равно	==	100 == 50;	false
		50 == 50;	true
Не равно	!=	100 != 50;	true
		50 != 50;	false
Больше	>	100 > 50;	true
		50 > 50;	false
Больше или равно	>=	100 >= 50;	true
		50 >= 50;	true
Меньше	<	100 < 50;	false
		50 < 50;	false
Меньше или равно	<=	100 <= 50;	false
		50 <= 50;	true

Оператор if

Обычно программа выполняется по порядку, строка за строкой. Оператор `if` позволяет проверить условие (например, равны ли две переменные) и изменить ход выполнения программы, направив ее в другое русло, которое будет зависеть от результата сравнения.

Простейшая форма оператора `if` имеет следующий вид:

```
if (условие)
   выражение;
```

Условие в круглых скобках может быть любым выражением, но обычно оно содержит операторы отношений. Если это выражение возвращает `false`, то последующий оператор пропускается. Если же оно возвращает значение `true`, то оператор выполняется. Рассмотрим следующий пример:

```
if (bigNumber > smallNumber)
   bigNumber = smallNumber;
```

Здесь сравниваются значения переменных `bigNumber` и `smallNumber`. Если значение переменной `bigNumber` больше, то во второй строке этого программного фрагмента ее значение устанавливается равным значению переменной `smallNumber`.

Поскольку блок выражений, заключенных в фигурные скобки, эквивалентен одному выражению, то это свойство позволяет за строкой с оператором `if` использовать целые блоки, которые могут быть довольно обширными:

```
if (условие)
{
   выражение1;
```

```
    выражение2;
    выражение3;
}
```

Вот простой пример применения блока выражений:

```
if (bigNumber > smallNumber)
{
   bigNumber = smallNumber;
   cout << "bigNumber: " << bigNumber << "\ n";
   cout << "smallNumber: " << smallNumber << "\ n";
}
```

На этот раз, если значение переменной `bigNumber` больше значения переменной `smallNumber`, то большая переменная не только устанавливается равной значению меньшей переменной, но на экран выводятся также информационные сообщения. В листинге 4.5 показан еще один пример ветвления программы, основанный на использовании операторов отношений.

Листинг 4.5. Пример ветвления с использованием операторов отношений

```
1: // Листинг 4.5. Демонстрирует использование
2: // инструкции if совместно с операторами отношений
3: #include <iostream.h>
4: int main()
5: {
6:    int RedSoxScore, YankeesScore;
7:    cout << "Enter the score for the Red Sox: ";
8:    cin >> RedSoxScore;
9:
10:   cout << "\ nEnter the score for the Yankees: ";
11:   cin >> YankeesScore;
12:
13:   cout << "\ n";
14:
15:   if (RedSoxScore > YankeesScore)
16:      cout << "Go Sox!\ n";
17:
18:   if (RedSoxScore < YankeesScore)
19:   {
20:      cout << "Go Yankees!\ n";
21:      cout << "Happy days in New York!\ n";
22:   }
23:
24:   if (RedSoxScore == YankeesScore)
25:   {
26:      cout << "A tie? Naah, can't be.\ n";
27:      cout << "Give me the real score for the Yanks: ";
28:      cin >> YankeesScore;
29:
30:    if (RedSoxScore > YankeesScore)
```

```
31:         cout << "Knew it! Go Sox!";
32:
33:     if (YankeesScore > RedSoxScore)
34:         cout << "Knew it! Go Yanks!";
35:
36:     if (YankeesScore == RedSoxScore)
37:         cout << "Wow, it really was a tie!";
38:     }
39:
40:     cout << "\ nThanks for telling me.\ n";
41:     return 0;
42: }
```

РЕЗУЛЬТАТ

```
Enter the score for the Red Sox: 10
Enter the score for the Yankees: 10

A tie? Naah, can't be
Give me the real score for the Yanks: 8
Knew it! Go Sox!
Thanks for telling me..
```

АНАЛИЗ В этой программе пользователю предлагается ввести счет очков для двух бейсбольных команд. Введенные очки сохраняются в целочисленных переменных. Значения этих переменных сравниваются оператором `if` в строках 15, 18 и 24.

ПРЕДУПРЕЖДЕНИЕ

Многие начинающие программисты по невнимательности ставят точку с запятой после выражения с оператором `if`:

```
if(SomeValue < 10);
  SomeValue = 10;
```

В этом программном фрагменте было задумано сравнить значение переменной `SomeValue` с числом `10` и, если окажется, что оно меньше десяти, установить его равным этому числу, т.е. зафиксировать минимальное значение переменной `SomeValue` на уровне `10`. При выполнении этого программного фрагмента обнаруживается, что переменная `SomeValue` (вне зависимости от ее исходного значения) всегда устанавливается равной 10. В чем же дело? А дело в том, что оператор `if`, вернее, связанное с ним выражение сравнения, оканчивается точкой с запятой, создавая тем самым бездействующую инструкцию. Помните, что для компилятора отступ не играет никакой роли. Приведенный выше программный фрагмент можно переписать по-другому:

```
if(SomeValue < 10) // проверка
; // пустое выражение, контролируемое оператором if
SomeValue = 10; // присваивание
```

При удалении ненужной точки с запятой последняя строка этого фрагмента станет частью конструкции с оператором `if` и программа заработает в соответствии с намерением программиста.

Если очки одной команды больше очков другой, на экран выводится соответствующее сообщение. Если сравниваемые очки равны, программа выполняет блок выражений, который начинается в строке 25 и оканчивается в строке 38. В этом блоке снова запрашивается счет очков для команды из Нью-Йорка, после чего вновь выполняется сравнение результатов игры команд.

Обратите внимание: если начальный счет команды Yankees превышает счет команды Red Sox, то оператор `if` в строке 15 возвратит значение `false` и строка 16 не будет выполняться. Проверка же в строке 18 даст истинный результат (`true`) и будут выполнены выражения в строках 20 и 21. Затем с помощью оператора `if` в строке 24 будет проверено равенство очков; результат этого тестирования должен быть ложным (`false`) (иначе и быть не может, если проверка в строке 18 дала в результате значение `true`). В этом случае программа пропустит целый блок и перейдет сразу к выполнению строки 39.

В данном примере получение истинного результата одним оператором `if` не избавляет от выполнения проверок другими операторами `if`.

Использование отступов в программных кодах

Обратите внимание, как в листинге 4.4 используются отступы в конструкции с оператором `if`. Наверное, фанаты программирования могли бы развязать войну по поводу того, какой стиль выделения программных блоков лучше использовать. И хотя возможны десятки различных стилей, чаще других используются три перечисленных ниже.

- Начальная открывающая скобка располагается после условия, а закрывающая фигурная скобка, которая завершает блок операторов, выравнивается по одной линии с оператором `if`:

```
if (условие){
   выражение
}
```

- Фигурные скобки располагаются под словом `if`, выравниваясь по одной линии, а операторы блока записываются с отступом:

```
if (условие)
{
  выражение
}
```

- Отступ используется как для обеих фигурных скобок, так и для выражений блока:

```
if (условие)
{
 выражение
 }
```

Вариант, используемый в этой книге, отражает лишь пристрастие автора и ни к чему вас не обязывает.

Ключевое слово else

Довольно часто в программах требуется, чтобы при выполнении некоторого условия (т.е. когда это условие возвратит значение `true`) программа выполняла один блок команд, а при его невыполнении (т.е. когда это условие возвратит значение `false`) —

другой блок. В листинге 4.4 программист намеревался выводить на экран одно сообщение, если первая проверка (RedSoxScore > Yankees) возвращает значение true, и другое сообщение, если эта проверка возвращает значение false.

Показанный выше способ последовательного использования нескольких операторов if для проверки ряда условий прекрасно работает, но слишком громоздкий. Улучшить читабельность программы в подобных случаях можно с помощью ключевого слова else (листинг 4.6):

```
if (условие)
   выражение;
else
   выражение;
```

Листинг 4.6. Пример использования ключевого слова else

```
1:  // Листинг 4.6. Пример конструкции с ключевыми
2:  // словами if и else
3:  #include <iostream.h>
4:  int main()
5:  {
6:    int firstNumber, secondNumber;
7:    cout << "Please enter a big number: ";
8:    cin >> firstNumber;
9:    cout << "\ nPlease enter a smaller number: ";
10:   cin >> secondNumber;
11:   if (firstNumber > secondNumber)
12:     cout << "\ nThanks!\ n";
13:   else
14:     cout << "\ nOops. The second is bigger!";
15:
16:   return 0;
17: }
```

РЕЗУЛЬТАТ

```
Please enter a big number: 10

Please enter a smaller number: 12

Oops. The second is bigger!
```

АНАЛИЗ В строке 11 проверяется условие, заданное в операторе if. Если это условие истинно, будет выполнена строка 12, после чего работа программы завершится в строке 16. Если же это условие возвратит false, программа продолжит работу со строки 14. Если в строке 13 удалить ключевое слово else, строка 14 будет выполнена в любом случае, вне зависимости от выполнения условия. Но в данной конструкции if-else будет выполняться либо блок после if, либо блок после else.

Помните, что в конструкции if-else можно использовать не только отдельные выражения, но и целые блоки выражений, заключенных в фигурные скобки.

Оператор if

Ниже приводится синтаксис оператора if.

Форма 1:

```
if (условие)
выражение;
следующее выражение;
```

Если условие возвращает true, то выполняется выражение, а за ним и следующее выражение. Если условие возвратит false, то выражение игнорируется, а программа переходит к выполнению следующего выражения.

Помните, что вместо выражения может использоваться целый блок, заключенный в фигурные скобки.

Форма 2:

```
if (условие)
   выражение1;
else
   выражение2;
следующее выражение;
```

Если условие возвращает значение true, выполняется выражение1, в противном случае выполняется выражение2. После этого выполняется следующее выражение.

Пример:

```
if(SomeValue < 10);
  cout << "SomeValue is less than 10");
else
  cout << "SomeValue is not less than 10!");
cout << "Done." << endl;
```

Сложные конструкции с if

Нет никаких ограничений на использование любых операторов в блоках выражений в конструкции if-else, в том числе на использование дополнительных операторов if и else. В этом случае будет получена вложенная конструкция из нескольких операторов if:

```
if (условие1)
{
  if (условие2)
    выражение1;
  else
  {
    if (условие3)
      выражение2;
    else
      выражение3;
  }
}
else
  выражение4;
```

Смысл этой конструкции из нескольких операторов `if` можно расшифровать так: **если** `условие1` **истинно и** `условие2` **истинно, выполните** `выражение1`. **Если** `условие1` **истинно, а** `условие2` **— нет, тогда проверьте** `условие3` **и, если оно истинно, выполните** `выражение2`. **Если** `условие1` **истинно, а** `условие2` **и** `условие3` **— нет, тогда выполните** `выражение3`. **Наконец, если** `условие1` **ложно, выполните** `выражение4`. **Да, вложенные операторы** `if` **могут кого угодно запутать!**

Пример использования такой сложной конструкции с несколькими операторами `if` показан в листинге 4.7.

Листинг 4.7. Сложные конструкции с вложенными операторами `if`

```
1: // Листинг 4.7. Пример сложной конструкции с
2: // вложенными операторами if
3: #include <iostream.h>
4: int main()
5: {
6:    // Запрашиваем два числа
7:    // Присваиваем числа переменным bigNumber и littleNumber
8:    // Если значение bigNumber больше значения littleNumber,
9:    // проверяем, делится ли большее число на меньшее без остатка
10:   // Если да, проверяем, не равны ли они друг другу
11:
12:   int firstNumber, secondNumber;
13:   cout << "Enter two numbers.\ nFirst: ";
14:   cin >> firstNumber;
15:   cout << "\ nSecond: ";
16:   cin >> secondNumber;
17:   cout << "\ n\ n";
18:
19:   if (firstNumber >= secondNumber)
20:   {
21:    if ( (firstNumber % secondNumber) == 0) // evenly divisible?
22:    {
23:      if (firstNumber == secondNumber)
24:         cout << "They are the same!\ n";
25:      else
26:         cout << "They are evenly divisible!\ n";
27:    }
28:    else
29:      cout << "They are not evenly divisible!\ n";
30:   }
31:   else
32:      cout << "Hey! The second one is larger!\ n";
33:   return 0;
34: }
```

```
Enter two numbers.
First: 10
```

РЕЗУЛЬТАТ

```
Second: 2
They are evenly divisible!
```

АНАЛИЗ

Сначала пользователю предлагается ввести два числа (по очереди), затем эти числа сравниваются. С помощью первого оператора `if` (в строке 19) мы хотим убедиться в том, что первое число больше или равно второму. Если мы убеждаемся в обратном, то выполняется выражение после оператора `else`, представленного в строке 31.

Если первое сравнение возвращает `true`, то выполняется блок инструкций, начинающийся в строке 20, где с помощью второго оператора `if` в строке 21 проверяется предположение, что первое число делится на второе без остатка (т.е. с остатком, равным нулю). Если это предположение подтверждается, то первое число либо кратно второму, либо они вообще равны друг другу. Оператор `if` в строке 23 проверяет версию о равенстве чисел, а затем на экран выводится сообщение, уведомляющее о выявленном соотношении.

Если оператор `if` в строке 21 возвращает значение `false`, то выполняется оператор `else` в строке 28.

Использование фигурных скобок для вложенных операторов if

Фигурные скобки можно не использовать в конструкциях с оператором `if`, если эта конструкция состоит только из одного выполняемого выражения. Это справедливо и в случае вложения нескольких операторов `if`, как показано ниже:

```
if (x > y)          // если x больше y
    if (x < z)      // и если x меньше z,
        x = y;      // тогда присваиваем x значение y
```

Однако при создании сложных вложенных конструкций без использования фигурных скобок бывает трудно разобраться, какое выражение какому оператору `if` принадлежит.

Не забывайте, что пробелы и отступы делают программу понятнее для программиста, но никак не влияют на работу компилятора. Даже если вы покажете с помощью отступа, что данный оператор `else` относится к конструкции этого оператора `if`, компилятор может с вами не согласиться. Данная проблема иллюстрируется в листинге 4.8.

Листинг 4.8. Пример использования фигурных скобок для правильного сопоставления операторов `else` и `if`

```
1:  // Листинг 4.8. Пример использования фигурных скобок
2:  // в конструкциях с вложенными операторами if
3:  #include <iostream.h>
4:  int main()
5:  {
6:    int x;
7:    cout << "Enter a number less than 10 or greater than 100: ";
8:    cin >> x;
9:    cout << "\ n";
```

```
10:
11:   if (x >= 10)
12:    if (x > 100)
13:       cout << "More than 100, Thanks!\ n";
14:   else     // к какому оператору if относится этот оператор
15:    cout << "Less than 10, Thanks!\ n";
16:
17:   return 0;
18:  }
```

РЕЗУЛЬТАТ

```
Enter a number less than 10 or greater than 100: 20
Less than 10, Thanks!
```

АНАЛИЗ Программа запрашивает ввод числа меньше 10 или больше 100 и должна проверить введенное значение на соответствие выдвинутому требованию, а затем вывести сообщение.

Если оператор `if`, расположенный в строке 11, возвращает `true`, то выполняется следующее выражение (строка 12). В нашем примере строка 12 выполняется в случае, если введенное число больше 10. Однако в строке 12 также содержится оператор `if`, который возвращает `true`, если введенное число не больше 100. Если же это число больше 100, выполняется строка 13.

Если введенное число меньше 10, оператор `if` в строке 11 возвратит `false`. В этом случае должно выполниться выражение за оператором `else` (строка 15), которое выводит на экран соответствующее сообщение. Но оказывается, что если ввести число меньше 10, то программа просто завершит свою работу.

Судя по отступу, оператор `else` в строке 14 образует единую конструкцию с оператором `if` в строке 11. Но в действительности оператор `else` связан с оператором `if` в строке 12, поэтому программа будет работать не так, как планировал программист.

С точки зрения компилятора в этой программе на языке C++ нет никаких ошибок. Программа не работает как хотелось из-за логической ошибки. Более того, даже при тестировании этой программы может создаться впечатление, что она работает правильно. Ведь при вводе числа больше 100 программа работает нормально и дефект не проявляется.

В листинге 4.9 показано, как можно исправить эту ошибку с помощью фигурных скобок.

Листинг 4.9. Пример надлежащего использования фигурных скобок в конструкции с оператором `if`

```
1:  // Листинг 4.9. Пример надлежащего использования
2:  // фигурных скобок для вложенных операторов if
3:  #include <iostream.h>
4:  int main()
5:  {
6:    int x;
7:    cout << "Enter a number less than 10 or greater than 100: ";
8:    cin >> x;
9:    cout << "\ n";
10:
```

```
11:   if (x >= 10)
12:   {
13:    if (x > 100)
14:       cout << "More than 100, Thanks!\ n";
15:   }
16:   else          // теперь все ясно!
17:        cout << "Less than 10, Thanks!\ n";
18:   return 0;
19:  }
```

РЕЗУЛЬТАТ

```
Enter a number less than 10 or greater than 100: 20
```

АНАЛИЗ Фигурные скобки, поставленные в строках 12 и 15, превращают все, что стоит между ними, в одно выражение, и теперь оператор `else` в строке 16 явно связан с оператором `if`, стоящим в строке 11, как и было задумано.

Пользователь ввел число `20`, поэтому оператор `if` в строке 11 возвратил значение `true`; однако оператор `if` в строке 13 возвратил `false`, поэтому сообщение не было выведено на экран. Было бы лучше, если бы программист использовал еще один оператор `else` после строки 14, который выводил бы сообщение о том, что введенное число не отвечает требованиям.

ПРИМЕЧАНИЕ

> Программы, приведенные в этой книге, написаны для демонстрации частных вопросов, рассматриваемых в данном разделе. Они намеренно написаны как можно проще; при этом не ставилась цель предусмотреть все возможные ошибки, как это делается в профессиональных программах.

Логические операторы

Довольно часто у нас возникает необходимость проверять не одно условное выражение, а сразу несколько. Например, правда ли, что `x` больше `y`, а также что `y` больше `z`? Наша программа, прежде чем выполнить соответствующее действие, должна установить, что оба эти условия истинны либо одно из них ложно.

Представьте себе высокоорганизованную сигнальную систему, обладающую следующей логикой. Если сработала установленная на двери сигнализация **И** время суток после шести вечера, **И** сегодня **НЕТ** праздника **ИЛИ** сегодня выходной, нужно вызывать полицию. Для проверки всех условий нужно использовать три логических оператора C++. Они перечислены в табл. 4.2.

Таблица 4.2. Логические операторы

Оператор	*Символ*	*Пример*
И	&&	выражение1 && выражение2
ИЛИ	||	выражение1 || выражение2
НЕТ	!	!*выражение*

Логическое И

Логический оператор И вычисляет два выражения, и если оба выражения возвращают `true`, то и оператор И также возвращает `true`. Если правда то, что вы голодны, **И** правда то, что у вас есть деньги, значит, справедливо и то, что вы можете пойти в супермаркет и купить себе что-нибудь на обед. Например, логическое выражение

```
if ( (x == 5) && (y == 5) )
```

возвратит значение `true`, если и обе переменные — `x` и `y` — равны `5`. Это же выражение возвратит `false`, если хотя бы одна из переменных не равна `5`. Обратите внимание, что выражение возвращает `true` только в том случае, если истинны обе его части.

Логический оператор И обозначается *двумя* символами `&&`. Одиночный символ `&` соответствует совсем другому оператору, о котором пойдет речь на занятии 21.

Логическое ИЛИ

Логический оператор ИЛИ также вычисляет два выражения. Если любое из них истинно, то и оператор ИЛИ возвращает `true`. Если у вас есть деньги **ИЛИ** у вас есть кредитная карточка, вы можете оплатить счет. При этом нет необходимости в соблюдении двух условий сразу: иметь и деньги, и кредитную карточку. Вам достаточно выполнения одного из них (хотя и то и другое — еще лучше). Например, выражение

```
if ( (x == 5) || (y == 5) )
```

возвратит значение `true`, если либо значение переменной `x`, либо значение переменной `y`, либо они оба равны `5`.

Обратите внимание: логический оператор ИЛИ обозначается *двумя* символами `||`. Оператор, обозначаемый одиночным символом `|`, — это совсем другой оператор, о котором пойдет речь на занятии 21.

Логическое НЕТ

Логический оператор НЕТ возвращает значение `true`, если тестируемое выражение является ложным (имеет значение `false`). И наоборот, если тестируемое выражение является истинным, то оператор НЕТ возвратит `false`! Например, выражение

```
if ( !(x == 5) )
```

возвратит значение `true` только в том случае, если `x` не равно числу `5`. Это же выражение можно записать и по-другому:

```
if (x != 5)
```

Вычисление по сокращенной схеме

Предположим, компилятору повстречалось следующее логическое выражение:

```
if ( (x == 5) && (y == 5) )
```

В таком случае компилятор сначала оценит первое выражение (`x == 5`) и, если оно возвратит `false` (т.е. `x` не равно числу `5`), не станет вычислять второе выражение (`y == 5`), поскольку для истинности всего выражения с оператором И нужно, чтобы обе его составляющие были истинными.

Аналогично, если компилятору повстречается выражение с оператором ИЛИ

```
if ( (x == 5) || (y == 5) )
```

и первое выражение окажется истинным (`x == 5`), то компилятор также не станет вычислять второе выражение (`y == 5`), поскольку ему достаточно одного истинного результата, чтобы признать истинным все выражение.

Приоритеты операторов отношений

Операторы отношений и логические операторы используются в выражениях языка C++ и возвращают значения `true` или `false`. Подобно всем другим операторам, им присущ некоторый уровень приоритета (см. приложение А), который определяет порядок вычисления операторов отношений. Этот момент нужно обязательно учитывать при определении значения такого выражения, как

```
if ( x > 5 && y > 5 || z > 5)
```

В данном случае о намерениях программиста можно только догадываться. Возможно, он хотел, чтобы это выражение возвращало значение `true`, если `x` и `y` больше 5 или если `z` больше 5. С другой стороны, может быть, программист хотел, чтобы это выражение возвращало `true` только в том случае, если `x` больше 5 и либо `y`, либо `z` больше 5.

Если `x` равен 3, а `y` и `z` оба равны 10, то при использовании первой интерпретации намерений программиста это выражение возвратит значение `true` (`z` больше 5, поэтому игнорируем значения `x` и `y`), но при использовании второй интерпретации вернется значение `false` (оно не может дать значение `true`, поскольку для этого требуется, чтобы значение `x` было больше 5, а после установления этого факта результат вычисления выражения справа от оператора `&&` не важен, ведь для истинности всего выражения обе его части должны быть истинными).

Разобраться в приоритетах операторов в подобных выражениях довольно сложно, поэтому стоит воспользоваться круглыми скобками — ведь с их помощью можно не только изменить последовательность выполнения операторов, обусловленную их приоритетами, но и сделать ясными подобные запутанные выражения:

```
if ( (x > 5) && (y > 5 || z > 5) )
```

При использовании предложенных выше значений это выражение возвращает значение `false`. Поскольку оказалось, что `x` (его значение равно 3) не больше 5, то выражение слева от оператора И возвращает `false`, а следовательно, и все выражение целиком тоже возвратит `false`. Помните, что оператор И возвращает `true` только в том случае, когда обе части выражения возвращают `true`. Например, ваш вкус можно считать хорошим только в том случае, если о надетой на вас вещи можно сказать, что она модная и что вам она идет.

Часто дополнительные круглые скобки стоит использовать только для четкого определения того, что именно вы хотите сгруппировать. Помните, что цель программиста — написать программу, которая прекрасно работает, а также легко читается и понимается.

Подробнее об истине и лжи

В языке C++ нуль эквивалентен значению false, а все другие числовые значения эквивалентны значению true. Поскольку любое выражение всегда имеет значение, многие программисты пользуются преимуществами этой эквивалентности значений в выражениях условия оператора if. Такое выражение, как

```
if (x)      // если x не равен нулю, то условие истинно
   x = 0;
```

можно читать следующим образом: если переменная x имеет ненулевое значение, устанавливаем ее равной нулю. Чтобы сделать смысл этого выражения более очевидным, можно записать его так:

```
if (x != 0)   // если x не нуль
   x = 0;
```

Оба выражения одинаково правомочны, но последнее понятнее. И еще один момент. Чтобы программа не превратилась в сплошное шаманство, лучше все-таки проверять истинность некоторых логических условий, а не равенство выражения нулю.

Следующие два выражения также эквивалентны:

```
if (!x)       // истинно, если x равен нулю
if (x == 0)   // если x равен нулю
```

Однако, второе выражение проще для понимания и гораздо очевиднее, поскольку явно проверяется математическое значение переменной x.

Рекомендуется

Используйте круглые скобки, чтобы более четко указать порядок выполнения операторов или изменить их приоритеты.

Используйте фигурные скобки в конструкциях с вложенными операторами if, чтобы четко определить взаимосвязи между частями конструкции и избежать ошибок.

Не рекомендуется

Не используйте выражение if(x) как эквивалент выражения if(x != 0). Последний вариант предпочтительнее, поскольку четче видна логика проверки условия.

Не используйте выражение if(!x) как эквивалент выражения if(x == 0). Последний вариант предпочтительнее, поскольку четче видна логика проверки условия.

Условный оператор

Условный оператор (?:) — это единственный оператор в языке C++, который работает сразу с тремя операндами.

Синтаксис использования условного оператора следующий:

(*выражение1*) ? (*выражение2*) : (*выражение3*)

Эта строка читается таким образом: если выражение1 возвращает true, то выполняется выражение2, в противном случае выполняется выражение3. Обычно возвращаемое значение присваивается некоторой переменной.

В листинге 4.10 показано использование условного оператора вместо оператора `if`.

Листинг 4.10. Пример использования условного оператора

```
1:  // Листинг 4.10. Пример использования условного оператора
2:  //
3:  #include <iostream.h>
4:  int main()
5:  {
6:    int x, y, z;
7:    cout << "Enter two numbers.\ n";
8:    cout << "First: ";
9:    cin >> x;
10:   cout << "\ nSecond: ";
11:   cin >> y;
12:   cout << "\ n";
13:
14:   if (x > y)
15:    z = x;
16:   else
17:    z = y;
18:
19:   cout << "z: " << z;
20:   cout << "\ n";
21:
22:   z = (x > y) ? x : y;
23:
24:   cout << "z: " << z;
25:   cout << "\ n";
26:   return 0;
27: }
```

РЕЗУЛЬТАТ

```
Enter two numbers.
First: 5

Second: 8

z: 8
z: 8
```

АНАЛИЗ Сначала создается три целочисленные переменные: `x`, `y` и `z`. Значения первых двух вводятся пользователем. Затем в строке 14 выполняется инструкция `if`, которая позволяет узнать, какое из введенных значений больше, причем выявленное большее значение присваивается переменной `z`. Это значение выводится на экран в строке 19.

Ту же самую проверку выполняет в строке 22 условный оператор и присваивает переменной `z` большее значение. Он читается следующим образом: "Если `x` больше `y`, возвращаем значение `x`; в противном случае возвращаем значение `y`". Возвращаемое

значение присваивается переменной z. Это значение выводится на экран в строке 24. Как видите, инструкция, содержащая условный оператор, является более коротким эквивалентом инструкции if...else.

Резюме

Данное занятие охватывает большой объем материала. Вы узнали, что представляют собой операторы и выражения в языке C++, какие разновидности операторов существуют в C++ и как работает оператор if.

Теперь вы знаете, что блок выражений, заключенный внутри пары фигурных скобок, можно использовать вместо одиночного выражения.

Вы также узнали, что каждое выражение возвращает некоторое значение и что это значение можно проверить с помощью инструкции if или условного оператора. Теперь с помощью логических операторов вы можете проверять ряд условий и сравнивать различные значения с помощью операторов отношений. Кроме того, используя оператор присваивания, вы научились присваивать значения переменным.

Вы также познакомились с приоритетами операторов и теперь знаете, как с помощью круглых скобок изменить порядок выполнения операторов, обусловленный их приоритетами, и упростить анализ программного кода.

Вопросы и ответы

Зачем нужны круглые скобки, если можно определить последовательность выполнения операторов по их приоритетам?

Действительно, как программисту, так и компилятору должны быть хорошо известны приоритеты выполнения всех операторов. Но, несмотря на это, стоит использовать круглые скобки, если они облегчают понимание программы, а значит, и дальнейшую работу с ней.

Если операторы отношений всегда возвращают значения `true` **или** `false`**, то почему любое ненулевое значение считается истинным (**`true`**)?**

Операторы отношений всегда возвращают значения `true` или `false`, но в выражениях условий можно использовать любые другие выражения, возвращающие числовые значения. Например:

```
if ( (x = a + b) == 35 )
```

Это вполне подходящее условие для выражения на языке C++. При его выполнении будет вычислено значение даже в том случае, если сумма переменных `a` и `b` не равна числу `35`. Кроме того, обратите внимание, что переменной `x` в любом случае будет присвоено значение переменных `a` и `b`.

Какое воздействие оказывают на программу символы табуляции, пробелы и символы перехода на новую строку?

Символы табуляции, пробелы и символы разрывов строк (все их часто называют символами пробелов) никак не влияют на программу, хотя могут сделать ее более читабельной.

Отрицательные числа считаются истинными или ложными?

Все числа, не равные нулю (как положительные, так и отрицательные), воспринимаются как истинные.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Что такое выражение?
2. Является ли запись `x = 5 + 7` выражением? Каково его значение?
3. Каково значение выражения `201 / 4`?
4. Каково значение выражения `201 % 4`?
5. Если переменные `myAge`, `a` и `b` имеют тип `int`, то каковы будут их значения после выполнения выражения:

```
myAge = 39;
a = myAge++;
b = ++myAge;
```

6. Каково значение выражения `8+2*3`?
7. Какая разница между выражениями `if(x = 3)` и `if(x == 3)`?
8. Будут ли следующие выражения возвращать `true` или `false`?

   а) `0`

   б) `1`

   в) `-1`

   г) `x = 0`

   д) `x == 0 // предположим, что x имеет значение 0`

Упражнения

1. Напишите один оператор `if`, который проверяет две целочисленные переменные и присваивает переменной с бо́льшим значением меньшее значение, используя только один дополнительный оператор `else`.
2. Проанализируйте следующую программу. Представьте, что вы ввели три значения. Какой результат вы ожидаете получить?

```
1: #include <iostream.h>
2: int main()
3: {
4:    int a, b, c;
5:    cout << "Please enter three numbers\ n";
6:    cout << "a:  ";
7:    cin >> a;
```

```
8:   cout << "\ nb:  ";
9:   cin >> b;
10:  cout << "\ nc:  ";
11:  cin >> c;
12:
13:  if (c = (a-b))
14:  { cout << "a:  ";
15:  cout << a;
16:  cout << "minus b:  ";
17:  cout << b;
18:  cout << "equals c:  ";
19:  cout << c << endl;}
20:  else
21:  cout << "a-b does not equal c:  " << endl;
22:  return 0;
23: }
```

3. Введите программу из упражнения 2; скомпилируйте, скомпонуйте и запустите ее на выполнение. Введите числа 20, 10 и 50. Вы получили результат, который и ожидали? Почему нет?
4. Проанализируйте эту программу и спрогнозируйте результат:

```
1:  #include <iostream.h>
2:  int main()
3:  {
4:    int a = 1, b = 1, c;
5:    if (c = (a-b))
6:    cout << "The value of c is: " << c;
7:    return 0;
8:  }
```

5. Введите, скомпилируйте, скомпонуйте и запустите на выполнение программу из упражнения 4. Каков был результат? Почему?

День 5-й

Функции

Несмотря на то что при объектно-ориентированном программировании внимание акцентируется не на функциях, а на объектах, функции тем не менее остаются центральным компонентом любой программы. Итак, сегодня вы узнаете:

- Что такое функция и из чего она состоит
- Как объявлять и определять функции
- Как передавать параметры функциям
- Как возвращать значение функции

Что такое функция

Функция по своей сути — это подпрограмма, которая может манипулировать данными и возвращать некоторое значение. Каждая программа C++ имеет по крайней мере одну функцию `main()`, которая при запуске программы вызывается автоматически. Функция `main()` может вызывать другие функции, те, в свою очередь, могут вызывать следующие и т.д.

Каждая функция обладает собственным именем, и, когда оно встречается в программе, управление переходит к телу данной функции. Этот процесс называется вызовом функции (или обращением к функции). По возвращении из функции выполнение программы возобновляется со строки, следующей после вызова функции. Такая схема выполнения программы показана на рис. 5.1.

Хорошо разработанные функции должны выполнять конкретную и вполне понятую задачу. Сложные задачи следует "разбивать" на несколько более простых, достаточно легко реализуемых с помощью отдельных функций, которые затем могут вызываться по очереди.

Различают два вида функций: определяемые пользователем (нестандартные) и встроенные. Встроенные функции являются составной частью пакета компилятора и предоставляются фирмой-изготовителем. Нестандартные функции создаются самим программистом.

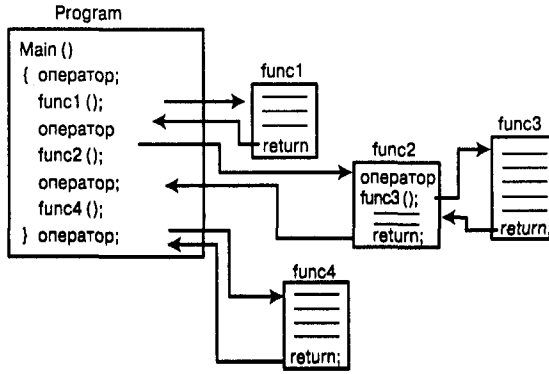


Рис. 5.1. Когда программа вызывает функцию, управление переходит к телу функции, а затем выполнение программы возобновляется со строки, следующей после вызова

Возвращаемые значения, параметры и аргументы

Функции могут *возвращать* значения. После обращения к функции она может выполнить некоторые действия, а затем в качестве результата своей работы послать назад некоторое значение. Оно называется *возвращаемым значением*, причем тип этого значения обязательно должен быть объявлен. Таким образом, запись

```
int myFunction();
```

объявляет, что функция `myFunction` возвращает целочисленное значение.

В функцию можно также и посылать некоторые значения. Описание посылаемых значений называется списком параметров.

```
int myFunction(int someValue, float someFloat);
```

Это объявление означает, что функция `myFunction` не только возвращает целое число, но и принимает два значения в качестве параметров: целочисленное и вещественное.

Параметр описывает *тип* значения, которое будет передано функции при ее вызове. Фактические значения, передаваемые в функцию, называются *аргументами*.

```
int theValueReturned = myFunction(5,6.7);
```

Здесь целая переменная `theValueReturned` инициализируется значением, возвращаемым функцией `myFunction`, и что в качестве аргументов этой функции передаются значения `5` и `6,7`. Тип аргументов должен соответствовать объявленным типам параметров.

Объявление и определение функций

Использование функций в программе требует, чтобы функция сначала была объявлена, а затем определена. Посредством объявления функции компилятору сообщается ее имя, тип возвращаемого значения и параметры. Благодаря определению функ-

ции компилятор узнает, как функция работает. Ни одну функцию нельзя вызвать в программе, если она не была предварительно объявлена. Объявление функции называется *прототипом*.

Объявление функции

Существует три способа объявления функции.

- Запишите прототип функции в файл, а затем используйте выражение с `#include`, чтобы включить его в свою программу.
- Запишите прототип функции в файл, в котором эта функция используется.
- Определите функцию перед тем, как ее вызовет любая другая функция. В этом случае определение функции одновременно и объявляет ее.

Несмотря на то что функцию можно определить непосредственно перед использованием и таким образом избежать необходимости создания прототипа функции, такой стиль программирования не считается хорошим по трем причинам.

Во-первых, требование располагать функции в файле в определенном порядке затрудняет поддержку программы в процессе изменения условий ее использования.

Во-вторых, вполне возможна ситуация, когда функции `A()` необходимо вызвать функцию `B()`, но не исключено также, что при некоторых обстоятельствах и функции `B()` потребуется вызвать функцию `A()`. Однако невозможно определить функцию `A()` до определения функции `B()` и в то же время функцию `B()` до определения функции `A()`, т.е. по крайней мере одна из этих функций обязательно должна быть предварительно объявлена.

В-третьих, прототипы функций — это хорошее и сильное подспорье при отладке. Если согласно прототипу объявлено, что функция принимает определенный набор параметров или что она возвращает значение определенного типа, а затем в программе делается попытка использовать функцию, не соответствующую объявленному прототипу, то компилятор заметит эту ошибку еще на этапе компиляции программы, что позволит избежать неприятных сюрпризов в процессе ее выполнения.

Прототипы функций

Прототипы многих встроенных функций уже записаны в файлы заголовков, добавляемые в программу с помощью `#include`. Для функций, создаваемых пользователями, программист должен сам позаботиться о включении в программу соответствующих прототипов.

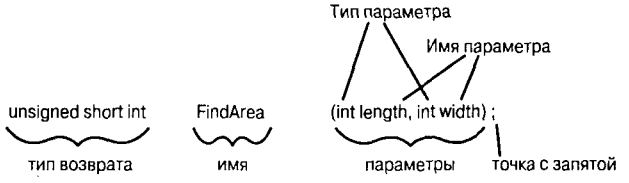


Рис. 5.2. Составные части прототипа функции

Прототип функции представляет собой выражение, оканчивающееся точкой с запятой, и состоит из типа возвращаемого значения функции и сигнатуры. Под *сигнатурой* функции подразумевается ее имя и список формальных параметров.

Список формальных параметров представляет собой список всех параметров и их типов, разделенных запятыми. Составные части прототипа функции показаны на рис. 5.2.

В прототипе и в определении функции тип возвращаемого значения и сигнатура должны соответствовать. Если такого соответствия нет, компилятор покажет сообщение об ошибке. Однако прототип функции не обязан содержать имена параметров, он может ограничиться только указанием их типов. Например, прототип, приведенный ниже, абсолютно правомочен:

```
long Area(int, int);
```

Этот прототип объявляет функцию с именем `Area()`, которая возвращает значение типа `long` и принимает два целочисленных параметра. И хотя такая запись прототипа вполне допустима, это не самый лучший вариант. Добавление имен параметров делает ваш прототип более ясным. Та же самая функция, но уже с именованными параметрами, выглядит гораздо понятнее:

```
long Area(int length, int width);
```

Теперь сразу ясно, для чего предназначена функция и ее параметры.

Обратите внимание на то, что для каждой функции всегда известен тип возвращаемого значения. Если он явно не объявлен, то по умолчанию принимается тип `int`. Однако ваши программы будут понятнее, если для каждой функции, включая `main()`, будет явно объявлен тип возвращаемого значения. В листинге 5.1 приводится программа, которая содержит прототип функции `Area()`.

Листинг 5.1. Объявление, определение и использование функции

```
1:  // Листинг 5.1. Использование прототипов функций
2:  int
3:  #include <iostream.h>
4:  int Area(int length, int width); //прототип функции
5:
6:  int main()
7:  {
8:    int lengthOfYard;
9:    int widthOfYard;
10:   int areaOfYard;
11:
12:   cout << "\ nHow wide is your yard? ";
13:   cin >> widthOfYard;
14:   cout << "\ nHow long is your yard? ";
15:   cin >> lengthOfYard;
16:
17:   areaOfYard= Area(lengthOfYard,widthOfYard);
18:
19:   cout << "\ nYour yard is ";
20:   cout << areaOfYard;
21:   cout << " square feet\ n\ n";
22:             return 0;
23: }
24:
25:   int Area(int yardLength, int yardWidth)
26: {
27:      return yardLength * yardWidth;
28: }
```

РЕЗУЛЬТАТ

```
How wide is your yard? 100
How long is your yard? 200
Your yard is 20000 square feet
```

АНАЛИЗ

Прототип функции `Area()` объявляется в строке 4. Сравните прототип с определением функции, представленным в строке 25. Обратите внимание, что их имена, типы возвращаемых значений и типы параметров полностью совпадают. Если бы они были различны, то компилятор показал бы сообщение об ошибке. Единственное обязательное различие между ними состоит в том, что прототип функции оканчивается точкой с запятой и не имеет тела.

Обратите также внимание на то, что имена параметров в прототипе — `length` и `width` — не совпадают с именами параметров в определении: `yardLength` и `yardWidth`. Как упоминалось выше, имена в прототипе не используются; они просто служат описательной информацией для программиста. Соответствие имен параметров прототипа именам параметров в определении функции считается хорошим стилем программирования; но это не обязательное требование.

Аргументы передаются в функцию в порядке объявления и определения параметров, но без учета какого бы то ни было совпадения имен. Если в функцию `Area()` первым передать аргумент `widthOfYard`, а за ним — аргумент `lengthOfYard`, то эта функция использует значение `widthOfYard` для параметра `yardLength`, а значение `lengthOfYard` — для параметра `yardWidth`. Тело функции всегда заключается в фигурные скобки, даже если оно состоит только из одной строки, как в нашем примере.

Определение функции

Определение функции состоит из заголовка функции и ее тела. Заголовок подобен прототипу функции за исключением того, что параметры в данном случае именованные и в конце заголовка отсутствует точка с запятой.

Рис. 5.3. Заголовок и тело функции

Тело функции представляет собой набор выражений, заключенных в фигурные скобки. Заголовок и тело функции показаны на рис. 5.3.

Функции

Синтаксис прототипа функции:

```
тип_возврата имя_функции ([тип [имя_параметра]...];
{
   выражения;
}
```

Прототип функции сообщает компилятору тип возвращаемого значения, имя функции и список параметров. Наличие параметров не обязательно, но если они все-таки имеются, в прототипе должны быть объявлены их типы. Имена параметров перечислять не обязательно. Строка прототипа всегда оканчивается точкой с запятой (;).

Определение функции должно соответствовать своему прототипу по типу возвращаемого значения и списку параметров. Оно должно содержать имена всех параметров, а тело определения функции должно заключаться в фигурные скобки. Все выражения внутри тела функции должны оканчиваться точкой с запятой, кроме заголовка функции, который оканчивается закрывающей круглой скобкой.

Если функция возвращает значение, она должна содержать выражение с оператором `return`. Это выражение может находиться в любой части определения функции, но обычно оканчивает его.

Для каждой функции задается тип возвращаемого значения. Если он явно не определен, по умолчанию устанавливается тип возврата `int`. Старайтесь всегда указывать тип возвращаемого значения в явном виде. Если функция не возвращает никакого значения, то в качестве типа возвращаемого значения используйте `void`.

Примеры прототипов функций:

```
long FindArea(long length, long width); // возвращает значение типа long, имеет два параметра
void PrintMessage(int messageNumber); // возвращает значение типа void, имеет один параметр
int GetChoice(); // возвращает значение типа int, не имеет параметров
BadFunction(); // возвращает значение типа int, не имеет параметров
```

Примеры определений функций:

```
long FindArea(long l, long w)
{
   return l * w;
}

void PrintMessage(int whichMsg)
{
   if (whichMsg == 0)
      cout << "Hello.\n";
   if (whichMsg == 1)
      cout << "Goodbye.\n";
   if (whichMsg > 1)
      cout << "I'm confused.\n";
}
```

Выполнение функций

При вызове функции ее выполнение начинается с выражения, которое стоит первым после открывающей фигурной скобки ({). В теле функции можно реализовать ветвление, используя условный оператор `if` (и некоторые другие операторы, которые рассматриваются на занятии 7). Функции могут также вызывать другие функции и даже самих себя (о рекурсии речь пойдет ниже в этой главе).

Локальные переменные

В функции можно не только передавать значения переменных, но и объявлять переменные внутри тела функции. Это реализуется с помощью локальных переменных, которые называются так потому, что существуют только внутри самой функции. Когда выполнение программы возвращается из функции к основному коду, локальные переменные удаляются из памяти.

Локальные переменные определяются подобно любым другим переменным. Параметры, переданные функции, тоже считаются локальными переменными, и их можно использовать как определенные внутри тела функции. В листинге 5.2 представлен пример использования параметров функции и переменных, локально определенных внутри функции.

Листинг 5.2. Использование локальных переменных и параметров функции

```
1:  #include <iostream.h>
2:
3:  float Convert(float);
4:  int main()
5:  {
6:    float TempFer;
7:    float TempCel;
8:
9:    cout << "Please enter the temperature in Fahrenheit: ";
10:   cin >> TempFer;
11:   TempCel = Convert(TempFer);
12:   cout << "\ nHere's the temperature in Celsius: ";
13:   cout << TempCel << endl;
14:       return 0;
15: }
16:
17: float Convert(float TempFer)
18: {
19:   float TempCel;
20:   TempCel = ((TempFer - 32) * 5) / 9;
21:   return TempCel;
22: }
```

```
Please enter the temperature in Fahrenheit: 212
Here's the temperature in Celsius: 100
Please enter the temperature in Fahrenheit: 32
Here's the temperature in Celsius: 0
Please enter the temperature in Fahrenheit: 85
Here's the temperature in Celsius: 29.4444
```

АНАЛИЗ В строках 6 и 7 объявляются две переменные типа `float`: одна (`TempFer`) для хранения значения температуры в градусах по Фаренгейту, а другая (`TempCel`) — в градусах по Цельсию. В строке 9 пользователю предлагается ввести температуру по Фаренгейту, и это значение затем передается функции `Convert()`.

После вызова функции `Convert()` программа продолжает выполнение с первого выражения в теле этой функции, представленного строкой 19, где объявляется локальная переменная, также названная `TempCel`. Обратите внимание, что эта локальная переменная — не та же самая переменная `TempCel`, которая объявлена в строке 7. Эта переменная существует только внутри функции `Convert()`. Значение, переданное в качестве параметра `TempFer`, также является лишь переданной из функции `main()` локальной копией одноименной переменной.

В функции `Convert()` можно было бы задать параметр `FerTemp` и локальную переменную `CelTemp`, что не повлияло бы на работу программы. Чтобы убедиться в этом, можете ввести новые имена и перекомпилировать программу.

Локальной переменной `TempCel` присваивается значение, которое получается в результате выполнения следующих действий: вычитания числа `32` из параметра `TempFer`, умножения этой разности на число `5` с последующим делением на число `9`. Результат вычислений затем возвращается в качестве значения возврата функции, и в строке 11 оно присваивается переменной `TempCel` функции `main()`. В строке 13 это значение выводится на экран.

В нашем примере программа запускалась трижды. В первый раз вводится значение `212`, чтобы убедиться в том, что точка кипения воды по Фаренгейту (`212`) сгенерирует правильный ответ в градусах Цельсия (`100`). При втором испытании вводится значение точки замерзания воды. В третий раз — случайное число, выбранное для получения дробного результата.

В качестве примера попробуйте запустить программу снова с другими именами переменных, как показано ниже.

Должен получиться тот же результат.

Каждая переменная характеризуется своей областью видимости, определяющей время жизни и доступность переменной в программе. Переменные, объявленные внутри некоторого блока программы, имеют область видимости, ограниченную этим блоком. К ним можно получить доступ только в пределах этого блока, и после того, как выполнение программы выйдет за пределы, все его локальные переменные автоматически удаляются из памяти. Глобальные же переменные имеют глобальную область видимости и доступны из любой точки программы.

Обычно область видимости переменных очевидна по месту их объявления, но некоторые исключения все же существуют. Подробнее об этом вы узнаете при рассмотрении циклов в занятии 7.

```
1:  #include <iostream.h>
2:
3:  float Convert(float);
4:  int main()
5:  {
6:    float TempFer;
7:    float TempCel;
8:
9:    cout << "Please enter the temperature in Fahrenheit: ";
10:   cin >> TempFer;
11:   TempCel = Convert(TempFer);
12:   cout << "\ nHere's the temperature in Celsius: ";
13:   cout << TempCel << endl;
14:   return 0;
15:  }
16:
17:  float Convert(float Fer)
18:  {
19:   float Cel;
20:   Cel = ((Fer - 32) * 5) / 9;
21:   return Cel;
22:  }
```

Обычно с использованием переменных в функциях не возникает больших проблем, если ответственно подходить к присвоению имен и следить за тем, чтобы в пределах одной функции не использовались одноименные переменные.

Глобальные переменные

Переменные, определенные вне тела какой-либо функции, имеют глобальную область видимости и доступны из любой функции в программе, включая `main()`.

Локальные переменные, имена которых совпадают с именами глобальных переменных, не изменяют значений последних. Однако такая локальная переменная *скрывает* глобальную переменную. Если в функции есть переменная с тем же именем, что и у глобальной, то при использовании внутри функции это имя относится к локальной переменной, а не к глобальной. Это проиллюстрировано в листинге 5.3.

Листинг 5.3. Демонстрация использования глобальных и локальных переменных

```
1:  #include <iostream.h>
2:  void myFunction();       // прототип
3:
4:  int x = 5, y = 7;        // глобальные переменные
5:  int main()
6:  {
7:
8:    cout << "x from main: " << x << "\ n";
9:    cout << "y from main: " << y << "\ n\ n";
10:   myFunction();
11:   cout << "Back from myFunction!\ n\ n";
```

```
12:    cout << "x from main: " << x << "\ n";
13:    cout << "y from main: " << y << "\ n";
14:    return 0;
15: }
16:
17: void myFunction()
18: {
19:    int y = 10;
20:
21:    cout << "x from myFunction: " << x << "\ n";
22:    cout << "y from myFunction: " << y << "\ n\ n";
23: }
```

РЕЗУЛЬТАТ

```
x from main: 5
y from main: 7

x from myFunction: 5
y from myFunction: 10

Back from myFunction!

x from main: 5
y from main: 7
```

АНАЛИЗ Эта простая программа иллюстрирует несколько ключевых моментов, связанных с использованием локальных и глобальных переменных, на которых часто спотыкаются начинающие программисты. В строке 4 объявляются две глобальные переменные — `x` и `y`. Глобальная переменная `x` инициализируется значением 5, глобальная переменная `y` — значением 7.

В строках 8 и 9 в функции `main()` эти значения выводятся на экран. Обратите внимание, что хотя эти переменные не объявляются в функции `main()`, они и так доступны, будучи глобальными.

При вызове в строке 10 функции `myFunction()` выполнение программы продолжается со строки 18, а в строке 19 объявляется локальная переменная `y`, которая инициализируется значением 10. В строке 21 функция `myFunction()` выводит значение переменной `x`. При этом используется глобальная переменная `x`, как и в функции `main()`. Однако в строке 22 при обращении к переменной `y` на экран выводится значение локальной переменной `y`, в то время как глобальная переменная с таким же именем оказывается скрытой.

После завершения выполнения тела функции управление программой возвращается функции `main()`, которая вновь выводит на экран значения тех же глобальных переменных. Обратите внимание, что на глобальную переменную `y` совершенно не повлияло присвоение значения локальной переменной в функции `myFunction()`.

Глобальные переменные: будьте начеку

Следует отметить, что в C++ глобальные переменные почти никогда не используются. Язык C++ вырос из C, где использование глобальных переменных всегда было чревато возникновением ошибок, хотя обойтись без их применения также было не-

возможно. Глобальные переменные необходимы в тех случаях, когда программисту нужно сделать данные доступными для многих функций, а передавать данные из функции в функцию как параметры проблематично.

Опасность использования глобальных переменных исходит из их общедоступности, в результате чего одна функция может изменить значение переменной незаметно для другой функции. В таких ситуациях возможно появление ошибок, которые очень трудно выявить.

На занятии 14 вы познакомитесь с мощной альтернативой использованию глобальных переменных, которая предусмотрена в C++, но недоступна в языке C.

Подробнее о локальных переменных

О переменных, объявленных внутри функции, говорят, что они имеют *локальную область видимости*. Это означает, как упоминалось выше, что они видимы и пригодны для использования только в пределах той функции, в которой определены. Фактически в C++ можно определять переменные в любом месте внутри функции, а не только в ее начале. Областью видимости переменной является блок, в котором она определена. Таким образом, если в теле функции будет блок, выделенный парой фигурных скобок, и в этом блоке объявляется переменная, то она будет доступна только в пределах блока, а не во всей функции, как показано в листинге 5.4.

Листинг 5.4. Видимость локальных переменных

```
1:  // Листинг 5.4. Видимость переменных,
2:  // объявленных внутри блока
3:
4:  #include <iostream.h>
5:
6:  void myFunc();
7:
8:  int main()
9:  {
10:   int x = 5;
11:   cout << "\ nIn main x is: " << x;
12:
13:   myFunc();
14:
15:   cout << "\ nBack in main, x is: " << x;
16:    return 0;
17:  }
18:
19:  void myFunc()
20:  {
21:
22:   int x = 8;
23:   cout << "\ nIn myFunc, local x: " << x << endl;
24:
25:   {
26:     cout << "\ nIn block in myFunc, x is: " << x;
27:
```

```
28:     int x = 9;
29:
30:     cout << "\ nVery local x: " << x;
31:   }
32:
33:   cout << "\ nOut of block, in myFunc, x: " << x << endl;
34: }
```

РЕЗУЛЬТАТ

```
In main x is: 5
In myFunc, local x: 8

In block in myFunc, x is: 8
Very local x: 9
Out of block, in myFunc, x: 8
Back in main, x is: 5
```

АНАЛИЗ Эта программа начинается с инициализации локальной переменной `x` в функции `main()` (строка 10). Выведенное в строке 11 значение переменной `x` позволяет убедиться, что переменная `x` действительно была инициализирована числом `5`.

Затем в программе вызывается функция `MyFunc()`, в теле которой в строке 22 объявляется локальная переменная с тем же именем `x` и инициализируется значением `8`. Это значение выводится на экран в строке 23.

Блок, заключенный в фигурные скобки, начинается в строке 25, и в строке 26 снова выводится значение локальной переменной `x`. Но в строке 28 создается новая переменная с таким же именем `x`, которая является локальной по отношению к данному блоку. Эта переменная тут же инициализируется значением `9`.

Значение последней созданной переменной `x` выводится на экран в строке 30. Локальный блок завершается строкой 31, и переменная, созданная в строке 28, выходит за пределы видимости и удаляется из памяти.

В строке 33 на экран выводится значение той переменной `x`, которая была объявлена в строке 22. На нее никоим образом не повлияло определение новой переменной `x` в строке 28, и ее значение по-прежнему равно `8`.

В строке 34 заканчивается область видимости функции `MyFunc()` и ее локальная переменная `x` становится недоступной. Управление программой возвращается к строке 15, в которой выводится значение локальной переменной `x`, созданной в строке 10. Вы сами можете убедиться в том, что на нее не повлияла ни одна из одноименных переменных, определенных в функции `MyFunc()`.

Нужно ли специально говорить о том, что эта программа была бы гораздо менее путаной, если бы все три переменные имели уникальные имена!

Операторы, используемые в функциях

Фактически на количество или типы операторов, используемых в функциях, никаких ограничений не накладывается. И хотя внутри функции нельзя определить другую функцию, но из одной функции можно *вызывать* сколько угодно других функций; именно этим и занимается функция `main()` почти в каждой программе C++. Более того, функции могут вызывать даже самих себя (эта ситуация рассматривается в разделе, посвященном рекурсии).

Хотя на размер функции в C++ также никакого ограничения не накладывается, лучше, чтобы тело функции не разрасталось до неограниченных масштабов. Многие специалисты советуют сохранять небольшой размер функций, занимающий одну страницу экрана, позволяя тем самым видеть всю функцию целиком. Конечно же, это эмпирическое правило, часто нарушаемое даже очень хорошими программистами, но следует помнить: чем меньше функция, тем она проще для понимания и дальнейшего обслуживания.

Каждая функция должна выполнять одну задачу, которую легко понять. Если вы замечаете, что функция начинает разрастаться, подумайте о том, не пора ли создать новую функцию.

Подробнее об аргументах функций

Аргументы функции могут быть разного типа. Вполне допустимо написать функцию, которая, например, принимает в качестве своих аргументов одно значение типа `int`, два значения типа `long` и один символьный аргумент.

Аргументом функции может быть любое действительное выражение C++, включающее константы, математические и логические выражения и другие функции, которые возвращают некоторое значение.

Использование функций в качестве параметров функций

Несмотря на то что вполне допустимо для одной функции принимать в качестве параметра вторую функцию, которая возвращает некое значение, такой стиль программирования затрудняет чтение программы и ее отладку.

В качестве примера предположим, что у вас есть функции `double()`, `triple()`, `square()` и `cube()`, возвращающие некоторое значение. Вы могли бы записать следующую инструкцию:

```
Answer = (double(triple(square(cube(myValue)))));
```

Эта инструкция принимает переменную `myValue` и передает ее в качестве аргумента функции `cube()`, возвращаемое значение которой (куб числа) передается в качестве аргумента функции `square()`. После этого возвращаемое значение функции `square()` (квадрат числа), в свою очередь, передается в качестве аргумента функции `triple()`. Затем значение возврата функции `triple()` (утроенное число) передается как аргумент функции `double()`. Наконец, значение возврата функции `double()` (удвоенное число) присваивается переменной `Answer`.

Вряд ли можно с полной уверенностью говорить о том, какую задачу решает это выражение (было ли значение утроено до или после вычисления квадрата?); кроме того, в случае неверного результата выявить "виноватую" функцию окажется весьма затруднительно.

В качестве альтернативного варианта можно было бы каждый промежуточный результат вычисления присваивать промежуточной переменной:

```
unsigned long myValue = 2;
unsigned long cubed =  cube(myValue);      // 2 в кубе = 8
unsigned long squared = square(cubed);     // 8 в квадрате = 64
unsigned long tripled = triple(squared);   // 64 * 3 = 192
unsigned long Answer = double(tripled);    // 192 *2 = 384
```

Теперь можно легко проверить каждый промежуточный результат, и при этом очевиден порядок выполнения всех вычислений.

Параметры — это локальные переменные

Аргументы, переданные функции, локальны по отношению к данной функции. Изменения, внесенные в аргументы во время выполнения функции, не влияют на переменные, значения которых передаются в функцию. Этот способ передачи параметров известен как передача *как значения*, т.е. локальная копия каждого аргумента создается в самой функции. Такие локальные копии внешних переменных обрабатываются так же, как и любые другие локальные переменные функции. Эта идея иллюстрируется в листинге 5.5.

Листинг 5.5. Передача параметров как значений

```
1:   // Листинг 5.5. Передача параметров как значений
2:
3:   #include <iostream.h>
4:
5:   void swap(int x, int y);
6:
7:   int main()
8:   {
9:     int x = 5, y = 10;
10:
11:    cout << "Main. Before swap, x: " << x << " y: " << y << "\ n";
12:    swap(x,y);
13:    cout << "Main. After swap, x: " << x << " y: " << y << "\ n";
14:      return 0;
15:   }
16:
17:   void swap (int x, int y)
18:   {
19:    int temp;
20:
21:    cout << "Swap. Before swap, x: " << x << " y: " << y << "\ n";
22:
23:    temp = x;
24:    x = y;
25:    y = temp;
26:
27:    cout << "Swap. After swap, x: " << x << " y: " << y << "\ n";
28:
29:   }
```

РЕЗУЛЬТАТ

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

АНАЛИЗ В программе инициализируются две переменные в функции `main()`, а затем их значения передаются в функцию `swap()`, которая, казалось бы, должна поменять их значения. Однако после повторной проверки этих переменных в функции `main()` оказывается, что они не изменились.

Эти переменные инициализируются в строке 9, а отображение их значений на экране выполняется в строке 11. Затем вызывается функция `swap()`, и эти переменные передаются ей в качестве аргументов.

Выполнение программы переносится в функцию `swap()`, где в строке 21 снова выводятся значения тех же, уже знакомых нам переменных. Как и ожидалось, их значения от передачи в функцию не изменились. В строках 23–25 переменные меняются своими значениями, что подтверждается очередной проверкой в строке 27. Но это положение сохраняется лишь до тех пор, пока программа не вышла из функции `swap()`.

Затем управление программой передается строке 13, принадлежащей функции `main()`, которая показывает, что переменные получили назад свои исходные значения и все изменения, произошедшие в функции, аннулированы!

Напомним, что в данном случае переменные передаются в функцию `swap()` как значения, т.е. в функции `swap()` были созданы копии этих значений, которые являются локальными по отношению к этой функции. Обмен значениями, выполненный в строках 23–25, был реализован на этих локальных переменных, но это никак не повлияло на переменные, оставшиеся в функции `main()`.

На занятиях 8 и 10 вы узнаете альтернативные способы передачи параметров функциям, которые позволят изменять исходные переменные в функции `main()`.

Подробнее о возвращаемых значениях

Функции возвращают либо реальное значение, либо значение типа `void`, которое служит сигналом для компилятора, что никакое значение возвращено не будет.

Чтобы обеспечить возврат значения из функции, напишите ключевое слово `return`, а за ним значение, которое должно быть возвращено. В качестве возврата можно задавать как константные значения, так и целые выражения, например:

```
return 5;
return (x > 5);
return (MyFunction());
```

Все приведенные выше выражения являются правомочными установками возврата функций, если исходить из того, что функция `MyFunction()` сама возвращает некоторое значение. Второе выражение, `return (x > 5)`, будет возвращать `false`, если `x` не больше `5`, или `true`, если `x` больше `5`. Таким образом, если в возврате задается логическое выражение, то возвращаются не значения переменной `x`, а логические значения `false` или `true` (ложь или истина).

После того как в функции встретится ключевое слово `return`, будет выполнено выражение, стоящее за этим ключевым словом, и его результат будет возвращен в основную программу по месту вызова функции. После выполнения оператора `return` программа немедленно переходит к строке, следующей после вызова функции, и любые выражения, стоящие в теле функции после ключевого слова `return`, не выполняются.

Однако функция может содержать несколько операторов `return`. Эта идея иллюстрируется в листинге 5.6.

Листинг 5.6. Использование нескольких операторов `return`

```
1:  // Листинг 5.6. Использование нескольких
2:  // операторов return в теле функции
3:
4:  #include <iostream.h>
5:
6:  int Doubler(int AmountToDouble);
7:
8:  int main()
9:  {
10:
11:    int result = 0;
12:    int input;
13:
14:    cout << "Enter a number between 0 and 10,000 to double: ";
15:    cin >> input;
16:
17:    cout << "\ nBefore doubler is called... ";
18:    cout << "\ ninput: " << input << " doubled: " << result << "\ n";
19:
20:    result = Doubler(input);
21:
22:    cout << "\ nBack from Doubler...\ n";
23:    cout << "\ ninput: " << input << "  doubled: " << result << "\ n";
24:
25:
26:    return 0;
27: }
28:
29: int Doubler(int original)
30: {
31:    if (original <= 10000)
32:      return original * 2;
33:    else
34:      return -1;
35:    cout << "You can't get here!\ n";
36: }
```

Вывод

```
Enter a number between 0 and 10,000 to double: 9000

Before doubler is called...
input: 9000 doubled: 0

Back from doubler...

input: 9000  doubled: 18000

Enter a number between 0 and 10,000 to double: 11000

Before doubler is called...
input: 11000 doubled: 0
Back from doubler...
input: 11000 doubled: -1
```

АНАЛИЗ В строках 14 и 15 программа предлагает пользователю ввести число и сохраняет его в переменной input. В строке 18 отображается только что введенное число вместе со значением локальной переменной result. В строке 20 вызывается функция Doubler() и введенное значение передается ей как параметр. Результат выполнения функции присваивается локальной переменной result, и в строке 23 снова выводятся значения тех же переменных.

В строке 31, относящейся к функции Doubler(), значение переданного параметра сравнивается с числом 10 000. Если окажется, что оно не превышает 10 000, функция возвращает удвоенное значение исходного числа. Если оно больше 10 000, функция возвращает число -1 в качестве сообщения об ошибке.

Выражение в строке 35 никогда не будет достигнуто, потому что при любом значении переданного параметра (большем 10 000 или нет) возврат из функции будет осуществлен либо в строке 32, либо в строке 34, но в любом случае до строки 35. Хороший компилятор сгенерирует предупреждение, что это выражение не может быть выполнено, и хороший программист должен принять соответствующие меры!

Вопросы и ответы

В чем состоит разница между объявлениями `int main()` и `void main()` и какое из них лучше использовать? Ведь оба варианта работают одинаково хорошо, поэтому стоит ли применять первый вариант `int main(){ return 0;}`?

Оба объявления будут работать с большинством компиляторов, но только вариант `int main()` является ANSI-совместимым, следовательно, только объявление `int main()` гарантирует работу программы.

По существу, отличие состоит в следующем. При использовании объявления `int` функция `main()` возвращает значение для операционной системы. После завершения работы вашей программы это значение могут перехватить, например, программы пакетной обработки.

И хотя вы вряд ли будете использовать возвращаемое значение, стандарт ANSI требует его присутствия.

Значения параметров, используемые по умолчанию

Для каждого параметра, объявляемого в прототипе и определении функции, должно быть передано соответствующее значение в вызове функции. Передаваемое значение должно иметь объявленный тип. Следовательно, если некоторая функция объявлена как

```
long myFunction(int);
```

то она действительно должна принимать целочисленное значение. Если тип объявленного параметра не совпадет с типом передаваемого аргумента, компилятор сообщит об ошибке.

Из этого правила существует одно исключение, которое вступает в силу, если в прототипе функции для параметра объявляется стандартное значение. Это значение, которое используется в том случае, если при вызове функции для этого параметра не установлено никакого значения. Несколько изменим предыдущее объявление:

```
long myFunction (int x = 50);
```

Этот прототип нужно понимать следующим образом. Функция myFunction возвращает значение типа long и принимает параметр типа int. Но если при вызове этой функции аргумент предоставлен не будет, используйте вместо него число 50. А поскольку в прототипах функций имена параметров не обязательны, то последний вариант объявления можно переписать по-другому:

```
long myFunction (int = 50);
```

Определение функции не изменяется при объявлении значения параметра, задаваемого по умолчанию. Поэтому заголовок определения этой функции будет выглядеть по-прежнему:

```
long myFunction (int x)
```

Если при вызове этой функции аргумент не устанавливается, то компилятор присвоит переменной x значение 50. Имя параметра, для которого в прототипе устанавливается значение по умолчанию, может не совпадать с именем параметра, указываемого в заголовке функции: значение, заданное по умолчанию, присваивается по позиции, а не по имени.

Установку значений по умолчанию можно назначить любым или всем параметрам функции. Но одно ограничение все же действует: если какой-то параметр не имеет стандартного значения, то ни один из предыдущих по отношению к нему параметров также не может иметь стандартного значения.

Предположим, прототип функции имеет вид

```
long myFunction (int Param1, int Param2, int Param3);
```

тогда параметру Param2 можно назначить стандартное значение только в том случае, если назначено стандартное значение и параметру Param3. Параметру Param1 можно назначить стандартное значение только в том случае, если назначены стандартные значения как параметру Param2, так и параметру Param3. Использование значений, задаваемых параметрам функций по умолчанию, показано в листинге 5.7.

Листинг 5.7. Использование значений, заданных по умолчанию для параметров функций

```
1:  // Листинг 5.7. Использование стандартных
2:  // значений параметров
3:
4:  #include <iostream.h>
5:
6:  int VolumeCube(int length, int width = 25, int height = 1);
7:
8:  int main()
9:  {
10:    int length = 100;
11:    int width = 50;
12:    int height = 2;
13:    int volume;
14:
15:    volume = VolumeCube(length, width, height);
16:    cout << "First volume equals: " << volume << "\ n";
17:
18:    volume = VolumeCube(length, width);
19:    cout << "Second time volume equals: " << volume << "\ n";
20:
```

```
21:    volume = VolumeCube(length);
22:    cout << "Third time volume equals: " << volume << "\ n";
23:    return 0;
24:  }
25:
26:  VolumeCube(int length, int width, int height)
27:  {
28:
29:    return (length * width * height);
30:  }
```

РЕЗУЛЬТАТ

```
First volume equals: 10000
Second time volume equals: 5000
Third time volume equals: 2500
```

АНАЛИЗ

В прототипе функции `VolumeCube()`в строке 6 объявляется, что функция принимает три параметра, причем последние два имеют значения, устанавливаемые по умолчанию.

Эта функция вычисляет объем параллелепипеда на основании переданных размеров. Если значение ширины не передано, то ширина устанавливается равной 25, а высота — 1. Если значение ширины передано, а значение высоты нет, то по умолчанию устанавливается только значение высоты. Но нельзя передать в функцию значение высоты без передачи значения ширины.

В строках 10–12 инициализируются переменные, предназначенные для хранения размеров параллелепипеда по длине, ширине и высоте. Эти значения передаются функции `VolumeCube()` в строке 15. После вычисления объема параллелепипеда результат выводится в строке 16.

В строке 18 функция `VolumeCube()` вызывается снова, но без передачи значения для высоты. В этом случае для вычисления объема параллелепипеда используется значение высоты, заданное по умолчанию, и полученный результат выводится в строке 19.

При третьем вызове функции `VolumeCube()` (строка 21) не передается ни значение ширины, ни значение высоты. Поэтому вместо них используются значения, заданные по умолчанию, и полученный результат выводится в строке 22.

Рекомендуется

Помните, что параметры функции действуют внутри нее, подобно локальным переменным.

Не рекомендуется

Не устанавливайте значение по умолчанию для первого параметра, если для второго параметра используемого по умолчанию значения не предусмотрено.

Не забывайте, что аргументы, переданные в функцию как значения, не могут повлиять на переменные, используемые при вызове функции.

Не забывайте, что изменения, внесенные в глобальную переменную в одной функции, изменяют значение этой переменной для всех функций.

Перегрузка функций

В языке C++ предусмотрена возможность создания нескольких функций с одинаковым именем. Это называется *перегрузкой функций*. Перегруженные функции должны отличаться друг от друга списками параметров: либо типом одного или нескольких параметров, либо различным количеством параметров, либо и тем и другим одновременно. Рассмотрим следующий пример:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

Функция `myFunction()` перегружена с тремя разными списками параметров. Первая и вторая версии отличаются типами параметров, а третья — их количеством.

Типы возвращаемых значений перегруженных функций могут быть одинаковыми или разными. Следует иметь в виду, что при создании двух функций с одинаковым именем и одинаковым списком параметров, но с различными типами возвращаемых значений, будет сгенерирована ошибка компиляции.

Перегрузка функций также называется *полиморфизмом функций*. *Поли* (гр. *poly*) означает *много*, *морфе* (гр. *morphē*) — форма, т.е. полиморфическая функция — это функция, отличающаяся многообразием форм.

Под полиморфизмом функции понимают существование в программе нескольких перегруженных версий функции, имеющих разные назначения. Изменяя количество или тип параметров, можно присвоить двум или нескольким функциям одно и то же имя. При этом никакой путаницы при вызове функций не будет, поскольку нужная функция определяется по совпадению используемых параметров. Это позволяет создать функцию, которая сможет, например, усреднять целочисленные значения, значения типа `double` или значения других типов без необходимости создавать отдельные имена для каждой функции — `AverageInts()`, `AverageDoubles()` и т.д.

Предположим, вы пишете функцию, которая удваивает любое передаваемое ей значение. При этом вы бы хотели иметь возможность передавать ей значения типа `int`, `long`, `float` или `double`. Без перегрузки функций вам бы пришлось создавать четыре разные функции:

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

С помощью перегрузки функций можно использовать следующие объявления:

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

Благодаря использованию перегруженных функций не нужно беспокоиться о вызове в программе нужной функции, отвечающей типу передаваемых переменных. При вызове перегруженной функции компилятор автоматически определит, какой именно вариант функции следует использовать. Перегрузка функции показана в листинге 5.8.

Листинг 5.8. Полиморфизм функций

```
1:  // Листинг 5.8. Пример
2:  // полиморфизма функций
3:
4:  #include <iostream.h>
5:
6:  int Double(int);
7:  long Double(long);
8:  float Double(float);
9:  double Double(double);
10:
11:  int main()
12:  {
13:    int    myInt = 6500;
14:    long   myLong = 65000;
15:    float  myFloat = 6.5F;
16:    double  myDouble = 6.5e20;
17:
18:    int    doubledInt;
19:    long   doubledLong;
20:    float  doubledFloat;
21:    double  doubledDouble;
22:
23:    cout << "myInt: " << myInt << "\ n";
24:    cout << "myLong: " << myLong << "\ n";
25:    cout << "myFloat: " << myFloat << "\ n";
26:    cout << "myDouble: " << myDouble << "\ n";
27:
28:    doubledInt = Double(myInt);
29:    doubledLong = Double(myLong);
30:    doubledFloat = Double(myFloat);
31:    doubledDouble = Double(myDouble);
32:
33:    cout << "doubledInt: " << doubledInt << "\ n";
34:    cout << "doubledLong: " << doubledLong << "\ n";
35:    cout << "doubledFloat: " << doubledFloat << "\ n";
36:    cout << "doubledDouble: " << doubledDouble << "\ n";
37:
38:    return 0;
39:  }
40:
41: int Double(int original)
42: {
43:  cout << "In Double(int)\ n";
44:  return 2 * original;
45: }
46:
47: long Double(long original)
48: {
49:  cout << "In Double(long)\ n";
```

```
50:  return 2 * original;
51: }
52:
53: float Double(float original)
54: {
55:  cout << "In Double(float)\ n";
56:  return 2 * original;
57: }
58:
59: double Double(double original)
60: {
61:  cout << "In Double(double)\ n";
62:  return 2 * original;
63: }
```

РЕЗУЛЬТАТ

```
myInt: 6500
myLong: 65000
myFloat: 6.5
myDouble: 6.5e+20
In Double(int)
In Double(long)
In Double(float)
In Double(double)
DoubledInt: 13000
DoubledLong: 130000
DoubledFloat: 13
DoubledDouble: 1.3e+21
```

АНАЛИЗ Функция `Double()` перегружается для приема параметров четырех типов: `int`, `long`, `float` и `double`. Прототипы функций занимают строки 6–9, а определения — строки 41–63.

В теле основной программы объявляется восемь локальных переменных. В строках 13–16 инициализируются первые четыре переменные, а в строках 28–31 остальным четырем переменным присваиваются результаты передачи значений первых четырех переменных функции `Double()`. Обратите внимание, что по виду вызова эти функции ничем не отличаются друг от друга. Но удивительное дело: вы передаете аргумент — и вызывается нужная функция!

Дело в том, что компилятор определяет тип переданного аргумента, на основании которого выбирает соответствующий вариант функции `Double()`. А результаты работы этой программы подтверждают ожидаемую очередность вызова вариантов этой перегруженной функции.

Дополнительные сведения о функциях

Поскольку функции являются важным элементом программирования, то было бы весьма полезно рассмотреть некоторые специальные темы, интерес к которым возрастает при возникновении нестандартных ситуаций. К числу таких специальных тем, которые способны оказать неоценимую услугу программисту, относятся подставляе-

мые inline-функции и рекурсия функций. Что касается рекурсии, то это замечательное изобретение программистов время от времени позволяет решать такие проблемы, которые практически не решаются никакими другими способами.

Подставляемые inline-функции

Обычно при определении функции компилятор резервирует в памяти только один блок ячеек для сохранения операторов функции. После вызова функции управление программой передается этим операторам, а по возвращении из функции выполнение программы возобновляется со строки, следующей после вызова функции. Если эту функцию вызывать 10 раз, то каждый раз ваша программа будет послушно отрабатывать один и тот же набор команд. Это означает, что существует только одна копия функции, а не 10.

Но каждый переход к области памяти, содержащей операторы функции, замедляет выполнение программы. Оказывается, что, когда функция невелика (т.е. состоит лишь из одной-двух строк), можно получить некоторый выигрыш в эффективности, если вместо переходов от программы к функции и обратно просто дать компилятору команду встроить код функции непосредственно в программу по месту вызова. Когда программисты говорят об эффективности, они обычно подразумевают скорость выполнения программы.

Если функция объявлена с ключевым словом `inline` (т.е. подставляемая), компилятор не создает функцию в памяти компьютера, а копирует ее строки непосредственно в код программы по месту вызова. Это равносильно вписыванию в программе соответствующих блоков вместо вызовов функций.

Обратите внимание, что использование подставляемых функций чревато и некоторыми издержками. Если функция вызывается 10 раз, то во время компиляции в программу будет вставлено 10 копий этой функции. За увеличение скорости выполнения программы нужно будет расплатиться размерами программного кода, в результате чего ожидаемого повышения эффективности программы может и не произойти.

Так какой же напрашивается вывод? Если в программе часто вызывается маленькая функция, состоящая из одной-двух строк, то это первый кандидат в подставляемые функции. Но если функция велика, то лучше воздержаться от ее многократного копирования в программе. Использование подставляемой функции демонстрируется в листинге 5.9.

Листинг 5.9. Использование подставляемых `inline`-функций

```
1:  // Листинг 5.9. Подставляемые inline-функции
2:
3:  #include <iostream.h>
4:
5:  inline int Double(int);
6:
7:  int main()
8:  {
9:   int target;
10:
11:  cout << "Enter a number to work with:
12:  cin >> target;
13:  cout << "\ n";
14:
15:  target = Double(target);
```

```
16:  cout << "Target: " << target << endl;
17:
18:  target = Double(target);
19:  cout << "Target: " << target << endl;
20:
21:
22:  target = Double(target);
23:  cout << "Target: " << target << endl;
24:    return 0;
25: }
26:
27: int Double(int target)
28: {
29:    return 2*target;
30: }
```

РЕЗУЛЬТАТ

```
Enter a number to work with: 20
Target: 40
Target: 80
Target: 160
```

АНАЛИЗ

В строке 5 объявляется подставляемая функция `Double()`, принимающая параметр типа `int` и возвращающая значение типа `int`. Это объявление подобно любому другому прототипу за исключением того, что прямо перед типом возвращаемого значения стоит ключевое слово `inline`.

Результат компиляции этого прототипа равносилен замене в программе строки

```
target = 2 * target;
```

вызовом функции `Double()`:

```
target = Double(target);
```

К моменту выполнения программы копии функции уже расставлены по своим местам и программа готова к выполнению без частых переходов к функции и обратно.

ПРИМЕЧАНИЕ

> Ключевое слово `inline` служит для компилятора рекомендацией пользователя скопировать код функции в программу по месту вызова. Компилятор волен проигнорировать ваши рекомендации и сохранить обычное обращение к функции.

Рекурсия

Функция может вызывать самое себя. Это называется *рекурсией*, которая может быть прямой или косвенной. Когда функция вызывает самое себя, речь идет о прямой рекурсии. Если же функция вызывает другую функцию, которая затем вызывает первую, то в этом случае имеет место косвенная рекурсия.

Некоторые проблемы легче всего решаются именно с помощью рекурсии. Так рекурсия полезна в тех случаях, когда выполняется определенная процедура над данными, а затем эта же процедура выполняется над полученными результатами. Оба типа

рекурсии (прямая и косвенная) выступают в двух амплуа: одни в конечном счете заканчиваются и генерируют возврат, а другие никогда не заканчиваются и генерируют ошибку времени выполнения. Программисты считают, что последний вариант весьма забавен (конечно же, когда он случается с кем-то другим).

Важно отметить, что, когда функция вызывает самое себя, выполняется новая копия этой функции. При этом локальные переменные во второй версии независимы от локальных переменных в первой и не могут непосредственно влиять друг друга, по крайней мере не больше, чем локальные переменные в функции `main()` могут влиять на локальные переменные в любой другой функции, которую она вызывает, как было показано в листинге 5.4.

Чтобы показать пример решение проблемы с помощью рекурсии, рассмотрим ряд Фибоначчи:

1, 1, 2, 3, 5, 8, 13, 21, 34...

Каждое число ряда (после второго) представляет собой сумму двух стоящих впереди чисел. Задача может состоять в том, чтобы, например, определить 12-й член ряда Фибоначчи.

Один из способов решения этой проблемы лежит в тщательном анализе этого ряда. Первые два числа равны 1. Каждое последующее число равно сумме двух предыдущих. Таким образом, семнадцатое число равно сумме шестнадцатого и пятнадцатого. В общем случае *n*-е число равно сумме *(n-2)*-го и *(n-1)*-го при условии, если $n > 2$.

Для рекурсивных функций необходимо задать условие прекращения рекурсии. Обязательно должно произойти нечто, способное заставить программу остановить рекурсию, или же она никогда не закончится. В ряду Фибоначчи условием останова является выражение `n < 3`.

При этом используется следующий алгоритм:

1. Предлагаем пользователю указать, какой член в ряду Фибоначчи следует рассчитать.
2. Вызываем функцию `fib()`, передавая в качестве аргумента порядковый номер члена ряда Фибоначчи, заданный пользователем.
3. В функции `fib()` выполняется анализ аргумента (`n`). Если `n < 3`, функция возвращает значение `1`; в противном случае функция `fib()` вызывает самое себя (рекурсивно), передавая в качестве аргумента значение `n-2`, затем снова вызывает самое себя, передавая в качестве аргумента значение `n-1`, а после этого возвращает сумму.

Если вызвать функцию `fib(1)`, она возвратит `1`. Если вызвать функцию `fib(2)`, она также возвратит `1`. Если вызвать функцию `fib(3)`, она возвратит сумму значений, возвращаемых функциями `fib(2)` и `fib(1)`. Поскольку вызов функции `fib(2)` возвращает значение `1` и вызов функции `fib(1)` возвращает значение `1`, то функция `fib(3)` возвратит значение `2`.

Если вызвать функцию `fib(4)`, она возвратит сумму значений, возвращаемых функциями `fib(3)` и `fib(2)`. Мы уже установили, что функция `fib(3)` возвращает значение 2 (путем вызова функций `fib(2)` и `fib(1)`) и что функция `fib(2)` возвращает значение `1`, поэтому функция `fib(4)` просуммирует эти числа и возвратит значение `3`, которое будет являться четвертым членом ряда Фибоначчи.

Сделаем еще один шаг. Если вызвать функцию `fib(5)`, она вернет сумму значений, возвращаемых функциями `fib(4)` и `fib(3)`. Как мы установили, функция `fib(4)` возвращает значение `3`, а функция `fib(3)` — значение `2`, поэтому возвращаемая сумма будет равна числу `5`.

Описанный метод — не самый эффективный способ решения этой задачи (при вызове функции `fib(20)` функция `fib()` вызывается 13 529 раз!), тем не менее он работает. Однако будьте осторожны. Если задать слишком большой номер члена ряда Фибоначчи, вам может не хватить памяти. При каждом вызове функции `fib()` резервируется некоторая область памяти. При возвращении из функции память освобождается. Но при рекурсивных вызовах резервируются все новые области памяти, а при таком *подходе системная память может исчерпаться довольно быстро. Реализация функции* `fib()` показана в листинге 5.10.

ПРЕДУПРЕЖДЕНИЕ

При запуске программы, представленной в листинге 5.10, задавайте небольшие номера членов ряда Фибоначчи (меньше 15). Поскольку в этой программе используется рекурсия, возможны большие затраты памяти.

Листинг 5.10. Пример использования рекурсии для нахождения члена ряда Фибоначчи

```
1: #include <iostream.h>
2:
3: int fib (int n);
4:
5:  int main()
6:  {
7:
8:   int n, answer;
9:   cout << "Enter number to find: "; 10:   cin >> n;
10:
11:   cout << "\ n\ n";
12:
13:   answer = fib(n);
14:
15:   cout << answer << " is the " << n << "th Fibonacci number\ n"; 17: return 0;
16:  }
17:
18:  int fib (int n)
19:  {
20:   cout << "Processing fib(" << n << ")... "; 23:
21:   if (n < 3 )
22:   {
23:     cout << "Return 1!\ n";
24:     return (1);
25:   }
26:   else
27:   {
28:    cout << "Call fib(" << n-2 << ") and fib(" << n-1 << ").\ n";
29:    return( fib(n-2) + fib(n-1));
30:   }
31:  }
```

РЕЗУЛЬТАТЫ

```
Enter number to find: 6

Processing fib(6)... Call fib(4) and fib(5).
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(5)... Call fib(3) and fib(4).
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
8 is the 6th Fibonacci number
```

> Некоторые компиляторы испытывают затруднения с использованием операторов в выражениях с объектом `cout`. Если вы получите предупреждение в строке 28, заключите операцию вычитания в круглые скобки, чтобы строка 28 приняла следующий вид:
>
> ```
> 28:    cout << "Call fib(" << (n-2) << ") and fib(" << n-1 << ").\ n";
> ```

АНАЛИЗ

В строке 9 программа предлагает ввести номер искомого члена ряда и присваивает его переменной `n`. Затем вызывается функция `fib()` с аргументом `n`. Выполнение программы переходит к функции `fib()`, где в строке 20 этот аргумент выводится на экран.

В строке 21 проверяется, не меньше ли аргумент числа `3`, и, если это так, функция `fib()` возвращает значение `1`. В противном случае выводится сумма значений, возвращаемых при вызове функции `fib()` с аргументами `n-2` и `n-1`. Таким образом, эту программу можно представить как циклический вызов функции `fib()`, повторяющийся до тех пор, пока при очередном вызове этой функции не будет возвращено некоторое значение. Единственными вызовами, которые немедленно возвращают значения, являются вызовы функций `fib(1)` и `fib(2)`. Рекурсивное использование функции `fib()` проиллюстрировано на рис. 5.4 и 5.5.

В примере, изображенном на рисунках, переменная `n` равна значению `6`, поэтому из функции `main()` вызывается функция `fib(6)`. Выполнение программы переходит в тело функции `fib()`, и в строке 30 значение переданного аргумента сравнивается с числом `3`. Поскольку число `6` больше числа `3`, функция `fib(6)` возвращает сумму значений, возвращаемых функциями `fib(4)` и `fib(5)`:

```
38:    return( fib(n-2) + fib(n-1));
```

Это означает, что выполняется обращение к функциям `fib(4)` и `fib(5)` (поскольку переменная `n` равна числу `6`, то `fib(n-2)` — это то же самое, что `fib(4)`, а `fib(n-1)` — то же самое, что `fib(5)`). После этого функция `fib(6)`, которой в текущий момент пере-

дано управление программой, *ожидает*, пока сделанные вызовы не возвратят какое-нибудь значение. Дождавшись возврата значений, эта функция возвратит результат суммирования этих двух значений.

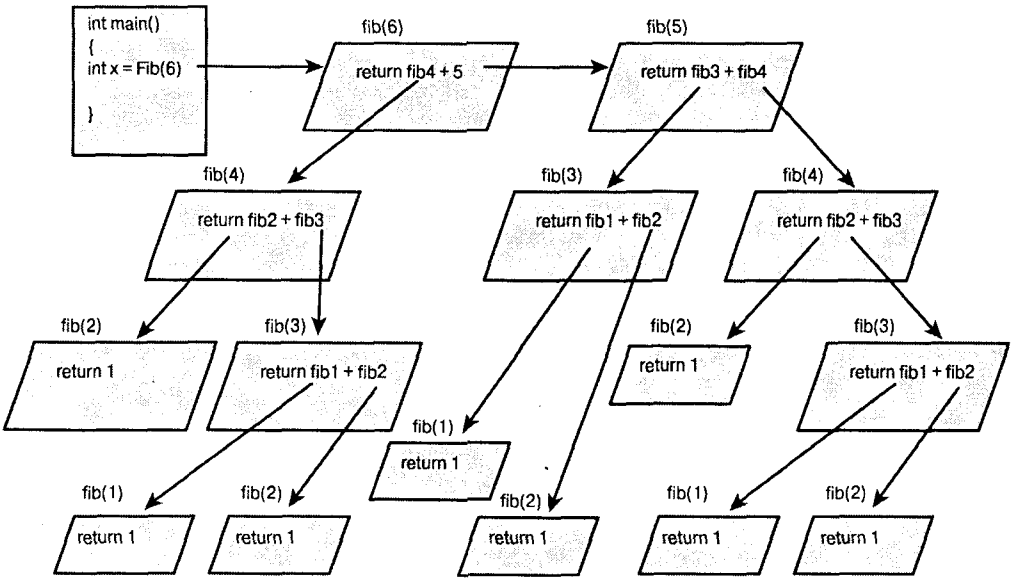


Рис. 5.4. Использование рекурсии

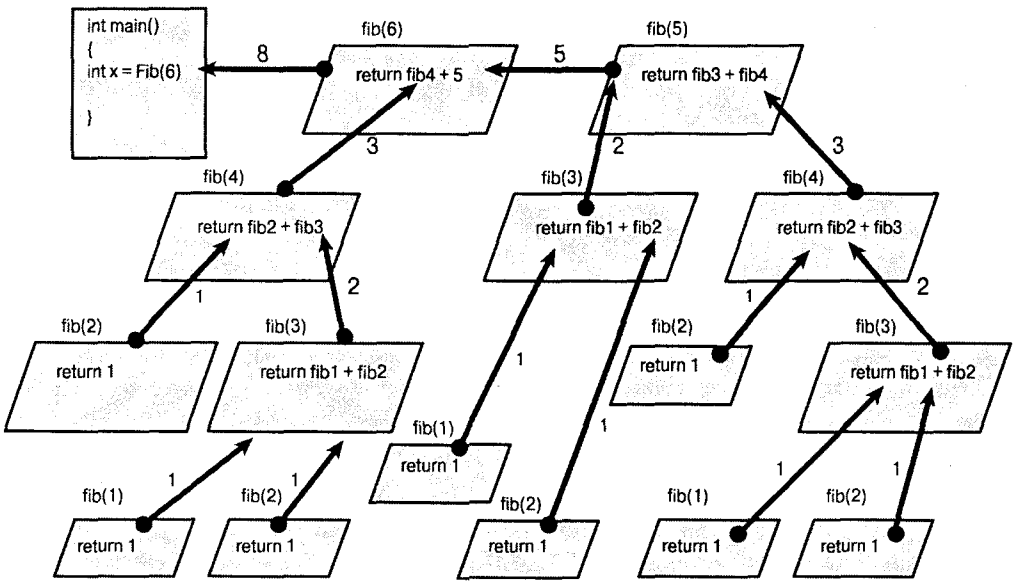


Рис. 5.5. Возвращение из рекурсии

Поскольку при вызове функции `fib(5)` передается аргумент, который не меньше числа 3, функция `fib()` будет вызываться снова, на этот раз с аргументами 4 и 3. А функция `fib(4)` вызовет, в свою очередь, функции `fib(3)` и `fib(2)`.

Результаты и промежуточные этапы работы программы, представленной в листинге 5.10, выводятся на экран. Скомпилируйте, скомпонуйте и выполните эту программу, введя сначала число 1, затем 2, 3, и так доберитесь до числа 6, внимательно наблюдая за отображаемой информацией.

Работа с этой программой предоставляет вам прекрасный шанс проверить возможности своего отладчика. Разместите точку останова в строке 20, а затем заходите в тело каждой вызываемой функции `fib()`, отслеживая значение переменной `n` при каждом рекурсивном вызове функции `fib()`.

В программировании на языке C++ рекурсия не частый гость, но в определенных случаях она является мощным и весьма элегантным инструментом.

Рекурсия относится к одной из самых сложных тем программирования. Данный раздел полезен для понимания основных идей ее реализации, однако не следует слишком расстраиваться, если вам не до конца ясны все детали работы рекурсии.

Работа функций – приподнимем завесу тайны

При вызове функции управление программой передается вызванной функции. Происходит передача параметров, после чего начинается выполнение операторов, составляющих тело функции. По завершении выполнения функции возвращается некоторое значение (если не определено, что функция возвращает тип `void`) и управление передается вызывающей функции.

Как же реализуется эта задача? Откуда программе известно, к какому блоку ей сейчас нужно перейти? Где хранятся переменные при передаче их в качестве аргументов? Что происходит с переменными, которые объявляются в теле функции? Как передается назад возвращаемое значение? Откуда программе известно, с какого места ей нужно продолжить работу?

Многие авторы даже не делают попыток ответить на все эти вопросы, но без понимания принципов работы функций программирование вам покажется сплошным шаманством. Объяснение же потребует краткого освещения вопросов, связанных с памятью компьютера.

Уровни абстракции

Одно из основных препятствий для начинающих программистов — преодоление нескольких уровней абстрагирования от реальности. Компьютеры, конечно, всего лишь электронные машины. Они ничего не знают об окнах и меню, о программах или командах, они даже ничего не знают о единицах и нулях. Все, что происходит в действительности, связано лишь с измерением напряжения в различных точках интегральных микросхем. И даже это является абстракцией. Само электричество представляет собой лишь умозрительную концепция, обобщающую поведение элементарных частиц.

Некоторых программистов пугает любой уровень детализации, опускающийся ниже понятий о значениях, хранящихся в ОЗУ. В конце концов, вам не нужно понимать физику элементарных частиц, чтобы управлять автомобилем, печь пироги или бить по мячу. Точно так же, чтобы программировать, можно обойтись без понимания электроники компьютера.

Однако вы должны понимать, как в компьютере организована память. Без четкого представления о том, где располагаются ваши переменные после их создания и как передаются значения между функциями, программирование останется для вас непостижимой тайной.

Разбиение памяти

Когда вы начинаете работу со своей программой, операционная система (например, DOS или Microsoft Windows) выделяет различные области памяти в ответ на требования компилятора. Как программисту на C++, вам часто придется интересоваться пространством глобальных имен, свободной памятью, регистрами, памятью сегментов программы и стеками.

Глобальные переменные хранятся в пространстве глобальных имен. Подробнее о пространстве глобальных имен и свободной памяти речь пойдет на следующих уроках, а пока сосредоточимся на регистрах, оперативной памяти и стеках.

Регистры представляют собой специальную область памяти, встроенную прямо в центральное процессорное устройство, или центральный процессор (Central Processing Unit — CPU). На их плечи возложена забота о выполнении внутренних вспомогательных функций, описание большей части которых выходит за рамки этой книги. Но мы все-таки остановимся на рассмотрении набора регистров, ответственных за указание на следующую строку программы в любой момент времени. Назовем эти регистры (все вместе) указателями команд. Именно на указатель команды ложится ответственность следить за тем, какая строка программы должна выполняться следующей.

Сама программа находится в памяти компьютера, которая специально отведена для того, чтобы хранить операторы программы в двоичном формате. Каждая строка исходного текста программы транслируется в ряд команд, а каждая из этих команд хранится в памяти по своему адресу. Указатель команды содержит адрес следующей команды, предназначенной для выполнения. Эта идея иллюстрируется на рис. 5.6.

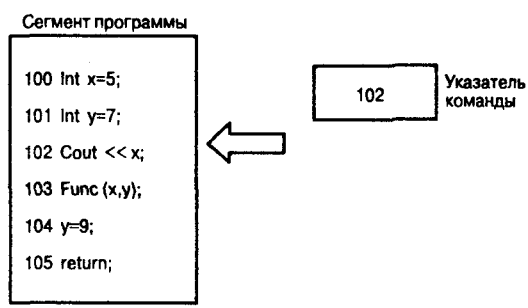


Рис. 5.6. Указатель команды

Стек — это специальная область памяти, выделенная для хранения данных вашей программы, требуемых каждой вызываемой функцией. Она называется стеком потому, что представляет собой очередь типа "*последним пришел — первым вышел*" и напоминает стопку тарелок в руках официанта (рис. 5.7).

Принцип "*последним пришел — первым вышел*" означает, что элемент, добавленный в стек последним, будет вынут из него первым. Большинство же очередей функционирует подобно очереди в театр: первый, кто занял очередь, первым из нее и выйдет (и войдет в театр). Стек скорее напоминает стопку монет, удерживаемых специальным приспособлением. Если расположить в нем 10 монет достоинством в 1 копейку, а затем попытаться вынуть несколько монет, то первой вы достанете ту, что была вставлена последней.

При помещении данных в стек он расширяется, а при возвращении данных из стека — сужается. Невозможно из стопки достать одну тарелку, не вынув предварительно все тарелки, помещенные в стопку перед ней. То же справедливо для данных в стеке памяти.

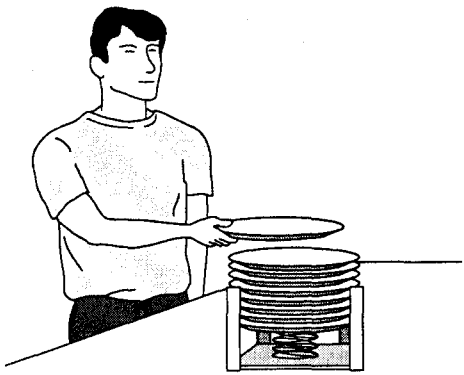


Рис. 5.7. Стек

Аналогия со стопкой тарелок приводится чаще всего. Такое сравнение довольно наглядно, но не вполне верно в смысле техники выполнения. Более точное представление позволит создать ряд прямоугольных полей, выровненных сверху вниз. Вершиной стека будет служить любое поле, на которое указывает в данный момент указатель вершины стека (эту роль выполняет другой регистр).

Все поля имеют последовательные адреса, и один из этих адресов хранится в регистре указателя вершины стека. Все, что находится ниже вершины стека, относится к стеку. Все, что находится выше вершины стека, игнорируется, как показано на рис. 5.8.

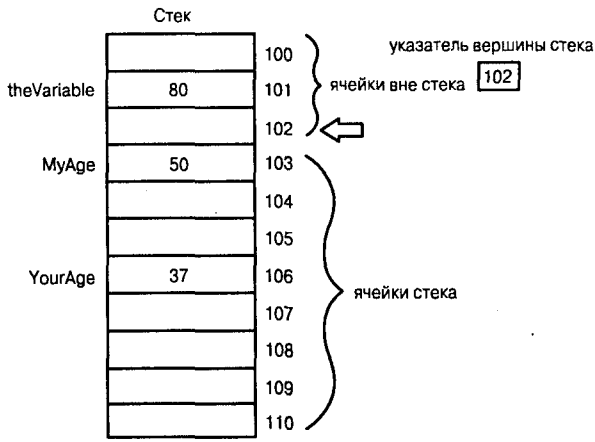


Рис. 5.8. Указатель вершины стека

При помещении некоторого значения в стек оно размещается в поле, расположенном над вершиной стека, после чего указатель вершины изменяется таким образом, чтобы указывать на новое значение. При удалении значения из стека в действитель-

ности происходит лишь изменение адреса указателя вершины стека таким образом, чтобы он указывал на подлежащий удалению элемент стека. Принцип действия схематически показан на рис. 5.9.

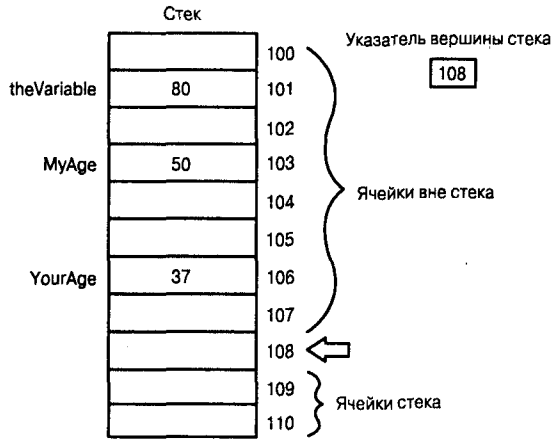


Рис. 5.9. Перемещение указателя вершины стека

Стек и функции

Ниже перечислены действия, происходящие с программой, выполняемой под управлением DOS, при переходе к телу функции.

1. Увеличивается адрес, содержащийся в указателе команды, чтобы указывать на инструкцию, следующую после вызова функции. Затем этот адрес помещается в стек и будет служить адресом возврата по завершении выполнения функции.
2. В стеке резервируется место для возвращаемого функцией значения объявленного вами типа. Если в системе с двухбайтовыми целыми для возвращаемого значения объявлен тип `int`, то к стеку добавляются еще два байта, но в эти байты ничего пока не помещается.
3. В указатель команды загружается адрес вызванной функции, который хранится в отдельной области памяти, отведенной специально для этих целей. Поэтому следующей выполняемой командой будет первый оператор вызванной функции.
4. Текущая вершина стека помечается и содержится в специальном указателе, именуемом *указателем стека*. Все, что добавляется в стек с этого момента и до тех пор, пока функция не завершится, рассматривается как локальные данные этой функции.
5. В стек помещаются все аргументы, передаваемые функции.
6. Выполняется команда, адрес которой находится в данный момент в указателе команды, т.е. первая строка кода функции.
7. По мере определения в стеке размещаются локальные переменные и функции.

Когда функция завершается, возвращаемое значение помещается в область стека, зарезервированную на этапе 2. Затем из стека удаляется все содержимое вплоть до указателя стека, благодаря чему стек очищается от локальных переменных и аргументов, переданных функции.

Затем из стека извлекаются значение возврата функции, которое присваивается переменной, вызвавшей функцию, и адрес команды, сохраненный в стеке на этапе 1, который присваивается указателю команд. Таким образом, программа продолжает свою работу со следующей строки после обращения к функции, владея уже значением, возвращенным из функции.

Некоторые детали этого процесса изменяются при переходе от компилятора к компилятору или от компьютера к компьютеру, но основная идея остается прежней независимо от операционной среды. В общем случае при вызове функции адрес возврата и ее параметры всегда помещаются в стек. На протяжении жизненного цикла функции в стек добавляются локальные переменные. По возвращении из функции все они удаляются из стека.

На следующих занятиях рассматриваются некоторые особенности других областей памяти, которые используются для хранения глобальных данных программы.

Резюме

На этом занятии вы познакомились с функциями. Функция в действительности представляет собой подпрограмму, которой можно передавать параметры и из которой можно возвращать значение. Каждый запуск программы C++ начинается с выполнения функции `main()`, которая, в свою очередь, может вызывать другие функции.

Функция объявляется с помощью прототипа функции, который описывает возвращаемое значение, имя функции и типы ее параметров. При желании функцию можно объявить подставляемой (с помощью ключевого слова `inline`). В прототипе функции можно также объявить значения, используемые по умолчанию для одного или нескольких параметров функции.

Определение функции должно соответствовать прототипу функции по типу возвращаемого значения, имени и списку параметров. Имена функций могут быть перегружены путем изменения количества или типа параметров. Компилятор находит нужную функцию на основе списка параметров.

Локальные переменные функции и аргументы, передаваемые функции, локальны по отношению к блоку, в котором они объявлены. Параметры, передаваемые как значения, представляют собой копии реальных переменных и не могут влиять на значения этих переменных в вызывающей функции.

Вопросы и ответы

Почему бы не сделать все переменные глобальными?

Когда-то именно так и поступали. Но по мере усложнения программ стало очень трудно находить в них ошибки, поскольку значения глобальных переменных могли быть изменены любой из функций, поэтому сложно было определить, какой именно блок программы виновен в ошибке. Многолетний опыт убедил программистов, что данные должны храниться локально (насколько это возможно) и доступ к изменению данных должен быть определен как можно более узким кругом.

Когда следует использовать в прототипе функции ключевое слово `inline`?

Если функция невелика (занимает не более одной-двух строк) и встраивание ее в код программы по всем местам вызова не увеличит существенно размер этой программы, то, возможно, имеет смысл объявить ее как `inline`.

Почему изменения, вносимые в теле функции в переменные, переданные как аргументы, не отражаются на значениях этих переменных в основном коде программы?

Аргументы обычно передаются в функцию как значения, т.е. аргумент в функции является на самом деле копией оригинального значения. Данная концепция подробно разъяснялась на этом занятии.

Как поступить, если необходимо, чтобы изменения, внесенные в функции, сохранились после возвращения из функции?

Эта проблема рассматривается на занятии 8. Использование указателей не только решает эту проблему, но также предоставляет способ обойти ограничение на возврат только одного значения из функции.

Что произойдет, если объявить следующие две функции:

```
int Area (int width, int length = 1); int Area (int size);
```

Будут ли они перегруженными? Условие уникальности списков параметров соблюдено, но в первом варианте для параметра определено значение, используемое по умолчанию.

Эти объявления будут скомпилированы, но, если вызвать функцию `Area ()` с одним параметром, будет сгенерирована ошибка компиляции, обусловленная неопределенностью между функциями `Area(int, int)` и `Area(int)`.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из вопросов, предложенных ниже.

Контрольные вопросы

1. В чем разница между объявлением прототипа функции и определением функции?
2. Должны ли имена параметров, указанные в прототипе, определении и вызове функции соответствовать друг другу?
3. Если функция не возвращает значение, как следует объявить такую функцию?
4. Если не объявить тип возврата, то какой тип будет принят по умолчанию для возвращаемого значения?
5. Что такое локальная переменная?
6. Что такое область видимости?
7. Что такое рекурсия?
8. Когда следует использовать глобальные переменные?
9. Что такое перегрузка функции?
10. Что такое полиморфизм?

Упражнения

1. Запишите прототип для функции с именем `Perimeter`, которая возвращает значение типа `unsigned long int` и принимает два параметра типа `unsigned short int`.
2. Запишите определение функции `Perimeter` согласно объявлению в упражнении 1. Два принимаемых ею параметра представляют длину и ширину прямоугольника, а функция возвращает его периметр (удвоенная длина плюс удвоенная ширина).
3. **Жучки**: что неправильно в этой функции?

```
#include <iostream.h>
void myFunc(unsigned short int x);
int main()
{
unsigned short int x, y;
y = myFunc(int);
cout << "x: " << x << " y: " << y << "\ n";
}

void myFunc(unsigned short int x)
{
return (4*x);
}
```

4. **Жучки**: что неправильно в этой функции?

```
#include <iostream.h>
int myFunc(unsigned short int x);
int main()
{
unsigned short int x, y;
y = myFunc(x);
cout << "x: " << x << " y: " << y << "\ n";
}

int myFunc(unsigned short int x);
{
return (4*x);
}
```

5. Напишите функцию, которая принимает два параметра типа `unsigned short int` и возвращает результат деления первого параметра на второй. Функция не должна выполнять операцию деления, если второе число равно нулю, но в этом случае она должна возвратить значение `-1`.
6. Напишите программу, которая запрашивает у пользователя два числа и вызывает функцию, записанную при выполнении упражнения 5. Выведите результат или сообщение об ошибке, если функция возвратит значение, равное `-1`.
7. Напишите программу, которая запрашивает число и показатель степени. Напишите рекурсивную функцию, которая возводит число в степень путем многократного умножения числа на самое себя, т.е. если число равно `2`, а показатель степени равен `4`, то эта функция должна возвратить число `16`.

Базовые классы

Базовые классы расширяют встроенные средства языка C++, что способствует решению сложных проблем, которые ставит перед программистами реальная жизнь. Сегодня вы узнаете:

- Что представляют собой классы и объекты
- Как определить новый класс и создать объекты этого класса
- Что представляют собой функции-члены и переменные-члены
- Что такое конструктор и как его использовать

Создание новых типов

Вы уже познакомились с типами переменных, включая беззнаковые целые и символы. Тип переменной несет в себе немало информации. Например, если объявить переменные `Height` и `Width` как беззнаковые короткие целые (`unsigned short int`), то каждая из них сможет хранить целое число в диапазоне 0–65 535, занимая при этом только два байта. Если же вы попытаетесь присвоить такой переменной значение, отличное от беззнакового целого числа, то получите сообщение об ошибке. Это значит, что с помощью такой переменной вы не сможете хранить свое имя, так что даже и не пытайтесь сделать это.

Лишь объявив переменные `Height` и `Width` беззнаковыми короткими целыми, вы получаете возможность сложить их или присвоить одной из них значение другой.

Итак, тип переменной определяет:

- ее размер в памяти;
- тип данных, которые она может хранить;
- операции, которые могут выполняться с ее участием.

Тип данных является категорией. К нему можно отнести автомобиль, дом, человека, фрукты, геометрическую фигуру и т.п. В языке C++ программист может создать любой нужный ему тип, и каждый из этих типов может совмещать свойства и функциональные возможности встроенных базовых типов.

Зачем создавать новый тип

Программы обычно пишут для решения таких реальных проблем, как отслеживание информации о служащих или имитация работы отопительной системы. И хотя решать сложные проблемы можно с помощью программ, написанных только с использованием одних целочисленных значений и символов, решения выглядели бы значительно проще, если бы можно было создавать уникальные типы для различных объектов. Другими словами, имитацию работы отопительной системы было бы гораздо легче реализовать, если бы можно было создавать переменные, представляющие помещения, тепловые датчики, термостаты и бойлеры. И чем ближе эти переменные соответствуют реальности, тем легче написать такую программу.

Классы и члены классов

Новый тип создается путем объявления класса. *Класс* — это просто коллекция переменных (причем часто различных типов), скомбинированная с набором связанных функций.

Автомобиль можно представлять себе по-разному, например как коллекцию, состоящую из колес, дверей, сидений, окон и т.д. Или же, думая об автомобиле, можно представить себе его способность двигаться, увеличивать скорость, тормозить, останавливаться, парковаться и т.д. Класс позволяет инкапсулировать различные запчасти автомобиля и его разнообразные функции в одну коллекцию, которая называется объектом.

Инкапсуляция всего, что мы знаем об автомобиле, в один класс имеет для программиста ряд преимуществ. Ведь все сведения собраны вместе в одном объекте, на который легко ссылаться, копировать и манипулировать его данными. Клиенты вашего класса, т.е. части программы, работающие с этим классом, могут использовать ваш объект, не беспокоясь о том, что находится в нем или как именно он работает.

Класс может состоять из любой комбинации типов переменных, а также типов других классов. Переменные в классе называют переменными-членами или данными-членами. Класс `Car` может иметь переменные-члены, представляющие сидения, радиоприемник, шины т.д.

Переменные-члены, известные также как данные-члены, принадлежат только своему классу. Переменные-члены — это такие же составные части класса, как колеса и мотор — составные части автомобиля.

Функции в классе обычно выполняют действия над переменными-членами. Они называются *функциями-членами* или *методами* класса. В число методов класса `Car` могут входить `Start()` и `Break()`. Класс `Cat` может иметь такие данные-члены, которые представляют возраст и вес животного, а функциональная часть этого класса может быть представлена методами `Sleep()`, `Meow()` и `ChaseMice()`.

Функции-члены принадлежат своему классу, как и переменные-члены. Они оперируют переменными-членами и определяют функциональные возможности класса.

Объявление класса

Для объявления класса используйте ключевое слово `class`, за которым следует открывающая фигурная скобка, а за ней — список данных-членов и методов класса. Объявление завершается закрывающей фигурной скобкой и точкой с запятой. Вот, например, как выглядит объявление класса `Cat`:

```
class Cat
{
    unsigned int itsAge;
    unsigned int itsWeight;
    void Meow();
};
```

При объявлении класса Cat память не резервируется. Это объявление просто сообщает компилятору о существовании класса Cat, о том, какие данные он содержит (itsAge и itsWeight), а также о том, что он умеет делать (метод Meow()). Кроме того, данное объявление сообщает компилятору о размере класса Cat, т.е. сколько места должен зарезервировать компилятор для каждого объекта класса Cat. Поскольку в данном примере для целого значения требуется четыре байта, то размер объекта Cat составит восемь байтов (четыре байта для переменной itsAge и четыре — для itsWeight). Метод Meow() не требует выделения памяти в объекте.

Несколько слов об используемых именах

На программиста возложена ответственность за присвоение имен переменным-членам, функциям-членам и классам. Как упоминалось на занятии 3, всегда следует давать понятные и осмысленные имена. Например, Cat (Кот), Rectangle (Прямоугольник) и Employee (Служащий) — вполне подходящие имена для классов, а Meow() (Мяу), ChaseMice() (ДогониМышку) и StopEngine() (ОстановкаДвигателя) — прекрасные имена для методов, поскольку из их названий понятно, что они делают. Многие программисты сопровождают имена своих переменных-членов префиксами its (например, itsAge, itsWeight, itsSpeed). Это помогает отличить переменные-члены от переменных, не являющихся членами класса.

В языке C++ имеет значение регистр букв, и все имена классов должны следовать одному образцу. Исходя из этого, вам никогда не придется вспоминать, как именно пишется название вашего класса: Rectangle, rectangle или RECTANGLE. Некоторые программисты любят добавлять к имени каждого класса однобуквенный префикс c (от слова class) например, cCat или cPerson, в то время как другие используют для имени только прописные или же только строчные буквы. Я предпочитаю начинать имена классов с прописной буквы, например Cat или Person.

Также многие программисты начинают имена функций с прописных букв, а для имен всех остальных переменных используют только строчные буквы. Слова, являющиеся составными частями имен, разделяют обычно символами подчеркивания (например, Chase_Mice) или просто начинают каждое слово с прописной буквы (например, ChaseMice или DrawCircle).

Важно придерживаться одного стиля на протяжении всей программы. По мере приобретения опыта программирования ваш собственный стиль написания программ включит в себя соглашения не только по присвоению имен, но также и по отступам, выравниванию фигурных скобок и оформлению комментариев.

Обычно солидные компании по разработке программных продуктов имеют специальные отделы, которые занимаются вопросами стандартизации, охватывающими и стилевые особенности программ. Это гарантирует, что все разработчики смогут легко читать программы, созданные их коллегами.

Определение объекта

Объект нового типа определяется таким же способом, как и любая целочисленная переменная:

```
unsigned int GrossWeight; // определяем беззнаковое целое
Cat Frisky;               // определяем объект Cat
```

В этих программных строках определяется переменная с именем `GrossWeight`, которая имеет тип `unsigned int`, а также определяется объект `Frisky`, который является объектом класса (или имеет тип) `Cat`.

Классы в сравнении с объектами

Вам никогда не придет в голову поиграть с кошкой как с абстрактным понятием, скорее вы приласкаете свою настоящую мурку. Не нужно много говорить о том, какая разницу между кошкой вообще и конкретным котом, от которого шерсть по всей комнате и царапины на ножках стульев. Точно такая же разница между классом `Cat`, представляющим собой некую абстракцию, и отдельным объектом класса `Cat`. Следовательно, `Frisky` — это объект типа `Cat` в том самом смысле, в котором `GrossWeight` — переменная типа `unsigned int`.

Итак, мы пришли к тому, что объект — это отдельный экземпляр некоторого класса.

Получение доступа к членам класса

После определения реального объекта класса `Cat`, например `Frisky`, у нас может возникнуть необходимость в получении доступа к членам этого объекта. Для этого используется оператор прямого доступа (`.`). Следовательно, чтобы присвоить число 50 переменной-члену `Weight` объекта `Frisky`, можно записать

```
Frisky.Weight = 50;
```

Аналогично, для вызова метода `Meow()` достаточно использовать следующую запись:

```
Frisky.Meow();
```

Когда нужно использовать некоторый метод класса, выполняется вызов этого метода. В данном примере вызывается метод `Meow()` объекта `Frisky`.

Значения присваиваются объектам, а не классам

В языке C++ нельзя присваивать значения типам данных, они присваиваются только переменным. Например, такая запись неверна:

```
int = 5;        // неверно
```

Компилятор расценит это как ошибку, поскольку нельзя присваивать число типу `int`. Вместо этого нужно определить целочисленную переменную и присвоить число `5` этой переменной. Например:

```
int x;          // определяем x как переменную типа int
x = 5;          // присвоение переменной x значения 5
```

Таким образом, число 5 присваивается переменной x, которая имеет тип int. Из тех же соображений недопустима следующая запись:

```
Cat.itsAge=5;          // неверно
???
```

Если Cat — это класс, а не объект, то компилятор отметит это выражение как ошибочное, поскольку нельзя присвоить число 5 переменной itsAge класса (т.е. типа) Cat. Вместо этого нужно определить объект класса Cat и присвоить число 5 соответствующей переменной-члену этого объекта. Например:

```
Cat Frisky;            // это определение аналогично int x;
Frisky.itsAge = 5;     // это присвоение аналогично x = 5;
```

Что объявишь, то и будешь иметь

Представьте себе, что вы гуляете со своим трехлетним ребенком, показываете ему кошку и говорите: "Это Фриски, чудесная кошка, ну-ка Фриски, залай". Даже маленький ребенок рассмеется и скажет: "Нет, кошки не умеют лаять".

Если вы запишете:

```
Cat Frisky;            // создаем кошку (объект) по имени Frisky
Frisky.Bark();         // велим Frisky залаять
```

то компилятор тоже сообщит вам, что даже виртуальные кошки лаять не умеют, поскольку для них не объявлен такой метод. В классе Cat есть метод Meow() (мяукать). Если же вы не определите в классе Cat метод Meow(), то компилятор не позволит вашей кошке даже мяукать.

Рекомендуется

Используйте ключевое слово class для объявления класса.

Используйте оператор прямого доступа (.) для получения доступа к переменным-членам и методам класса.

Не рекомендуется

Не путайте объявление с определением. Объявление заявляет о существовании класса, а определение резервирует память для объекта.

Не путайте класс с объектом.

Не присваивайте значения классу. Присваивайте значения переменным-членам объекта.

Ограничение доступа к членам класса

В объявлении класса используются и другие ключевые слова. Двумя самыми важными из них являются public (открытый) и private (закрытый), определяющие доступ к членам класса.

Все члены класса — данные и методы — являются закрытыми по умолчанию. К закрытым членам можно получить доступ только с помощью методов самого класса. Открытые члены доступны для всех других функций программы. Определение доступа

к членам класса имеет очень важное значение, и именно при решении этой задачи начинающие программисты часто сталкиваются с трудностями. Чтобы прояснить ситуацию, рассмотрим пример, который уже приводился выше в этой главе:

```
class Cat
{
     unsigned int itsAge;
     unsigned int itsWeight;
     void Meow();
};
```

В этом объявлении переменные `itsAge` и `itsWeight`, а также метод `Meow()` являются закрытыми, поскольку все члены класса закрытые по умолчанию. Если требуется изменить доступ к членам класса, то это следует сделать явно.

Если в программе будет описан класс `Cat`, как показано выше, то обращение к переменной-члену `itsAge` из функции `main()` вызовет ошибку компиляции:

```
Cat Boots;
Boots.itsAge = 5; // Ошибка! Нельзя обращаться к закрытым данным
```

И в самом деле, сначала компилятору указывается, что члены `itsAge`, `itsWeight` и `Meow()` можно использовать только внутри класса `Cat`, а затем делается попытка использовать во внешней функции переменную-член `itsAge`, безраздельно принадлежащую объекту `Boots` класса `Cat`. Хотя объект `Boots` реально существует в программе, это не означает, что можно получать доступ к членам данного объекта, закрытым для постороннего глаза.

Именно эти моменты с определением доступа к членам класса служат источником бесконечных недоразумений у начинающих программистов. Я прямо-таки слышу ваш удивленный вопрос: "Если в программе объявлен реальный объект `Boots` класса `Cat`, почему же нельзя присвоить значение переменной-члену этого объекта, даже обратившись к ней с помощью оператора прямого доступа?"

Дело в том, что в объявлении класса `Cat` ничего не говорится о ваших правах обращаться к членам этого класса, а это значит, что вы таких прав не имеете. Только собственные методы объекта `Boots` всегда имеют доступ ко всем данным класса, как открытым, так и закрытым. Даже несмотря на то что вы сами создали класс `Cat`, это не дает вам права возвращать или изменять в программе его данные, которые являются закрытыми.

Однако из любого положения есть выход. Чтобы получить доступ к переменным-членам класса `Cat`, откройте их следующим способом:

```
class Cat
{
public:
     unsigned int itsAge;
     unsigned int itsWeight;
     void Meow();
};
```

Теперь благодаря ключевому слову `public` все члены класса (`itsAge`, `itsWeight` и `Meow()`) стали открытыми.

В листинге 6.1 показано объявление класса `Cat` с открытыми переменными-членами.

Листинг 6.1. Доступ к открытым членам простого класса

```
1: // Пример объявление класса с
2: // открытыми членами
3:
4: #include <iostream.h> // для использования cout
5:
6: class Cat // объявляем класс
7: {
8: public: // следующие члены являются открытыми
9:   int itsAge;
10:  int itsWeight;
11: };
12:
13:
14: int main()
15: {
16:  Cat Frisky;
17:  Frisky.itsAge =5; // присваиваем значение переменной-члену
18:  cout << "Frisky is a cat who is ";
19:  cout << Frisky.itsAge << " years old.\n";
20:  return 0;
21: }
```

РЕЗУЛЬТАТ

```
Frisky is a cat who is 5 years old.
```

АНАЛИЗ В строке 6 содержится ключевое слово `class`. Оно сообщает компилятору о том, что следующий после него блок является объявлением класса. Имя нового класса стоит сразу после ключевого слова `class`. В данном случае у нас объявляется класс `Cat`.

Тело объявления класса начинается с открывающей фигурной скобки в строке 7 и заканчивается закрывающей фигурной скобкой и точкой с запятой в строке 11. Строка 8 содержит ключевое слово `public`, которое означает, что до тех пор, пока не встретится ключевое слово `private` или конец объявления класса, все последующие члены объявляются открытыми.

В строках 9 и 10 объявляются переменные-члены `itsAge` и `itsWeight`.

В строке 14 начинается функция `main()` программы. `Frisky` определяется в строке 16 как экземпляр класса `Cat`, т.е. как объект класса `Cat`. В строке 17 возраст объекта `Frisky` (значение переменной `itsAge`) устанавливается равным `5`. А в строках 18 и 19 переменная-член `itsAge` используется для вывода данных на экран.

ПРИМЕЧАНИЕ

Попробуйте заблокировать символом комментария строку 8 и перекомпилировать программу. Компилятор покажет сообщение об ошибке в строке 17, поскольку к переменной `itsAge` больше нет открытого доступа, ведь по умолчанию все члены класса объявляются как закрытые.

Оставьте данные класса закрытыми

Согласно общей стратегии использования классов переменные-члены класса следует оставлять закрытыми. Благодаря этому достигается инкапсуляция данных внутри класса. Доступ следует открывать только к функциям-членам класса, обеспечивающим доступ к его закрытым данным (эти функции еще называют *методами доступа*). Эти методы можно вызывать из любого места в программе для возвращения или установки значений закрытых переменных-членов.

Зачем же используются в программе такие посредники между закрытыми членами класса и остальной программой? Не проще ли открыть данные класса для внешнего доступа, вместо того чтобы работать с методами доступа?

Дело в том, что применение методов доступа позволяет скрыть от пользователя детали хранения данных в объектах, в то же время снабжая их методами использования этих данных. В результате можно модернизировать способы хранения и обработки данных внутри класса, не переписывая при этом методы доступа и вызовы их во внешнем программном коде.

Если для некоторой внешней функции в программе, возвращающей возраст объекта `Cat`, открыть непосредственный доступ к переменной `itsAge`, то эту функцию пришлось бы переписывать в том случае, если автор класса `Cat` решит изменить способ хранения этого компонента данных. Однако если между внешней функцией и данными класса будет стоять функция-член `GetAge()`, то класс `Cat` можно будет модернизировать сколько угодно раз, что никак не повлияет на способ вызова функции `GetAge()` в основном коде программы. При вызове в программе метода доступа не нужно знать, хранится ли нужное значение в переменной типа `unsigned integer` или `long` либо оно вычисляется при запросе.

Такой подход облегчает эксплуатацию вашей программы и ее поддержку в будущем. Можно сказать, что он продлевает жизнь программе, поскольку, изменяя классы, можно существенно модернизировать выполнение программы, не затрагивая при этом основного кода.

В листинге 6.2 показан класс `Cat`, в котором в этот раз объявлены закрытые переменные-члены и открытые методы доступа к закрытым данным. Обратите внимание, что перед вами не выполняемый вариант программы, а только объявление класса.

Листинг 6.2. Объявление методов доступа к данным класса

```
1: // Объявление класса Cat
2: // Переменные-члены объявляются закрытыми, а открытые методы доступа
3: // обеспечивают инициализацию переменных-членов и возвращение их значений
4:
5: class Cat
6: {
7: public:
8:     // открытые методы доступа
9:   unsigned int GetAge();
10:  void SetAge(unsigned int Age);
11:
12:  unsigned int GetWeight();
13:  void SetWeight(unsigned int Weight);
14:
15:    // открытые функции-члены
```

```
16:  void Meow();
17:
18:    // закрытые переменные-члены
19: private:
20:  unsigned int itsAge;
21:  unsigned int itsWeight;
22:
23: };
```

АНАЛИЗ: Этот класс имеет пять открытых методов. В строках 9 и 10 содержатся объявления методов обеспечения доступа к переменной-члену `itsAge`. А в строках 12 и 13 объявляются методы доступа к переменной-члену `itsWeight`. Эти функции-члены инициализируют переменные и возвращают их значения.

В строке 16 объявляется открытая функция-член `Meow()`. Функция `Meow()` не является методом доступа. Она не получает и не устанавливает значение переменной-члена, а выполняет другой вид сервиса для класса, выводя слово `Meow` на экран.

Сами переменные-члены объявляются в строках 20 и 21.

Чтобы установить возраст кота `Frisky`, нужно передать соответствующее значение методу `SetAge()`:

```
Cat Frisky;
Frisky.SetAge(5);  // устанавливаем возраст Frisky с помощью открытого метода доступа
```

Ограничение доступа к данным – это не способ защиты данных, а лишь средство облегчения программирования

Объявление методов или данных закрытыми позволяет компилятору заблаговременно находить ошибки программирования. Начинающие программисты часто ошибочно полагают, что объявляя данные закрытыми, тем самым скрывают некоторую секретную информацию от пользователей, не имеющих соответствующих прав доступа. В действительности это не так. По этому поводу Страуструп (Stroustrup), изобретатель языка C++, сказал: "Механизмы управления доступом в C++ обеспечивают защиту от несчастного случая, но не от мошенника" (*ARM*, 1990).

Рекомендуется	Не рекомендуется
Объявляйте закрытыми переменные-члены класса (с помощью ключевого слова `private`). **Объявляйте** открытыми методы доступа к закрытым данным-членам класса. **Используйте** для обработки данных-членов закрытые функции-члены класса.	**Не пытайтесь** использовать закрытые переменные-члены вне класса.

Ключевое слово class

Ключевое слово class имеет следующий синтаксис:

```
class имя_класса
{
// здесь находятся ключевые слова управления доступом
// здесь объявляются переменные и методы класса
};
```

Ключевое слово class используется для объявления новых типов. Класс — это коллекция данных-членов класса, которые представляют собой переменные различных типов, включая другие классы. Класс также содержит функции класса, или методы, которые используются для выполнения действий над данными класса, а также для выполнения других видов сервиса внутри класса.

Определение объектов нового типа во многом подобно определению любых переменных. Сначала указывается тип (класс), а затем имя переменной (объект). Для обращения к членам класса (данным и функциям) используется оператор точки (.).

Для объявления открытых или закрытых разделов класса используются ключевые слова управления доступом public или private. По умолчанию действует закрытый *режим доступа*. *Каждое ключевое слово изменяет режим управления доступом с момента использования этого ключевого слова и до конца объявления класса или до тех пор, пока не встретится следующее ключевое слово управления доступом.* Все объявления классов оканчиваются закрывающей фигурной скобкой и точкой с запятой.

Пример 1:

```
class Cat
{
public:
   unsigned int Age;
   unsigned int Weight;
   void Meow();
};

Cat Frisky;
Frisky.Age = 8;
Frisky.Weight = 18;
Frisky.Meow();
```

Пример 2:

```
class Car
{
public:                              // следующие пять объявлений являются открытыми
   void Start();
   void Accelerate();
   void Brake();
   void SetYear(int year);
   int GetYear();
```

```
private:                          // все остальные объявления - закрытые
   int Year;
   Char Model [255];
};                                // конец объявления класса

Car OldFaithful;                  // создаем экземпляр класса
int bought;                       // локальная переменная типа int
OldFaithful.SetYear(84);          // присваиваем переменной число 84
bought = OldFaithful.GetYear();   // устанавливаем переменную bought равной 84
OldFaithful.Start();              // вызываем метод Start()
```

Определение методов класса

Как упоминалось выше, методы доступа обеспечивают интерфейс для работы с закрытыми переменными-членами класса. Для методов доступа, как и для всех других объявленных методов класса, следует определять выполнение. Таким образом, методы объявляются и определяются в классе.

Определение функции-члена начинается с имени класса, за которым следуют два двоеточия, имя функции и ее параметры. В листинге 6.3 показано объявление простого класса Cat, в котором присутствуют определения ранее объявленных методов доступа к данным и одной обычной функции-члена.

Листинг 6.3. Определение методов простого класса

```
1: // Пример определения методов в
2: // объявлении класса
3:
4: #include <iostream.h>        // для объекта cout
5:
6: class Cat                    // начало объявления класса
7: {
8:  public:                     // начало раздела public
9:    int GetAge0;              // метод доступа
10:   void SetAge (int age);    // метод доступа
11:   void Meow();              // обычный метод
12: private:                    // начало раздела private
13:   int itsAge;               // переменная-член
14: };
15:
16: // GetAge, открытая функция доступа,
17: // возвращает значение переменной-члена itsAge
18: int Cat::GetAge()
19: {
20:  return itsAge;
21: }
22:
23: // Определение открытой функции доступа SetAge
```

```
24: // Функция SetAge
25: // инициирует переменную-член itsAge
26: void Cat::SetAge(int age)
27: {
28:  // устанавливаем переменную-член itsAge равной
29:  // значению, переданному с помощью параметра age
30:  itsAge = age;
31: }
32:
33: // Определение метода Meow
34: // возвращает void
35: // параметров нет
36: // используется для вывода на экран текста "Meow"
37: void Cat::Meow()
38: {
39:  cout << "Meow.\n";
40: }
41:
42: // Создаем виртуальную кошку, устанавливаем ее возраст, разрешаем
43: // ей мяукнуть, сообщаем ее возраст, затем снова "мяукаем".
44: int main()
45: {
46:  Cat Frisky;
47:  Frisky.SetAge(5);
48:  Frisky.Meow();
49:  cout << "Frisky is a cat who is ";
50:  cout << Frisky.GetAge() << " years old.\n";
51:  Frisky.Meow();
52:  return 0;
53: }
```

РЕЗУЛЬТАТ

```
Meow.
Frisky is a cat who is 5 years old.
Meow.
```

АНАЛИЗ В строках 6–14 содержится определение класса `Cat`. Строку 8 занимает ключевое слово `public`, которое сообщает компилятору, что за ним следует набор открытых членов класса. В строке 9 содержится объявление открытого метода `GetAge()`, который предоставляет доступ к закрытой переменной-члену `itsAge`, объявляемой в строке 13. В строке 10 объявляется открытая функция доступа `SetAge()`, которая принимает в качестве аргумента целочисленное значение и присваивает переменной `itsAge` значение этого аргумента.

В строке 11 объявляется метод `Meow()`. Этот метод не является функцией доступа к данным-членам класса, а используется для вывода на экран слова `Meow`.

В строке 12 начинается закрытый раздел, который включает только одно объявление закрытой переменной-члена `itsAge` (строка 13). Объявление класса завершается закрывающей фигурной скобкой и точкой с запятой в строке 14.

Строки 18–21 содержат определение функции-члена `GetAge()`. Этот метод не принимает никаких параметров и возвращает целое значение. Обратите внимание на то, что

при определении методов класса используется имя класса, за которым следуют два двоеточия и имя функции (строка 18). Благодаря этому синтаксису компилятор узнает, что определяемая здесь функция GetAge() — это функция, объявленная в классе Cat. За исключением строки заголовка, GetAge() создается точно так же, как и другие функции.

Определение функции GetAge() занимает только одну строку, в которой указывается, что эта функция возвращает значение переменной-члена itsAge. Обратите внимание, что функция main() не может получить доступ к этой переменной, поскольку она объявлена в закрытом разделе класса Cat. При этом из функции main()можно обратиться к открытому методу GetAge(). А поскольку метод GetAge() является функцией-членом класса Cat, то он имеет все права доступа к переменной-члену itsAge. В результате функция GetAge() возвращает значение переменной itsAge в функцию main().

В строке 26 начинается определение функции-члена SetAge(). Она принимает целочисленный параметр и присваивает переменной itsAge значение этого параметра (строка 30). Являясь членом класса Cat, функция SetAge() имеет прямой доступ к переменной-члену itsAge.

В строке 37 начинается определение метода Meow() класса Cat. Этот метод занимает всего одну строку, в которой выводится на экран слово Meow, а затем выполняется переход на новую строку. Помните, что для перехода на новую строку используется символ \n.

В строке 44 начинается тело функции main(); она не принимает никаких аргументов. В строке 46 в функции main() объявляется объект класса Cat с именем Frisky. В строке 47 переменной-члену itsAge присваивается значение 5 с помощью метода доступа SetAge(). Обратите внимание, что в вызове этого метода указывается имя объекта (Frisky), за которым следует оператор прямого доступа (.), и имя самого метода (SetAge()). Таким способом можно вызывать любые другие методы класса.

В строке 48 вызывается функция-член Meow(), а в строке 49 на экран выводится значение переменной-члена с использованием функции доступа GetAge(). В строке 51 функция Meow() вызывается снова.

Конструкторы и деструкторы

Существует два способа определения целочисленной переменной. Во-первых, можно определить переменную, а затем (несколько ниже в программе) присвоить ей некоторое значение, например:

```
int Weight;      // определяем переменную
...              // здесь следуют другие выражения
Weight =7;       // присваиваем значение переменной
```

Можно также определить переменную и немедленно ее инициализировать, например:

```
int Weight = 7; // определяем и инициализируем значением 7
```

Операция инициализации сочетает в себе определение переменной с присвоением начального значения. Причем ничто не может помешать вам впоследствии изменить это значение. Кроме того, инициализация, проведенная одновременно с определением, гарантирует, что переменная не будет содержать мусор, оставшийся в выделенных переменной ячейках памяти.

Как же инициализировать переменные-члены класса? Для этого в классе используется специальная функция-член, называемая *конструктором*. При необходимости конструктор может принимать параметры, но не может возвращать значения даже типа `void`. Конструктор — это метод класса, имя которого совпадает с именем самого класса.

Объявив конструктор, вам также стоит объявить и деструктор. Если конструкторы служат для создания и инициализации объектов класса, то деструкторы удаляют из памяти отработавшие объекты и освобождают выделенную для них память. Деструктору всегда присваивается имя класса с символом тильды (~) вначале. Деструкторы не принимают никаких аргументов и не возвращают никаких значений. Объявление деструктора класса `Cat` будет выглядеть следующим образом:

```
~Cat();
```

Конструкторы и деструкторы, заданные по умолчанию

Если вы не объявите конструктор или деструктор, то компилятор сделает это за вас. Стандартные конструктор и деструктор не принимают аргументов и не выполняют никаких действий.

Вопросы и ответы

Конструктор называется стандартным из-за отсутствия аргументов или из-за того, что создается компилятором в том случае, если в классе не объявляется никакой другой конструктор?

Стандартный конструктор, или конструктор по умолчанию, характеризуется тем, что не принимает никаких аргументов, причем неважно, создан ли этот конструктор автоматически компилятором или самим программистом. Стандартный конструктор всегда используется по умолчанию.

Однако что касается деструкторов, то тут есть свои отличия. Стандартный деструктор предоставляется компилятором. Поскольку все деструкторы не имеют параметров, то главной отличительной чертой стандартного деструктора является то, что он не выполняет никаких действий, т.е. имеет пустое тело функции.

Использование конструктора, заданного по умолчанию

Какая же польза от конструктора, который ничего не выполняет? Зачастую это нужно только для протокола. Все объекты должны быть локализованы в программе, поэтому их создание и удаление сопровождается вызовом соответствующей функции, которая при этом может ничего и не делать. Так, для объявления объекта без передачи параметров, например

```
Cat Rags;      // Rags не получает никаких параметров
```

необходимо иметь следующий конструктор:

```
Cat();
```

Конструктор вызывается при определении объекта класса. Если для создания объекта класса `Cat` следует передать два параметра, то конструктор класса `Cat` определяется следующим образом:

```
Cat Frisky (5,7);
```

Если конструктор принимает один параметр, определение объекта будет иметь следующий вид:

```
Cat Frisky (3);
```

В случае, когда конструктор вообще не принимает параметров (т.е. является *стандартным*), отпадает необходимость использования круглых скобок:

```
Cat Frisky;
```

Этот случай является исключением из правила, гласящего, что все функции требуют наличия круглых скобок, даже если они вовсе не принимают параметров. Вот почему можно спокойно записать такое определение:

```
Cat Frisky;
```

Эта запись интерпретируется как обращение к стандартному конструктору. В ней отсутствует передача параметров и, как следствие, круглые скобки.

Обратите внимание, что вы не обязаны постоянно использовать стандартный конструктор, предоставляемый компилятором. Всегда можно написать собственный стандартный конструктор, т.е. конструктор без параметров. Вы вольны наделить свой стандартный конструктор телом функции, в котором будет выполняться инициализация класса.

Чтобы придать законченность своему труду, при объявлении конструктора не забудьте объявить и деструктор, даже если вашему деструктору нечего делать. И хотя справедливо то, что и стандартный конструктор будет корректно работать, отнюдь не повредит объявить собственный деструктор. Это сделает вашу программу более ясной.

В листинге 6.4 в знакомый уже вам класс `Cat` добавлены конструктор и деструктор. Конструктор используется для инициализации объекта `Cat` и установки его возраста равным предоставляемому вами значению. Обратите внимание на то, в каком месте программы вызывается деструктор.

Листинг 6.4. Использование конструкторов и деструкторов

```
1: // Пример объявления конструктора и
2: // деструктора в классе Cat
3:
4: #include <iostream.h>          // для объекта cout
5:
6: class Cat                     // начало объявления класса
7: {
8: public:                       // начало открытого раздела
9:  Cat(int initialAge);         // конструктор
10:  ~Cat();                     // деструктор
11:  int GetAge();               // метод доступа
12:  void SetAge(int age);       // метод доступа
13:  void Meow();
14: private:                     // начало закрытого раздела
15:  int itsAge;                 // переменная-член
16: };
17:
18: // конструктор класса Cat
19: Cat::Cat(int initialAge)
```

```
20: {
21:  itsAge = initialAge;
22: }
23:
24: Cat::~Cat()                  // деструктор, не выполняющий действий
25: {
26: }
27:
28: // GetAge, открытая функция обеспечения доступа,
29: // возвращает значение переменной-члена itsAge
30: int Cat::GetAge()
31: {
32:  return itsAge;
33: }
34:
35: // Определение SetAge, открытой
36: // функции обеспечения доступа
37:
38: void Cat::SetAge(int age)
39: {
40:  // устанавливаем переменную-член itsAge равной
41:  // значению, переданному параметром age
42:  itsAge = age;
43: }
44:
45: // Определение метода Meow
46: // возвращает void
47: // параметров нет
48: // используется для вывода на экран текста "Meow"
49: void Cat::Meow()
50: {
51:  cout << "Meow.\n";
52: }
53:
54: // Создаем виртуальную кошку, устанавливаем ее возраст, разрешаем
55: // ей мяукнуть, сообщаем ее возраст, затем снова "мяукаем" и изменяем возраст кошки.
56: int main()
57: {
58:  Cat Frisky(5);
59:  Frisky.Meow();
60:  cout << "Frisky is a cat who is ";
61:  cout << Frisky.GetAge() << " years old.\n";
62:  Frisky.Meow();
63:  Frisky.SetAge(7);
64:  cout << "Now Frisky is " ;
65:  cout << Frisky.GetAge() << " years old.\n";
66:  return 0;
67: }
```

РЕЗУЛЬТАТ

```
Meow.
Frisky is a cat who is 5 years old.
Meow.
Now Frisky is 7 years old.
```

АНАЛИЗ

Листинг 6.4 подобен листингу 6.3 за исключением того, что в строке 9 добавляется конструктор, который принимает в качестве параметра целочисленное значение. В строке 10 объявляется деструктор, который не принимает никаких параметров. Помните, что деструкторы никогда не принимают параметров; кроме того, ни конструкторы, ни деструкторы не возвращают никаких значений — даже значения типа `void`.

В строках 19–22 определяется выполнение конструктора, аналогичное выполнению функции доступа `SetAge()`, которая также не возвращает никакого значения.

В строках 24–26 определяется деструктор `~Cat()`. Эта функция не выполняет никаких действий, но коль вы объявляете ее в классе, нужно обязательно включить и ее определение.

В строке 58 содержится определение объекта класса `Cat` с именем `Frisky`. В конструктор объекта `Frisky` передается значение `5`. В данном случае нет никакой необходимости вызывать функцию-член `SetAge()`, поскольку объект `Frisky` создавался с использованием значения 5, присвоенного переменной-члену `itsAge`, как показано в строке 61. В строке 63 переменной `itsAge` объекта `Frisky` присваивается значение `7`, на этот раз с помощью функции `SetAge()`. Новое значение выводится на экран в строке 65.

Рекомендуется	Не рекомендуется
Используйте конструкторы для инициализации объектов.	**Не пытайтесь** с помощью конструктора или деструктора возвращать какое бы то ни было значение. **Не передавайте** деструкторам параметры.

Объявление функций-членов со спецификатором const

В языке C++ предусмотрена возможность объявить метод класса таким образом, что такому методу будет запрещено изменять значения переменных-членов класса. Для этого в объявлении функции используется ключевое слово `const`, стоящее после круглых скобок, но перед точкой с запятой. Например, объявим таким образом функцию-член `SomeFunction()`, которая не принимает аргументов и возвращает значение типа `void`:

```
void SomeFunction() const;
```

Функции доступа к данным часто объявляются со спецификатором `const`. В классе `Cat` есть две функции доступа:

```
void SetAge(int anAge);
int GetAge();
```

Функция SetAge() не может быть объявлена со спецификатором const, поскольку она изменяет значение переменной-члена itsAge. А в объявлении функции GetAge() может и даже должен использоваться спецификатор const, поскольку она не должна ничего изменять в классе. Функция GetAge() просто возвращает текущее значение переменной-члена itsAge. Следовательно, объявление этих функций необходимо записать в таком виде:

```
void SetAge(int anAge);
int GetAge() const;
```

Если некоторая функция объявлена с использованием спецификатора const, а в ее выполнении происходит изменение какого-либо члена объекта, то компилятор покажет сообщение об ошибке. Например, если записать функцию GetAge() таким образом, чтобы она подсчитывала, сколько раз запрашивался возраст объекта Cat, будет обязательно сгенерирована ошибка компиляции, поскольку при таком подсчете (т.е. при вызове функции GetAge()) происходит изменение объекта Cat.

Используйте спецификатор const везде в объявлениях функций-членов, если они не должны изменять объект. Это позволит компилятору лучше отслеживать ошибки и поможет вам при отладке программы.

Использовать const в объявлениях методов, не изменяющих объект, считается хорошим стилем программирования. Это позволяет компилятору лучше отслеживать ошибки еще до запуска программы на выполнение.

Чем отличается интерфейс от выполнения класса

Как уже упоминалось, клиенты — это составные части программы, которые создают и используют объекты вашего класса. Открытый интерфейс класса (объявление класса) можно представить себе в виде соглашения с этими клиентами, в котором указываются способы взаимодействия клиентов с классом.

Например, в объявлении класса Cat указывается, что программа-клиент может инициализировать любой возраст объекта этого класса с помощью функции доступа SetAge() и возвратить это значение с помощью функции доступа GetAge(). При этом гарантируется, что каждый объект класса Cat сможет вывести сообщение Meow на экран с помощью функции-члена Meow(). Обратите внимание, что в открытом интерфейсе класса ничего не говорится о закрытой переменной-члене itsAge, которая используется при выполнении класса и не должна интересовать клиентов. Значение возраста можно возвратить из объекта с помощью GetAge() и установить с помощью SetAge(), но сама переменная itsAge, в которой хранится это значение, скрыта от клиентов.

Если объявить функцию GetAge() со спецификатором const, а именно этого требуют правила хорошего тона программирования, в соглашение также будет внесен пункт о том, что функцию GetAge() нельзя использовать для изменения значений объекта класса Cat.

В языке C++ осуществляется строгий контроль за типами данных, поэтому подобное соглашение между классом и клиентами будет законом для компилятора, который сгенерирует ошибку компиляции в случае нарушения этого соглашения. В листинге 6.5 показан пример программы, которую не удастся скомпилировать из-за нарушения этих самых соглашений.

ПРЕДУПРЕЖДЕНИЕ Листинг 6.5 не компилируется!

Листинг 6.5. Пример нарушения соглашений интерфейса

```
1: // Пример ошибки компиляции, связанной
2: // с нарушениями соглашений интерфейса класса
3:
4: #include <iostream.h>   // для объекта cout
5:
6: class Cat
7:  {
8:  public:
9:    Cat(int initialAge);
10:   ~Cat();
11:   int GetAge() const; // метод доступа const
12:   void SetAge (int age);
13:   void Meow();
14: private:
15:   int itsAge;
16: };
17:
18: // конструктор класса Cat
19: Cat::Cat(int initialAge)
20: {
21:  itsAge = initialAge;
22:  cout << "Cat constructor\n";
23: }
24: Cat::~Cat() // деструктор, который не выполняет никаких действий
25: {
26:   cout << "Cat destructor\n";
27: }
28: // функция GetAge объявлена как const,
29: // но мы нарушаем это условие!
30: int Cat::GetAge() const
31: {
32:   return (itsAge++); // это нарушение соглашения интерфейса!
33: }
34:
35: // определение функции SetAge как открытого
36: // метода доступа к данным класса
37:
38: void Cat::SetAge(int age)
39: {
40:  // присваиваем переменной-члену itsAge
41:  // значение переданного параметра age
42:  itsAge = age;
43: }
```

```
44:
45: // Определение метода Meow
46: // возвращает void
47: // параметров нет
48: // используется для вывода на экран текста "Meow"
49: void Cat::Meow()
50: {
51:  cout << "Meow.\n";
52: }
53:
54: // демонстрирует различные нарушения
55: // интерфейса, что приводит к ошибкам компиляции
56: int main()
57: {
58:  Cat Frisky;     // не соответствует объявлению
59:  Frisky.Meow();
60:  Frisky.Bark();    // Нет, кошки не лают.
61:  Frisky.itsAge = 7; // переменная itsAge закрыта
62:  return 0;
63: }
```

АНАЛИЗ Как упоминалось выше, эта программа не компилируется. Поэтому и отсутствуют результаты ее работы.

Эту программу было забавно писать, поскольку в нее специально закладывались ошибки.

В строке 11 `GetAge()`объявляется как функция доступа к данным-членам класса без права их изменения, на что указывает спецификатор `const`. Однако в теле функции `GetAge()`, а именно в строке 32, выполняется приращение переменной-члена `itsAge`. А поскольку этот метод объявлен как `const`, он не имеет права изменять значение переменной `itsAge`. Следовательно, во время компиляции программы на этой строке будет зафиксирована ошибка.

В строке 13 объявляется метод `Meow()`, в этот раз без использования ключевого слова `const`. И хотя такое упущение не является ошибкой, это далеко не лучший стиль программирования. Если учесть, что этот метод не должен изменять значения переменных-членов класса `Cat`, то его следовало бы определить со спецификатором `const`.

В строке 58 показано определение объекта класса `Cat` с именем `Frisky`. В этом варианте программы класс `Cat` имеет конструктор, который принимает в качестве параметра целочисленное значение. Это означает обязательность передачи параметра заданного типа. Поскольку в строке 58 никакой параметр не передается, компилятор зафиксирует ошибку.

ПРИМЕЧАНИЕ

> Если в классе объявляется какой-либо конструктор, компилятор в этом случае не станет предлагать со своей стороны никакого другого конструктора, даже если определение объекта по форме не будет соответствовать объявленному конструктору. В подобных случаях компилятор покажет сообщение об ошибке.

В строке 60 вызывается метод `Bark()`. Этот метод вообще не был объявлен, следовательно, ни о каком его использовании и речи быть не может.

В строке 61 делается попытка присвоить переменной `itsAge` значение 7. Поскольку переменная `itsAge` относится к числу закрытых данных-членов, то при компиляции программы здесь будет зафиксировано покушение на частную собственность класса.

Почему для отслеживания ошибок лучше использовать компилятор

Кажется невероятным написать программу, не допуская никаких ошибок. Тем не менее некоторые программисты способны на подобные чудеса, хотя, конечно, таких кудесников очень немного. Большинство из них, как и все нормальные люди, делают ошибки. Поэтому нашлись программисты, которые разработали систему, способную помочь в отслеживании ошибок путем перехвата и исправления их на ранней стадии создания программ.

Хотя сообщения об ошибках, выявленных компилятором, действуют на нервы, это намного лучше возникновения ошибок при выполнении программы. Если бы компилятор был менее дотошный, то велика вероятность, что ваша программа дала бы сбой в самый неподходящий момент, например во время презентации.

Ошибки компиляции, т.е. ошибки, выявленные на стадии компиляции, гораздо безобиднее ошибок выполнения, которые проявляются после запуска программы. Компилятор будет дотошно и однотипно сообщать об обнаруженной им ошибке. Напротив, ошибка выполнения может не обнаруживать себя до поры до времени, но потом проявиться в самый неподходящий момент. Поскольку ошибки компиляции заявляют о себе при каждом сеансе компиляции, то их легко идентифицировать и исправить, чтобы больше о них не вспоминать. Чтобы добиться создания программ, которые не станут со временем выкидывать фокусы, программист должен помочь компилятору в отслеживании ошибок, используя спецификаторы в объявлениях для предупреждения возможных сбоев.

Где следует располагать в программе объявления классов и определения методов

Каждая функция, объявленная в классе, должна иметь определение. Определение также называется выполнением функции. Подобно другим функциям, определение метода класса состоит из заголовка и тела функции.

Определение должно находиться в файле, который компилятор может легко найти. Большинство компиляторов C++ предпочитают, чтобы такой файл имел расширение `.c` или `.cpp`. В этой книге используется расширение `.cpp`, но вам стоит выяснить предпочтения собственного компилятора.

Многие компиляторы полагают, что файлы с расширением `.c` содержат программы, написанные на языке C, а файлы с расширением `.cpp` — программы на C++. Вы можете использовать любое расширение, но именно `.cpp` сведет к минимуму возможные недоразумения в программах на C++.

Объявления классов можно поместить в один файл с программой, но это не считается хорошим стилем программирования. В соглашении, которого придерживаются многие программисты, рекомендуется помещать объявление в файл заголовка, имя которого обычно совпадает с именем файла программы, но в качестве расширения используются такие варианты, как .h, .hp или .hpp. В этой книге для имен файлов заголовков используется расширение .hpp, но вам стоит выяснить предпочтения собственного компилятора.

Например, можно поместить объявление класса Cat в файл с именем CAT.hpp, а определение методов класса — в файл с именем CAT.cpp. Затем нужно включить файл заголовка в код файла с расширением .cpp. Для этого в начале программного кода в файле CAT.cpp используется следующая команда:

```
#include "Cat.hpp"
```

Эта команда дает указание компилятору ввести содержимое файла CAT.hpp в данном месте программы. Результат выполнения команды include такой же, как если бы вы переписали с клавиатуры в это место программы полное содержимое соответствующего файла заголовка. Имейте в виду, что некоторые компиляторы чувствительны к регистру букв и требуют точного соответствия написания имен файла в директиве #include и на диске.

Зачем же нужно отделять файл заголовка с расширением .hpp от файла программы с расширением .cpp, если мы все равно собираемся вводить содержимое файла заголовка назад в файл программы? Как показывает практика, большую часть времени клиентов вашего класса не волнуют подробности его выполнения. При чтении небольшого файла заголовка они получают всю необходимую информацию и могут игнорировать файл с подробностями выполнения этого класса. Кроме того, не исключено, что содержимое файла заголовка с расширением .hpp вам захочется включить не в один, а в несколько файлов программ.

Объявление класса сообщает компилятору, что представляет собой этот класс, какие данные он содержит и какими функциями располагает. Объявление класса называется его интерфейсом, поскольку оно сообщает пользователю, как взаимодействовать с классом. Интерфейс обычно хранится в файле с расширением .hpp, который называется файлом заголовка.

Из определения функции компилятор узнает, как она работает. Определение функции называется выполнением метода класса и хранится в файле с расширением .cpp. Подробности выполнения класса касаются только автора класса. Клиентам же класса, т.е. частям программы, использующим этот класс, не нужно знать, как выполняются функции.

Выполнение с подстановкой

Можно выполнять подстановку методов с помощью ключевого слова inline точно так же, как это делалось с обычными функциями. Для этого ключевое слово inline нужно разместить перед типом возвращаемого значения. Например, определение подставляемой функции-члена GetWeight() имеет следующий вид:

```
inline int Cat::GetWeight()
{
return itsweight; // возвращает переменную-член Weight
}
```

Можно также поместить определение функции в объявление класса, что автоматически делает такую функцию подставляемой:

```
class Cat
{
public:
int GetWeight() { return itsWeight; }  // подставляемая функция
void SetWeight(int aWeight);
};
```

Обратите внимание на синтаксис определения функции `GetWeight()`. Тело подставляемой функции начинается сразу же после объявления метода класса, причем после круглых скобок нет никакой точки с запятой. Подобно определению обычной функции, определение метода начинается с открывающей фигурной скобки и оканчивается закрывающей фигурной скобкой. Как обычно, пробелы значения не имеют, и то же самое определение можно записать несколько иначе:

```
class Cat
{
public:
int GetWeight() const
{
return itsWeight;
}                                    // подставляемая функция
void SetWeight(int aWeight);
};
```

В листингах 6.6 и 6.7 вновь создается класс `Cat`, но в данном случае объявление класса содержится в файле `CAT.hpp`, а выполнение — в файле `CAT.cpp`. Кроме того, в листинге 6.7 метод доступа к данным класса и метод `Meow()` являются подставляемыми.

Листинг 6.6. Объявление класса `CAT` в файле `CAT.hpp`

```
1: #include <iostream.h>
2: class Cat
3: {
4:  public:
5:        Cat (int initialAge);
6:        ~Cat();
7:        int GetAge() const { return itsAge;}       // подставляемая функция!
8:        void SetAge (int age) { itsAge = age;}     // подставляемая функция!
9:        void Meow() const { cout << "Мяу.\n";}     // подставляемая функция!
10: private:
11:       int itsAge;
12: };
```

Листинг 6.7. Выполнение класса Cat в файле CAT.cpp

```
1:  // Пример использования подставляемых функций
2:  // и включения файла заголовка
3:
4:  #include "cat.hpp" // не забудьте включить файл заголовка!
5:
6:
7:  Cat::Cat(int initialAge)  //конструктор
8:  {
9:   itsAge = initialAge;
10: }
11:
12: Cat::~Cat() // деструктор, не выполняет никаких действий
13: {
14: }
15:
16: // Создаем виртуальную кошку, устанавливаем ее возраст, разрешаем
17: // ей мяукнуть, сообщаем ее возраст, затем снова "мяукаем" и изменяем возраст кошки.
18: int main()
19: {
20:  Cat Frisky(5);
21:  Frisky.Meow();
22:  cout << "Frisky is a cat who is ";
23:  cout << Frisky.GetAge() << " years old.\n";
24:  Frisky.Meow();
25:  Frisky.SetAge(7);
26:  cout << "Now Frisky is " ;
27:  cout << Frisky.GetAge() << " years old.\n";
28:  return 0;
29: }
```

РЕЗУЛЬТАТ

```
Meow.
Frisky is a cat who is 5 years old.
Meow.
Now Frisky is 7 years old.
```

АНАЛИЗ Программа, представленная в листингах 6.6 и 6.7, аналогична программе из листинга 6.4 за исключением того, что три метода класса объявляются подставляемыми, а само объявление класса вынесено в файл заголовка CAT.hpp.

В строке 7 объявляется функция GetAge() и тут же следует определение ее выполнения. Строки 8 и 9 занимают объявления еще двух встроенных функций, но их определения содержатся в другом файле.

В строке 4 листинга 6.7 с помощью команды #include "cat.hpp" в программу включается содержимое файла CAT.hpp. Компилятор получает команду считать содержимое файла cat.hpp и ввести его в данный файл, начиная со строки 5.

Возможность встраивания файлов в другие файлы позволяет хранить объявления классов отдельно от их выполнения и использовать тогда, когда в этом возникает необходимость. Это стандартный прием при создании программ на языке C++. Обычно объявления классов хранятся в файле с расширением .hpp, который затем включается в соответствующий файл .cpp с помощью директивы #include.

В строках 18–29 в точности повторяется тело функции `main()` из листинга 6.4. Цель этого повторения — показать, что применение подставляемых вариантов функций не внесло изменений в использование этих функций.

Классы, содержащие другие классы в качестве данных-членов

Нет ничего необычного в построении сложного класса путем объявления более простых классов и последующего включения их в объявление сложного класса. Например, можно объявить класс колеса, класс мотора, класс коробки передач и т.д., а затем объединить их в класс автомобиля. Тем самым объявляются и взаимоотношения между классами. Автомобиль имеет мотор, колеса и коробку передач.

Рассмотрим второй пример. Прямоугольник состоит из линий. Линия определяется двумя точками. Каждая точка определяется координатами `x` и `y`. В листинге 6.8 показано объявление класса `Rectangle`, которое содержится в файле `RECTANGLE.hpp`. Поскольку прямоугольник определяется четырьмя линиями, соединяющими четыре точки, и каждая точка имеет координаты на графике, то сначала будет объявлен класс `Point` для хранения координат `x,y` каждой точки. Листинг 6.9 содержит объявления обоих классов.

Листинг 6.8. Объявление классов точки и прямоугольника

```
1:  // Начало файла Rect.hpp
2:  #include <iostream.h>
3:  class Point   // хранит координаты x,y
4:  {
5:   // нет конструктора, используется конструктор по умолчанию
6:   public:
7:        void SetX(int x) { itsX = x; }
8:        void SetY(int y) { itsY = y; }
9:        int GetX() const { return itsX;}
10:       int GetY() const { return itsY;}
11:  private:
12:       int itsX;
13:       int itsY;
14: };     // конец объявления класса Point
15:
16:
17: class Rectangle
18: {
19:  public:
20:       Rectangle (int top, int left, int bottom, int right);
21:       ~Rectangle () {}
22:
23:       int GetTop() const { return itsTop; }
24:       int GetLeft() const { return itsLeft; }
25:       int GetBottom() const { return itsBottom; }
26:       int GetRight() const { return itsRight; }
27:
```

```
28:       Point GetUpperLeft() const { return itsUpperLeft; }
29:       Point GetLowerLeft() const { return itsLowerLeft; }
30:       Point GetUpperRight() const { return itsUpperRight; }
31:       Point GetLowerRight() const { return itsLowerRight; }
32:
33:       void SetUpperLeft(Point Location) {itsUpperLeft = Location;}
34:       void SetLowerLeft(Point Location) {itsLowerLeft = Location;}
35:       void SetUpperRight(Point Location) {itsUpperRight = Location;}
36:       void SetLowerRight(Point Location) {itsLowerRight = Location;}
37:
38:       void SetTop(int top) { itsTop = top; }
39:       void SetLeft (int left) { itsLeft = left; }
40:       void SetBottom (int bottom) { itsBottom = bottom; }
41:       void SetRight (int right) { itsRight = right; }
42:
43:       int GetArea() const;
44:
45:  private:
46:       Point itsUpperLeft;
47:       Point itsUpperRight;
48:       Point itsLowerLeft;
49:       Point itsLowerRight;
50:       int itsTop;
51:       int itsLeft;
52:       int itsBottom;
53:       int itsRight;
54: };
55: // конец файла Rect.hpp
```

Листинг 6.9. Содержимое файла RECT.cpp

```
1:  // Начало файла rect.cpp
2:  #include "rect.hpp"
3:  Rectangle::Rectangle(int top, int left, int bottom, int right)
4:  {
5:   itsTop = top;
6:   itsLeft = left;
7:   itsBottom = bottom;
8:   itsRight = right;
9:
10:  itsUpperLeft.SetX(left);
11:  itsUpperLeft.SetY(top);
12:
13:  itsUpperRight.SetX(right);
14:  itsUpperRight.SetY(top);
15:
16:  itsLowerLeft.SetX(left);
17:  itsLowerLeft.SetY(bottom);
18:
```

```
19:  itsLowerRight.SetX(right);
20:  itsLowerRight.SetY(bottom);
21: }
22:
23:
24: // Вычисляем площадь прямоугольника, отыскивая его стороны,
25: // определяем его длину и ширину, а затем перемножаем их
26: int Rectangle::GetArea() const
27: {
28:  int Width = itsRignt - itsLeft;
29:  int Height = itsTop - itsBottom;
30:  return (Width * Height);
31: }
32:
33: int main()
34: {
35:  //инициализируем локальную переменную Rectangle
36:  Rectangle MyRectangle (100, 20, 50, 80 );
37:
38:  int Area = MyRectangle.GetArea();
39:
40:  cout << "Area: "<< Area << "\n";
41:  cout << "Upper Left X Coordinate:";
42:  cout << MyRectangle.GetUpperLeft().GetX();
43:  return 0;
44: }
```

РЕЗУЛЬТАТ

```
Area: 3000
Upper Left X Coordinate: 20
```

АНАЛИЗ

В строках 3–14 листинга 6.8 объявляется класс `Point`, который используется для хранения конкретных координат `x,y` на графике. В данной программе класс `Point` практически не используется. Однако в других методах рисования он незаменим.

Внутри объявления класса `Point` (в строках 12 и 13) объявляются две переменные-члена (`itsX` и `itsY`). Эти переменные хранят значения координат точки. При увеличении координаты `x` мы перемещаемся на графике вправо. При увеличении координаты `y` мы перемещаемся на графике вверх. В других графиках могут использоваться другие системы координат (с другой ориентацией). Например, в некоторых программах построения окон значение координаты `y` увеличивается при перемещении в области окна вниз.

В классе `Point` используются подставляемые `inline`-функции доступа, предназначенные для чтения и установки координат точек `X` и `Y`. Эти функции объявляются в строках 7–10. В объектах класса `Point` используются стандартные конструктор и деструктор, предоставляемые компилятором по умолчанию. Следовательно, координаты точек должны устанавливаться в программе.

В строке 17 начинается объявление класса `Rectangle`, который включает четыре точки, представляющие углы прямоугольника.

Конструктор класса `Rectangle` принимает четыре целочисленных параметра, именуемых `top` (верхний), `left` (левый), `bottom` (нижний) и `right` (правый). Эти четыре па-

раметра, передаваемые конструктору, копируются в соответствующие четыре переменные-члена (см. листинг 6.9), после чего устанавливаются четыре точки (четыре объекта класса Point).

Помимо обычных функций доступа к данным-членам класса, в классе Rectangle предусмотрена функция GetArea(), объявленная в строке 43. Вместо хранения значения площади в виде переменной эта функция вычисляет площадь в строках 28 и 29 листинга 6.9. Для этого сначала вычисляются значения длины и ширины прямоугольника, а затем полученные результаты перемножаются.

Для получения координаты верхнего левого угла прямоугольника нужно получить доступ к точке UpperLeft и запросить ее значение X. Поскольку функция GetUpperLeft() является методом класса Rectangle, она может непосредственно получить доступ к закрытым данным этого класса, включая и доступ к переменной itsUpperLeft. Поскольку переменная itsUpperLeft является объектом класса Point, а переменная itsX этого объекта закрытая, функция GetUpperLeft() не может прямо обратиться к этой переменной. Вместо этого для получения значения переменной itsX она должна использовать открытую функцию доступа GetX().

В строке 33 листинга 6.9 начинается тело основной части программы. До выполнения строки 36 никакой памяти не выделялось и ничего, по сути, не происходило. Все, сделанное до сих пор, служило одной цели — сообщить компилятору, как создается точка и как создается прямоугольник (на случай, если в этом появится необходимость).

В строке 36 определяется прямоугольник (объект класса Rectangle) путем передачи реальных значений для параметров Top, Left, Bottom и Right.

В строке 37 создается локальная переменная Area типа int. Она предназначена для хранения площади созданного прямоугольника. Переменной Area присваивается значение, возвращаемое функцией-членом GetArea() класса Rectangle.

Клиент класса Rectangle может создать объект Rectangle и возвратить его площадь, не заботясь о нюансах выполнения функции GetArea().

В листинге 6.8 показано содержимое заголовочного файла Rect.hpp. Только лишь просмотрев заголовочный файл, который содержит объявление класса Rectangle, программист будет знать, что функция GetArea() возвращает значение типа int. Пользователя класса Rectangle не волнуют "производственные" секреты функции GetArea(). И в самом деле, автор класса Rectangle мог бы спокойно изменить выполнение функции GetArea(), и это бы не повлияло на программы, использующие класс Rectangle.

Вопросы и ответы

Каково различие между объявлением и определением?

Объявление вводит имя некоторого объекта, но не выделяет для него память, а вот с помощью определения как раз и выделяется память для конкретного объекта.

Структуры

Очень близким родственником ключевого слова class является ключевое слово struct, которое используется для объявления структуры. В языке C++ структура — это тот же класс, но с открытыми по умолчанию членами. Структуру можно объявить подобно тому, как объявляется класс, наделив ее такими же переменными-членами и

функциями. И в самом деле, если исповедовать хороший стиль программирования и всегда в явном виде объявлять открытые и закрытые разделы класса, то никаких отличий не должно быть.

Попытаемся повторно ввести содержимое листинга 6.8 с учетом следующих изменений:

- в строке 3 заменим объявление `class Point` объявлением `struct Point`;
- в строке 17 заменим объявление `class Rectangle` объявлением `struct Rectangle`.

Теперь вновь запустим нашу программу и сравним результаты. При этом никакой разницы вы заметить не должны.

Почему два ключевых слова несут одинаковую смысловую нагрузку

Вы, вероятно, удивлены тем, что два различных ключевых слова создают практически идентичные объявления. Так сложилось исторически. Язык C++ строился как расширение C. В языке C были структуры, но эти структуры не имели методов класса. Создатель C++, Бьерн Строустрап, опирался на структуры, но заменил имя типа данных `struct` типом `class`, чтобы тем самым заявить о новых расширенных функциональных возможностях этого нового образования.

Рекомендуется

Помещайте объявление класса в файл с расширением `.hpp`, а его выполнение — в файл с расширением `.cpp`.

Рекомендуется

Используйте спецификатор `const` везде, где это возможно.

Убедитесь, что вам полностью понятны классы, прежде чем переходить к следующему занятию.

Резюме

Сегодня вы научились создавать новые типы данных, именуемые классами. Вы узнали, как определять переменные этих новых типов, которые называются объектами.

Класс содержит данные-члены, которые представляют собой переменные различных типов, включая другие классы. Кроме того, в состав класса входят функции-члены, известные также как методы. Эти функции-члены используются для выполнения действий над данными-членами и обеспечения иного сервиса.

Члены класса — как данные, так и функции — могут быть открытыми и закрытыми. Открытые члены доступны для любой части программы, а закрытые — только для функций-членов данного класса.

Хорошим стилем программирования считается вынесение интерфейса, или объявления класса, в файл заголовка, который обычно имеет расширение `.hpp`. Выполнение класса записывается в файл с расширением `.cpp`.

Для инициализации объектов используются конструкторы класса. Когда эти объекты больше не нужны, они удаляются с помощью деструкторов, которые используются для освобождения памяти, выделенной для этих объектов методами класса.

Вопросы и ответы

Как определяется размер объекта класса?

Размер объекта класса в памяти определяется суммой размеров переменных-членов. Методы класса не занимают место в области памяти, выделенной для объекта.

Некоторые компиляторы так располагают переменные в памяти, что двухбайтовые переменные в действительности занимают несколько больше двух байтов памяти. При желании вы можете уточнить этот момент в документации на свой компилятор, но на данном этапе эти подробности, по всей вероятности, не будут иметь для вас большого значения.

Если объявить класс `Cat` с закрытым членом `itsAge`, а затем определить два объекта класса `Cat` с именами `Friscky` и `Boots`, то может ли объект `Boots` получить доступ к переменной-члену `itsAge` объекта `Friscky`?

Да. Закрытые данные доступны для функций-членов класса, и различные экземпляры одного класса могут обращаться к данным друг друга. Иными словами, если `Friscky` и `Boots` являются экземплярами класса `Cat`, то функции-члены объекта `Friscky` могут получить доступ как к своим данным (данным объекта `Friscky`), так и к данным объекта `Boots`.

Почему не следует делать все данные-члены открытыми?

Объявление данных-членов закрытыми позволяет клиенту класса использовать данные, не волнуясь о том, как они хранятся или вычисляются. Например, если класс `Cat` имеет метод `GetAge()`, клиенты класса `Cat` могут возвратить значение возраста кошки (объекта класса `Cat`), не заботясь о том, хранится ли оно в какой-нибудь переменной-члене определенного типа или вычисляется по запросу.

Если применение функции `const` для изменения класса вызывает ошибку компилятора, то почему бы просто не использовать ключевое слово `const` и тем самым гарантированно избежать сообщений об ошибках?

Если ваша функция-член логически не должна изменять класс, то использование ключевого слова `const` — прекрасный способ заручиться поддержкой компилятора при отыскании случайных ошибок. Например, у функции `GetAge()` нет видимых причин для изменения класса `Cat`, но в выполнении класса может присутствовать следующая строка:

```
if (itsAge = 100) cout << "Ого! Тебе уже сто лет\n";
```

Объявление функции `GetAge()` с использованием ключевого слова `const` заставило бы компилятор обнаружить ошибку. Вы ведь имели в виду сравнение значения переменной `itsAge` с числом 100, а вместо этого случайно выполнили операцию присвоения числа 100 переменной `itsAge`. Поскольку это присвоение изменяет класс, а вы (с помощью ключевого слова `const`) заявили, что этот метод не будет изменять класс, компилятор смог найти ошибку.

Ошибки такого рода, как правило, трудно найти простым просмотром текста программы. Мы часто видим то, что хотим увидеть. Гораздо опаснее, если на первый взгляд вам покажется, что программа работает правильно (даже после установки такого странного значения), но рано или поздно эта неприятность превратится в проблему.

Существует ли резон использовать структуры в программах на C++?

Многие программисты используют ключевое слово `struct` для классов, которые не имеют функций. Можно расценивать это как ностальгию по устаревшим структурам языка C, которые не могли иметь функций. Лично я считаю это ненужным и даже плохим стилем программирования. Ведь если сегодня данной структуре не нужны ме-

тоды, то не исключено, что они могут понадобиться ей завтра. И тогда вам придется либо заменять этот тип классом, либо нарушать свое же правило и работать со структурой, которая "не брезгует" присутствием в ней методов.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Что представляет собой оператор прямого доступа и для чего он используется?
2. Что резервирует память — объявление или определение?
3. Объявление класса является его интерфейсом или выполнением?
4. Какова разница между открытыми (`public`) и закрытыми (`private`) данными-членами?
5. Могут ли функции-члены быть закрытыми?
6. Могут ли переменные-члены быть открытыми?
7. Если объявить два объекта класса `Cat`, могут ли они иметь различные значения их переменных-членов `itsAge`?
8. Нужно ли объявления класса завершать точкой с запятой? А определения методов класса?
9. Как бы выглядел заголовок функции-члена `Meow` класса `Cat`, которая не принимает никаких параметров и возвращает значение типа `void`?
10. Какая функция вызывается для выполнения инициализации класса?

Упражнения

1. Напишите программу, которая объявляет класс с именем `Employee` (Служащие) с такими переменными-членами: `age` (возраст), `yearsOfService` (стаж работы) и `Salary` (зарплата).
2. Перепишите класс `Employee`, чтобы сделать данные-члены закрытыми и обеспечить открытые методы доступа для чтения и установки всех данных-членов.
3. Напишите программу с использованием класса `Employee`, которая создает два объекта класса `Employee`; устанавливает данные-члены `age`, `YearsOfService` и `Salary`, а затем выводит их значения.
4. На основе программы из упражнения 3 создайте метод класса `Employee`, который сообщает, сколько тысяч долларов зарабатывает служащий, округляя ответ до 1 000 долларов.
5. Измените класс `Employee` так, чтобы можно было инициализировать данные-члены `age`, `YearsOfService` и `Salary` в процессе "создания" служащего.

6. **Жучки:** что неправильно в следующем объявлении?

```
class Square
{
public:
        int Side;
}
```

7. **Жучки:** что весьма полезное отсутствует в следующем объявлении класса?

```
class Cat
{
        int GetAge() const;
private:
        int itsAge;
};
```

8. **Жучки:** какие три ошибки обнаружит компилятор в этом коде?

```
class TV
{
public:
        void SetStation(int Station);
        int GetStation() const;
private:
        int itsStation;
};
main()
{
        TV myTV;
        myTV.itsStation = 9;
        TV.SetStation(10);
        TV myOtherTv(2);
}
```

Циклы

Структура любой программы состоит из комбинации множества ветвлений и циклов. На четвертом занятии вы научились организовывать ветвление программы с помощью оператора `if`. Сегодня вы узнаете:

- Что такое циклы и как они используются
- Каковы методы организации циклов
- Как избежать чрезмерной вложенности конструкций `if/else`

Организация циклов

Для решения ряда задач часто требуется многократное выполнение одних и тех же действий. На практике это реализуется с помощью рекурсивных (см. занятие 5) или итеративных алгоритмов. Суть итеративного процесса заключается в повторении последовательности операций нужное количество раз.

История оператора goto

В те годы, когда программирование находилось еще на начальной стадии развития, использовались только небольшие по размеру и достаточно примитивные программы. Нельзя было назвать приятным и сам процесс их разработки. В таких программах циклы состояли из метки, последовательности команд и оператора безусловного перехода.

В C++ меткой называют идентификатор, за которым следует двоеточие (:). Метка всегда устанавливается перед оператором, на который необходимо будет передать управление. Для перехода на нужную метку используется оператор `goto`, за которым следует имя метки. Пример использования оператора `goto` приведен в листинге 7.1.

Листинг 7.1. Организация цикла с помощью оператора `goto`

```
1:    // Листинг 7.1.
2:    // Организация цикла с помощью goto
3:
4:    #include <iostream.h>
```

```
5:
6:    int main()
7:    {
8:          int counter = 0; // инициализация счетчика
9:    loop: counter ++;      // начало цикла
10:         cout << "counter: " << counter << "\ n";
11:         if (counter < 5) // проверка значения
12:           goto loop;     // возвращение к началу
13:
14:         cout << "Complete. Counter: " << counter << ".\ n";
15:         return 0;
16:   }
```

Результаты:

```
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Complete. Counter: 5.
```

Анализ: В строке 8 переменная `counter` инициализируется нулевым значением. Метка `loop:` в строке 9 показывает начало цикла. На каждой итерации значение `counter` увеличивается на единицу и выводится на экран. В строке 11 выполняется проверка значения переменной `counter`. Если оно меньше пяти, значит условие выполняется и управление передается оператору `goto`, в результате чего осуществляется переход на строку 9. Итеративный процесс выполняется до тех пор, пока значение переменной `counter` не достигнет пяти. После этого программа выходит за пределы цикла и на экран выводится окончательный результат.

Почему следует избегать оператора goto

Со временем нелестные высказывания в адрес оператора `goto` участились, впрочем, вполне заслуженно. С помощью оператора `goto` можно осуществлять переход в любую точку программы — вперед или назад. Такое беспорядочное использование этого оператора привело к появлению запутанных и абсолютно непригодных для восприятия программ, получивших жаргонное название "спагетти". Поэтому последние двадцать лет преподаватели программирования во всем мире твердили студентам одну и ту же фразу: "Никогда не используйте оператор `goto`".

На смену оператору `goto` пришли конструкции с несколько более сложной структурой, но и с более широкими возможностями: `for`, `while` и `do...while`. Несмотря на то что после полного искоренения оператора `goto` структура программ значительно прояснилась, негативные высказывания в его адрес следует признать преувеличенными. Как любой инструмент программирования, при правильном использовании оператор `goto` может оказаться достаточно полезным. В силу этого комитет ANSI принял решение оставить этот оператор в языке. Правда, вместе с этим родилась шутка: "Дети! Использование этого оператора в домашних условиях небезопасно!"

Организация циклов с помощью оператора while

В циклах, организованных с помощью оператора `while`, выполнение последовательности операций продолжается до тех пор, пока условие продолжения цикла истинно. В примере программы в листинге 7.1 значение переменной `counter` увеличивалось до тех пор, пока не стало равным пяти. Листинг 7.2 демонстрирует тот же алгоритм, реализованный с помощью оператора `while`.

Листинг 7.2. Организация цикла с помощью оператора `while`

```
1:    // Листинг 7.2.
2:    // Организация цикла с помощью оператора while
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:         int counter = 0;          // присвоение начального значения
9:
10:        while(counter < 5)        // проверка условия продолжения цикла
11:        {
12:             counter++;           // тело цикла
13:             cout << " counter: " << counter << "\ n";
14:        }
15:
16:    cout << " Complete. Counter: " << counter << ".\ n";
17:    return 0;
18:    }
```

РЕЗУЛЬТАТ

```
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Complete. Counter: 5.
```

АНАЛИЗ Эта несложная программа показывает пример организации цикла с помощью оператора `while`. В начале каждой итерации проверяется условие, и, если оно выполняется, управление передается на первый оператор цикла. В нашем примере условию продолжения цикла удовлетворяют все значения переменной `counter`, меньшие пяти (строка 10). Если условие выполняется, запускается следующая итерация цикла. В строке 12 значение счетчика увеличивается на единицу, а в строке 13 выводится на экран. Как только значение счетчика достигает пяти, тело цикла (строки 11–14) пропускается и управление передается в строку 15.

Сложные конструкции с оператором while

Сложность логического выражения, являющегося условием в операторе `while`, не ограничена. Это позволяет использовать в конструкции `while` любые логические выражения C++. При построении выражений допускается использование логических операций: `&&` (логическое И), `||` (логическое ИЛИ), а также `!` (логическое отрицание). В листинге 7.3 показан пример использования более сложных условий в конструкциях с оператором `while`.

Листинг 7.3. Сложные условия в конструкциях `while`

```
1:    // Листинг 7.3.
2:    // Сложные условия в конструкциях while
3:
4:    include <iostream.h>
5:
6:    int main()
7:    {
8:          unsigned short small;
9:          unsigned long  large;
10:         const unsigned short MAXSMALL=65535;
11:
12:     cout << "Enter a small number: ";
13:     cin >> small;
14:     cout << "Enter a large number: ";
15:     cin >> large;
16:
17:     cout << "small: " << small << "...";
18:
19:     // на каждой итерации проверяются три условия
20:     while (small < large && large > 0 && small < MAXSMALL)
22:     {
23:        if (small % 5000 == 0)  // после каждых 5000 строк выводится точка
24:          cout << ".";
25:
26:        small++;
27:
28:        large-=2;
29:     }
30:
31:     cout << "\ nSmall: " << small << " Large: " << large << endl;
32:     return 0;
33:   }
```

Результат

```
Enter a small number: 2
Enter a large number: 100000
small: 2.........
Small: 33335 Large: 33334
```

АНАЛИЗ Программа представляет собой простую логическую игру. Вначале предлагается ввести два числа — small и large. После этого меньшее значение увеличивается на единицу, а большее уменьшается на два до тех пор, пока они не "встретятся". Цель игры: угадать число, на котором значения "встретятся".

В строках 12–15 осуществляется ввод значений. В строке 20 проверяется три условия продолжения цикла.

1. Значение переменной small не превышает значения large.
2. Значение переменной large неотрицательное и не равно нулю.
3. Значение переменной small не превышает значения константы MAXSMALL.

Далее, в строке 23, вычисляется остаток от деления числа small на 5000, причем значение переменной small не изменяется. Если small делится на 5000 без остатка, результатом выполнения этой операции будет 0. В этом случае для визуального представления процесса вычислений на экран выводится точка. Затем в строке 26 значение переменной small увеличивается на 1, а в строке 28 значение large уменьшается на 2.

Цикл завершается, если хотя бы одно из условий перестает выполняться. После этого управление передается в строку 29, следующую за телом цикла.

Операторы break и continue

Часто бывает необходимо перейти на следующую итерацию цикла еще до завершения выполнения всех операторов тела цикла. Для этого используется оператор continue.

Кроме того, в ряде случаев требуется выйти за пределы цикла, даже если условия продолжения цикла выполняются. В этом случае используется оператор break.

Пример использования этих операторов приведен в листинге 7.4. Это несколько усложненный вариант уже знакомой игры. В этом случае, кроме меньшего и большего значений, предлагается ввести шаг и целевое значение. Как и в предыдущем примере, на каждой итерации цикла значение переменной small увеличивается на единицу. Значение large уменьшается на два, если меньшее число не кратно значению переменной шага (skip). Игра заканчивается, когда значение переменой small становится больше, чем значение large. Если значение переменной large совпадает с целевым значением (target), выводится сообщение и игра прерывается.

Цель игры состоит в том, чтобы угадать число, в которое "попадет" значение target.

Листинг 7.4. Использование break **и** continue

```
1:    // Листинг 7.4.
2:    // Пример использования операторов break и continue
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:       unsigned short small;
9:       unsigned long  large;
10:      unsigned long  skip;
11:      unsigned long target;
12:      const unsigned short MAXSMALL=65535;
13:
```

```
14:       cout << "Enter a small number: ";
15:       cin >> small;
16:       cout << "Enter a large number: ";
17:       cin >> large;
18:       cout << "Enter a skip number: ";
19:       cin >> skip;
20:       cout << "Enter a target number: ";
21:       cin >> target;
22:
23:       cout << "\ n"
24:
25:       // установка условий продолжения цикла
26:       while (small < large && large > 0 && small < MAXSMALL)
27:
28:       {
29:
30:            small++;
31:
32:            if (small % skip == 0)  // уменьшить значение large?
33:            {
34:              cout << "skipping on:" << small << endl;
35:              continue;
36:            }
37:
38:       if (large == target)     // проверка попадания в цель
39:            {
40:                 cout << " Target reached!";
41:                 break;
42:            }
43:
44:       large-=2;
45:       }     // конец цикла
46:
47:       cout << "\ nSmall: " << small << " Large: " << large << endl;
48:     return 0;
49:     }
```

РЕЗУЛЬТАТ

```
Enter a small number: 2
Enter a large number: 20
Enter a skip number: 4
Enter a target number: 6

skipping on 4
skipping on 8

Small: 10 Large: 8
```

Как видим, игра закончилась поражением пользователя, поскольку меньшее значение превысило большее, а цель так и не была достигнута.

В строке проверяются условия продолжения цикла. Если значение переменной small меньше значения large, а также если large больше нуля и small не превышает значение константы SMALLINT, управление передается первому оператору тела цикла.

В строке 32 вычисляется остаток от деления значения переменной small на значение skip. Если значение small кратно skip, оператор continue запускает следующую итерацию цикла (срока 26). В результате такого перехода пропускается проверка целевого значения и операция уменьшения значения переменной large.

Сравнение значений target и large выполняется в строке 38. Если эти значения равны, игра заканчивается победой пользователя. В этом случае программа выводит сообщение о победе, работа цикла прерывается оператором break и управление передается в строку 46.

Использование конструкции while(true)

В качестве условия, проверяемого при переходе на очередную итерацию цикла, может выступать любое выражение, корректное с точки зрения синтаксиса языка C++. Цикл выполняется до тех пор, пока это выражение истинно. Для организации так называемых бесконечных циклов в качестве такого выражения применяется логическая константа true. Листинг 7.5 демонстрирует пример бесконечного цикла, выполняющего счет до десяти.

Листинг 7.5. Еще один пример использования оператора `while`

```
1:    // Листинг 7.5.
2:    // Пример "бесконечного" цикла
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:       int counter = 0;
9:
10:      while (true)
11:      {
12:         counter ++;
13:         if (counter > 10)
14:              break;
15:      }
16:      cout << "Counter: " << counter << "\ n";
17:      return 0;
18:   }
```

РЕЗУЛЬТАТ

```
Counter: 11
```

АНАЛИЗ Понятно, что условие продолжения цикла, заданное в строке 10, будет выполняться всегда. В теле цикла (строка 12) значение переменной counter увеличивается на единицу. Работа цикла продолжается до тех пор, пока

значение counter не превысит 10. Выполнение цикла прерывается оператором break в строке 14, и на экран выводится значение переменной counter (строка 16).

Несмотря на то что данная программа работает, ее структуру нельзя назвать оптимальной. Это типичный пример некорректного использования оператора while. Правильным решением была бы организация проверки значения counter в условии продолжения цикла.

Гибкий синтаксис языка C++ позволяет решить одну и ту же задачу множеством различных способов. Поэтому важно научиться выбирать средство, наиболее подходящее в конкретной ситуации.

Организация циклов с помощью конструкции do...while

При организации циклов с помощью оператора while возможна ситуация, когда тело цикла вообще не будет выполняться. Поскольку условие продолжения цикла проверяется в начале каждой итерации, при нарушении истинности выражения, задающего это условие, выполнение цикла будет прервано еще до запуска первого оператора тела цикла. Пример такой ситуации приведен в листинге 7.6.

Листинг 7.6. Преждевременное завершение цикла с while

```
1:    // Листинг 7.6.
2:    // Если условие продолжения цикла не выполняется,
3:    // тело цикла пропускается.
4:
5:    #include <iostream.h>
6:
7:    int main()
8:    {
9:        int counter;
10:       cout << "How many hellos?: ";
11:       cin >> counter;
12:       while (counter > 0)
13:       {
14:          cout << "Hello!\ n";
15:          counter--;
16:       }
17:       cout << "Counter is OutPut: " << counter;
18:       return 0;
19:    }
```

РЕЗУЛЬТАТ

```
How many hellos?: 2
Hello!
Hello!
Counter is OutPut: 0

How many hellos?: 0
Counter is OutPut: 0
```

АНАЛИЗ В строке 10 вам предлагается ввести начальное значение счетчика, которое записывается в переменную counter. В строке 12 это значение проверяется, а затем в теле цикла уменьшается на единицу. При первом запуске программы начальное значение счетчика равнялось двум, поэтому тело цикла выполнялось дважды. Во втором случае было введено число 0. Понятно, что в этом случае условие продолжения цикла не выполнялось и тело цикла было пропущено. В результате приветствие не было выведено ни разу.

Как же поступить, чтобы сообщение выводилось по крайней мере один раз? С помощью оператора while это сделать невозможно, так как условие проверяется еще до выполнения тела цикла. Один из способов решения этой проблемы — использование оператора if для контроля начального значения переменной counter.

```
If (counter < 1)  // контроль начального значения
counter = 1;
```

Правда, это довольно «корявый» выход из ситуации.

Использование конструкции do...while

При использовании конструкции do...while условие проверяется после выполнения тела цикла. Это гарантирует выполнение операторов цикла по крайней мере один раз. В листинге 7.7 приведен измененный вариант предыдущей программы, в котором вместо оператора while используется конструкция do...while.

Листинг 7.7. Использование конструкции do...while

```
1:    // Листинг 7.7.
2:    // Пример использования конструкции do...while
3:
4:    include <iostream.h>
5:
6:    int main()
7:    {
8:        int counter;
9:        cout << "How many hellos? ";
10:       cin >> counter;
11:       do
12:       {
13:            cout << "Hello\ n";
14:            counter--;
15:       }    while (counter >0 );
16:       cout << "Counter is: " << counter << endl;
17:       return 0;
18:   }
```

РЕЗУЛЬТАТ

```
How many hellos? 2
Hello
Hello
Counter is: 0
```

АНАЛИЗ В строке 9 пользователю предлагается ввести начальное значение счетчика, которое записывается в переменную counter. В конструкции do...while условие проверяется в конце каждой итерации, что гарантирует выполнение тела цикла по меньшей мере один раз. В строке 13 на экран выводится текст приветствия, а в строке 14 значение переменной counter уменьшается на единицу. Условие продолжения цикла проверяется в строке 15. Если оно истинно, выполняется следующая итерация цикла (строка 13). В противном случае цикл завершается и управление передается в строку 16.

При использовании в конструкциях do...while операторы break и continue дают тот же результат, что и при использовании с оператором while. Единственное различие этих двух методов организации циклов состоит в проверке условия продолжения цикла. В первом случае оно контролируется перед выполнением тела цикла, а во втором — после него.

Оператор for

Для организации цикла с помощью оператора while необходимо выполнить три обязательных действия: установить начальные значения переменных цикла, а затем на каждой итерации проконтролировать выполнение условия продолжения цикла и изменить значение переменной цикла (листинг 7.8).

Листинг 7.8. Еще один пример использования оператора while

```
1:    // Листинг 7.8.
2:    // Еще один пример использования оператора while
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:       int counter = 0;
9:
10:      while(counter < 5)
11:      {
12:         counter++;
13:         cout << " Looping!  ";
14:      }
15:
16:   cout << "\ nCounter: " << counter << " \ n";
17:   return 0;
18:   }
```

РЕЗУЛЬТАТ

```
Looping!  Looping!  Looping!  Looping!  Looping!
Counter: 5.
```

АНАЛИЗ В строке 8 переменная цикла counter инициализируется нулевым значением. Затем в строке 10 проверяется условие продолжения цикла, а в строке 12 значение счетчика увеличивается на единицу. В строке 13 на экран выво-

дится сообщение, наглядно иллюстрирующее циклический процесс. Конечно, в цикле вашей программы могут выполняться и более серьезные операции.

Оператор for позволяет объединить три операции, необходимые для работы цикла, в одну. Кратко эти операции можно охарактеризовать так: инициализация, проверка условия и приращение счетчика цикла. Выражение с оператором for состоит из самого этого оператора, за которым в круглых скобках следуют три выражения, устанавливающих параметры выполнения цикла. Выражения в круглых скобках разделяются символами точки с запятой.

Первое выражение цикла for устанавливает начальное значение счетчика цикла. Счетчик, как правило, представляет собой целочисленную переменную, которая объявляется и инициализируется прямо в цикле for, хотя в C++ допускается использование в этом месте любого выражения, выводящего начальное значение счетчика каким-то косвенным путем. Второй параметр цикла for определяет условие продолжения цикла, которое также может быть представлено любым выражением. Это условие выполняет те же функции, что и в конструкции while. Третий параметр устанавливает значение приращения счетчика цикла (по умолчанию шаг приращения равен единице). В этой части также может использоваться любое корректное выражение или оператор C++. Нужно заметить, что, хотя параметры цикла for могут задаваться любыми корректными выражениями C++, для установки второго параметра обязательно должно использоваться выражение, возвращающее логическое значение. Пример использования цикла for приведен в листинге 7.9.

Листинг 7.9. Пример использования цикла `for`

```
1:    // Листинг 7.9.
2:    // Пример использования цикла for
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:    int counter;
9:    for (counter = 0; counter < 5; counter++)
10:       cout << " Looping! ";
11:
12:   cout << "\ nCounter: " << counter << ".\ n";
13:   return 0;
14:   }
```

РЕЗУЛЬТАТ

```
Looping!  Looping!  Looping!  Looping!  Looping!
Counter: 5.
```

АНАЛИЗ

В строке 9 программы с помощью оператора for задается цикл, отсчитывающий число итераций с помощью переменной counter. После каждого цикла проверяется условие продолжения цикла и значение переменной counter увеличивается на единицу. Тело цикла состоит из одного оператора, записанного в строке 10. В реальных программах тело цикла может состоять из любого количества операторов.

Сложные выражения с оператором for

При профессиональном использовании цикл for становится мощным и гибким инструментом программирования. Тот факт, что оператор for допускает установку трех независимых параметров цикла (инициализацию, условие продолжения и шаг), открывает неограниченные возможности в управлении работой цикла.

Параметры цикла for

Синтаксис установок параметров цикла for следующий:

```
for(инициализация, проверка, операция)
выражения;
```

Выражение инициализации используется для установки начального значения счетчика цикла или для выполнения какой-нибудь другой операции, подготавливающей работу цикла. Под проверкой понимают некое выражение на языке C++, которое выполняется перед каждой новой итерацией цикла и возвращает логическое значение. Если возвращается значение true, то программа выполняет строки тела цикла, после чего выполняется третье выражение в параметрах цикла, которое, как правило, приращивает значение счетчика на заданную величину.

Пример 1:

```
// напечатать Hello десять раз
for(int i=0; i<10; i++)
cout << "Hello!" < endl;
```

Пример 2:

```
for(int i = 0; i < 10; i++)
{
    cout << "Hello!" << endl;
    cout << "the value of i is: " << i << endl;
}
```

Цикл for работает в такой последовательности:

1. Присваивается начальное значение счетчику цикла.
2. Вычисляется значения выражения, устанавливающего условие продолжения цикла.
3. Если выражение условия возвращает true, то сначала выполняется тело цикла, а затем заданная операция над счетчиком цикла.

На каждой итерации шаги 2 и 3 повторяются.

Множественные инициализации и приращения счетчиков цикла

Синтаксис задания цикла for позволяет инициализировать несколько переменных-счетчиков, проверять сложные условия продолжения цикла или последовательно выполнять несколько операций над счетчиками цикла. Если присваиваются значения нескольким счетчикам или выполняется несколько операций, они записываются последовательно и разделяются запятыми. В листинге 7.10 инициализируются два счетчика, значения которых после каждой итерации увеличиваются на единицу.

Листинг 7.10. Использование нескольких счетчиков в цикле `for`

```
1:    //Листинг 7.10.
2:    // Использование нескольких счетчиков
3:    // в цикле for
4:
5:    #include <iostream.h>
6:
7:    int main()
8:    {
9:         for (int i=0, j=0; i<3; i++, j++)
10:           cout << "i: " << i << " j: " << j << endl;
11:        return 0;
12:    }
```

Результат

```
i: 0  j: 0
i: 1  j: 1
i: 2  j: 2
```

Анализ В строке 9 переменные `i` и `j` инициализируются нулевыми значениями. Затем проверяется условие `i<3` и, так как оно справедливо, выполняется первая итерация цикла. На каждой итерации осуществляется вывод значений счетчиков на экран. После этого выполняется третья часть конструкции `for`, в которой значения переменных-счетчиков увеличиваются на единицу. После выполнения строки 10 и изменения значений переменных условие проверяется снова. Если условие все еще справедливо, запускается следующая итерация цикла. Это происходит до тех пор, пока условие продолжения цикла не нарушится. В этом случае значения переменных не изменяются и управление передается следующему после цикла оператору.

Нулевые параметры цикла for

Любой параметр цикла `for` может быть опущен. Пропуск означает использование так называемого нулевого параметра. Нулевой параметр, как и любой другой, отделяется от остальных параметров цикла `for` символом точки с запятой (;). Если опустить первый и третий параметры цикла `for`, как показано в листинге 7.11, результат его применения будет аналогичен полученному при использовании оператора `while`.

Листинг 7.11. Нулевые параметры цикла `for`

```
1:    // Листинг 7.11.
2:    // Нулевые параметры цикла for
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:       int counter = 0;
9:
10:      for( ; counter < 5; )
```

```
11:       {
12:          counter++;
13:          cout << "Looping! ";
14:       }
15:
16:       cout << "\ nCounter: " << counter << ".\ n";
17:       return 0;
18:    }
```

РЕЗУЛЬТАТ

```
Looping!  Looping!  Looping!  Looping!  Looping!
Counter: 5.
```

АНАЛИЗ Очевидно, что результат выполнения такого цикла в точности совпадает с результатом выполнения цикла `while` из листинга 7.8. В строке 8 присваивается значение переменной `counter`. Установки параметров цикла `for`, показанные в строке 10, содержат только проверку условия продолжения цикла. Операция над переменной цикла в конструкции for также опущена. Таким образом, этот цикл можно представить в виде

```
while (counter < 5).
```

Рассмотренный пример еще раз показывает, что возможности языка C++ позволяют решить одну и ту же задачу множеством способов. Листинг 7.11 приведен скорее для иллюстрации гибкости возможностей C++, поскольку ни один опытный программист не будет использовать цикл `for` подобным образом. Тем не менее можно опустить даже все три параметра цикла `for`, а для управления циклом использовать операторы `break` и `continue`. Пример использования конструкции `for` без параметров приведен в листинге 7.12.

Листинг 7.12. Использование оператора `for` без параметров

```
1:    //Листинг 7.12.
2:    // Использование оператора for без параметров
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:       int counter=0;              // установка начального значения счетчика
9:       int max;
10:      cout << " How many hellos?";
11:      cin >> max;
12:      for (;;)                    // задание бесконечного цикла
13:      {
14:         if (counter < max)       // проверка значения
15:         {
16:              cout << "Hello!\ n";
17:              counter++;          // приращение значения счетчика
```

```
18:          }
19:          else
20:               break;
21:       }
22:    return 0;
23:    }
```

РЕЗУЛЬТАТ

```
How many hellos?3
Hello!
Hello!
Hello!
```

АНАЛИЗ В этом примере набор параметров оператора `for` максимально минимизирован. Опущены все три параметра — инициализация, условие и операция. Начальное значение счетчика присваивается в строке 8 еще до начала работы цикла. Условие продолжения цикла также проверяется в отдельной строке (строка 14), и, если оно истинно, выполняется операция тела цикла, после чего в строке 17 увеличивается значение счетчика. Если условие не выполняется, оператор `break` в строке 20 прерывает выполнение цикла.

Несмотря на то что рассмотренная программа выглядит достаточно нелепо, встречаются ситуации, когда конструкции `for( ;; )` и `while( true )` оказываются просто необходимыми. Более полезный пример использования таких конструкций будет приведен далее в этой главе после рассмотрения оператора `switch`.

Использование пустых циклов for

Поскольку синтаксис оператора `for` позволяет использовать при его описании цикла достаточно сложные конструкции, необходимость в теле цикла иногда вообще отпадает. Это означает, что тело цикла будет состоять из пустой строки, заканчивающейся символом точки с запятой (;). Данный символ можно размещать в одной строке с оператором `for`. Пример пустого цикла приведен в листинге 7.13.

Листинг 7.13. Использование оператора `for` для организации пустого цикла

```
1:    // Листинг 7.13.
2:    // Использование оператора for
3:    // для организации "пустого" цикла
4:
5:    #include <iostream.h>
6:    int main()
7:    {
8:         for (int i = 0; i<5; cout << "i: " << i++ << endl)
9:         ;
10:   return 0;
11:   }
```

РЕЗУЛЬТАТ

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

АНАЛИЗ Оператор for в строке 8 содержит все три параметра. Инициализация в данном случае состоит из описания переменной i и присвоения ей значения 0. Затем проверяется условие i<5, и, если оно выполняется, в третьей части оператора for значение переменной выводится на экран и увеличивается на единицу.

Поскольку все необходимые операции выполняются в самом операторе for, тело цикла можно оставить пустым. Такой вариант нельзя назвать оптимальным, так как запись в одной строке большого количества операций значительно усложняет восприятие программы. Правильнее было бы записать этот цикл таким образом:

```
8:     for (int i = 0; i<5; i++)
9:          cout << "i: " << i << endl;
```

Оба варианта записи равноценны, однако второй вариант гораздо читабельнее и понятнее.

Вложенные циклы

Цикл, организованный в теле другого цикла, называют вложенным. В этом случае внутренний цикл полностью выполняется на каждой итерации внешнего цикла. Листинг 7.14 демонстрирует заполнение элементов матрицы с помощью вложенного цикла.

Листинг 7.14. Вложенные циклы

```
1:     //Листинг 7.14.
2:     //Вложенные циклы с оператором for
3:
4:     #include <iostream.h>
5:
6:     int main()
7:     {
8:              int rows, columns;
9:              char theChar;
10:             cout << "How many rows? ";
11:             cin >> rows;
12:             cout << "How many columns? ";
13:             cin >> columns;
14:             cout << "What character? ";
15:             cin >> theChar;
16:             for (int i = 0; i<rows; i++)
17:             {
18:                  for (int j = 0; j<columns; j++)
19:                       cout << theChar;
20:                  cout << "\ n";
21:    }
22:    return 0;
23:    }
```

РЕЗУЛЬТАТ

```
How many rows? 4
How many columns? 12
What character? x
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
```

АНАЛИЗ

В начале программы пользователю предлагается ввести количество строк и столбцов матрицы, а также символ, которым будет заполняться матрица. В строке 16 задается начальное значение переменной `i`, после чего начинается выполнение тела внешнего цикла.

В первой строке тела внешнего цикла (строка 18) инициализируется еще один цикл. Переменной `j` присваивается значение `0` и начинается выполнение тела внутреннего цикла. В строке 19 символ, введенный при начале работы программы, выводится на экран. На этом первая итерация внутреннего цикла заканчивается. Вывод одной строки матрицы продолжается до тех пор, пока выполняется условие внутреннего цикла (`j<columns`). Как только значение переменной `j` становится равным значению `columns`, выполнение внутреннего цикла прекращается.

После вывода на экран строки матрицы (12 символов "x") управление передается в строку 20 и выводится символ новой строки. После этого проверяется условие внешнего цикла (`i<rows`) и, если оно справедливо, выполняется следующая итерация.

Обратите внимание: во второй итерации внешнего цикла внутренний цикл начинает выполняться с начала. Переменной `j` присваивается нулевое значение, что позволяет повторно выполнить все итерации внутреннего цикла.

Основная идея вложенных циклов состоит в том, что на каждой итерации внешнего цикла внутренний цикл выполняется полностью. Таким образом, результат выполнения данной программы будет следующим: заданный символ выводится для каждой строки столько раз, сколько указано в переменной `columns`, а количество выводимых строк определяется переменной `rows`.

Область видимости переменных-счетчиков циклов for

До недавнего времени область видимости переменных, описанных в цикле `for`, распространялась на весь текущий блок. Согласно новому стандарту, установленному ANSI, область видимости переменных, описанных в таком цикле, должна распространяться только на тело цикла. Следует заметить, что, несмотря на внесенные изменения, многие компиляторы продолжают поддерживать только старый стандарт. Набрав приведенный ниже фрагмент программного кода, можно проверить свой компилятор на соответствие новому стандарту.

```
#include <iostream.h>
int main()
{
    // Проверка области видимости переменной i
    for (int i = 0; i<5; i++)
        {
            cout << "i: " << i << endl;
        }
```

```
    i = 7;      // i находится за пределами области видимости
    return 0;
}
```

Если такая программа будет компилироваться без ошибок, значит, ваш компилятор еще не поддерживает нового стандарта ANSI.

Компиляторы, соответствующие новому стандарту, должны сгенерировать сообщение об ошибке для выражения `i = 7`. После внесения некоторых изменений программа будет восприниматься всеми компиляторами без ошибок.

```
#include <iostream.h>
int main()
{
    int i; //объявление переменной за пределами цикла
    for (int i = 0; i<5; i++)
    {
        cout << "i: " << i << endl;
    }
    i = 7;      // теперь переменная i будет корректно восприниматься всеми компиляторами
    return 0;
}
```

Обобщение сведений о циклах

На занятии 5 рассматривался пример построения ряда чисел Фибоначчи с использованием рекурсивного алгоритма. Напомним, что этот ряд начинается числами 1, 1, 2, 3, а все последующие его члены являются суммой двух предыдущих.

1, 1, 2, 3, 5, 8, 13, 21, 34...

Таким образом, *n*-й член ряда Фибоначчи вычисляется сложением *(n-1)*-го и *(n-2)*-го членов. Рассмотрим вариант решения этой задачи с помощью циклов (листинг 7.15).

Листинг 7.15. Нахождение *n*-го члена ряда Фибоначчи с помощью цикла

```
1:    // Листинг 7.15.
2:    // Нахождение n-го члена ряда Фибоначчи
3:    // с помощью цикла
4:
5:    #include <iostream.h>
6:
8:
9:    int fib(int position);
10:      int main()
11:      {
12:           int answer, position;
13:           cout << "Which position? ";
14:           cin >> position;
15:           cout << "\ n";
16:
17:           answer = fib(position);
```

```
18:             cout << answer << " is the ";
19:             cout << position << "Fibonacci number.\ n";
20:             return 0;
21:       }
22:
23:  int fib(int n)
24:  {
25:             int minusTwo=1, minusOne=1, answer=2;
26:
27:             if (n < 3)
28:                  return 1;
29:
30:             for (n -= 3; n; n--)
31:             {
32:                  minusTwo = minusOne;
33:                  minusOne = answer;
34:                  answer = minusOne + minusTwo;
35:             }
36:
37:             return answer;
38:  }
```

РЕЗУЛЬТАТ

```
Which position? 4
3 is the 4th Fibonacci number.
Which position? 5
5 is the 5th Fibonacci number.
Which position? 20
6765 is the 20th Fibonacci number.
Which position? 100
3314859971 is the 100th
Fibonacci number.
```

АНАЛИЗ Программа, представленная в листинге 7.15, позволяет найти значение любого члена ряда Фибоначчи. Использование рекурсии заменено циклом, организованным с помощью конструкции `for`. Кроме того, применение цикла уменьшает объем используемой памяти и время выполнения программы.

В строке 13 пользователю предлагается ввести порядковый номер искомого члена ряда Фибоначчи. Для нахождения этого значения используется функция `fib()`, в качестве параметра которой передается введенный порядковый номер. Если он меньше трех, функция возвращает значение `1`. Для вычисления значений, порядковый номер которых превышает `2`, используется приведенный ниже алгоритм.

1. Присваиваются начальные значения переменным: `minusTwo=1, minusOne=1, answer=2`. Значение переменной, содержащей номер искомой позиции, уменьшается на 3, поскольку две первые позиции обрабатываются выше.
2. Для каждого значения `n` вычисляем значение очередного члена последовательности. Делается это следующим образом:
   - переменной `minusTwo` присваивается значение переменной `minusOne`;
   - переменной `minusOne` присваивается значение переменной `answer`;

- значения переменных `minusOne` и `minusTwo` суммируются и записываются в `answer`;
- значение `n` уменьшается на единицу.

3. Как только `n` достигнет нуля, возвращается значение переменной `answer`.

Следуя описанному алгоритму, можно воспроизвести на листе бумаги ход выполнения программы. Для нахождения, к примеру, пяти первых членов последовательности на первом шаге записываем

1, 1, 2,

Остается определить еще два члена ряда. Следующий член будет равен (2+1=3), а для вычисления искомого члена теперь нужно сложить значения только что полученного члена и предыдущего — числа 2 и 3, в результате чего получаем 5. В сущности, на каждом шаге мы смещаемся на один член вправо и уменьшаем количество искомых значений.

Особое внимание следует уделить выражению условия продолжения цикла `for`, записанному как `n`. Это одна из особенностей синтаксиса языка C++. По-другому это выражение можно представить в виде `n!=0`. Поскольку в C++ число `0` соответствует значению `false`, при достижении переменной `n` нуля условие продолжения цикла не будет выполняться. Исходя из сказанного, описание цикла может быть переписано в виде

```
for (n-=3; n!=0; n--)
```

Подобная запись значительно облегчит его восприятие. С другой стороны, первоначальный вариант программы иллюстрирует общепринятую для C++ форму записи условия, поэтому не стоит умышленно ее избегать.

Скомпилируйте и запустите полученную программу. Сравните время, затрачиваемое на вычисление 25-го числа рекурсивным (см. занятие 5) и циклическим методами. Несомненно, рекурсивный вариант программы более компактный, однако многократный вызов функции, использующийся в любом рекурсивном алгоритме, заметно снижает его быстродействие. Поэтому использование цикла более приемлемо с точки зрения скорости выполнения. Кроме того, благодаря оптимизации арифметических операций в большинстве современных микропроцессоров превосходство нерекурсивных алгоритмов в скорости становится все более очевидным.

Испытывая программу, не вводите слишком большие номера членов ряда Фибоначчи. Значения членов ряда возрастают довольно быстро и ввод большого порядкового номера может привести к переполнению регистра памяти.

Оператор switch

На занятии 4 вы познакомились с операторами `if` и `if/else`. Однако в некоторых ситуациях применение оператора `if` может привести к возникновению конструкций с большим числом вложений, значительно усложняющих как написание, так и восприятие программы. Для решения этой проблемы в языке C++ предусмотрен оператор `switch`. Основным его отличием от оператора `if` является то, что он позволяет проверять сразу несколько условий, в результате чего ветвление программы организуется более эффективно. Синтаксис оператора `switch` следующий:

```
switch (выражение)
{
case ПервоеЗначение: оператор;
                     break;
```

```
case ВтороеЗначение:  оператор;
                      break;
....
case Значение_N:      оператор;
                      break;
default:              оператор;
}
```

В скобках за оператором switch может использоваться любое выражение, корректное с точки зрения синтаксиса языка. Вместо идентификатора *оператор* допускается использование любого оператора или выражения, а также последовательности операторов или выражений, результатом выполнения которых является целочисленное значение (или значение, которое может быть однозначно приведено к целочисленному типу). Поэтому использование логических операций или выражений сравнения здесь не допускается.

Оператор switch

Синтаксис использования оператора switch следующий:

```
switch (выражение)
{
case ПервоеЗначение: выражение;
case ВтороеЗначение: выражение;
....
case Значение_N: выражение;
default: выражение;
}
```

Оператор switch позволяет осуществлять ветвление программы по результатам выражения, возвращающего несколько возможных значений. Значение, возвращенное выражением, заданным в скобках оператора switch, сравнивается со значениями, указанными за операторами case, и в случае совпадения значений выполняется выражение в строке соответствующего оператора case. Будут выполняться все строки программы после выбранного оператора case до тех пор, пока не закончится тело блока оператора switch или не повстречается оператор break.

Если ни одно из значений операторов case не совпадет с возвращенным значением, то выполняются строки программы, стоящие после оператора default, в случае же отсутствия этого оператора в теле блока switch управление будет передано следующей за этим блоком строке программы.

Пример 1:

```
switch (choice)
{
case 0:
            cout << "Zero!" << endl;
            break;
case 1:
            cout << "One!" << endl;
            break;
case 2:
            cout << "Two!" << endl;
            break;
default:
            cout << "Default!" << endl;
}
```

Пример 2:

```
switch (choice)
{
case 0:
case 1:
case 2:
            cout << "Less than 3!" << endl;
            break;
case 3:
            cout << "Equals 3!" << endl;
            break;
default:
cout << "Greater than 3!" << endl;
}
```

При отсутствии оператора `break` после оператора или выражения, следующего за `case`, будет выполняться выражение очередного блока `case`. В большинстве случаев такая ситуация возникает, когда оператор `break` пропущен по ошибке. Поэтому, если `break` опускается умышленно, рекомендуем вставлять в соответствующую строку комментарий.

Пример использования оператора `switch` приведен в листинге 7.16.

Листинг 7.16. Использование оператора `switch`

```
1:    //Листинг 7.16.
2:    // Использование оператора switch
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:         unsigned short int number;
9:         cout << "Enter a number between 1 and 5: ";
10:        cin >> number;
11:        switch (number)
12:        {
13:             case 0:  cout << "Too small, sorry!";
14:                      break;
15:             case 5:  cout << "Good job!\ n";  // fall through
16:             case 4:  cout << "Nice Pick!\ n"; // fall through
17:             case 3:  cout << "Excellent!\ n"; // fall through
18:             case 2:  cout << "Masterful!\ n"; // fall through
19:             case 1:  cout << "Incredible!\ n";
20:                      break;
21:             default: cout << "Too large!\ n";
22:                      break;
23:        }
24:    cout << "\ n\ n";
25:    return 0;
26:    }
```

РЕЗУЛЬТАТ

```
Enter a number between 1 and 5: 3
Excellent!
Masterful!
Incredible!
Enter a number between 1 and 5: 8
Too large!
```

АНАЛИЗ

Сначала программа предлагает ввести число. Затем введенное число обрабатывается оператором switch. Если вводится 0, то это соответствует значению оператора case из строки 13 и на экран выводится сообщение Too small, sorry!, после чего оператор break завершает выполнение конструкции switch. Если вводится число 5, управление передается в строку 15 и выводится соответствующее сообщение. Затем выполняется строка 16, в которой также выводится сообщение, и так до строки 20. В этой строке оператор break завершает выполнение блока с оператором switch.

Таким образом, при вводе чисел от 1 до 5 на экран будет выводиться ряд сообщений. Если же вводится число, превышающее 5, выполняется строка 21 с оператором default, в которой выводится сообщение Too large!.

Обработка команд меню с помощью оператора switch

Вернемся к теме циклов с оператором for(;;). Такие конструкции называют бесконечными циклами, поскольку, если выполнение такого цикла не прервать оператором break, он будет работать бесконечно. Циклы подобного типа удобно использовать для обработки команд меню (листинг 7.17). Пользователь выбирает одну из предложенных команд, затем выполняется определенное действие и осуществляется возврат в меню. Так продолжается до тех пор, пока пользователь не выберет команду выхода.

В бесконечных циклах не существует условия, при нарушении которого цикл прерывается. Поэтому выйти из такого цикла можно только посредством оператора break.

Листинг 7.17. Пример бесконечного цикла

```
1:    //Листинг 7.17.
2:    //Обработка диалога с пользователем
3:    //посредством бесконечного цикла
4:    #include <iostream.h>
5:
6:    // прототипы функций
7:    int menu();
8:    void DoTaskOne();
9:    void DoTaskMany(int);
10:
11:   int main()
12:   {
13:
14:   bool exit = false;
15:   for (;;)
16:   {
17:        int choice = menu();
18:        switch(choice)
19:        {
20:             case (1):
21:                  DoTaskOne();
```

```
22:                    break;
23:               case (2):
24:                    DoTaskMany(2);
25:                    break;
26:               case (3):
27:                    DoTaskMany(3);
28:                    break;
29:               case (4):
30:                    continue;          // необязательная строка
31:                    break;
32:               case (5):
33:                    exit=true;
34:                    break;
35:               default:
36:                    cout << " Please select again!\ n";
37:                    break;
38:          }       // конец блока switch
39:
40:          if (exit)
41:               break;
42:     }            // и так до бесконечности
43:     return 0;
44:     }// конец функции main()
45:
46:     int menu()
47:     {
48:          int choice;
49:
50:        cout << " **** Menu ****\ n\ n";
51:        cout << "(1) Choice one.\ n";
52:        cout << "(2) Choice two.\ n";
53:        cout << "(3) Choice three.\ n";
54:        cout << "(4) Redisplay menu.\ n";
55:        cout << "(5) Quit.\ n\ n";
56:          cout << ": ";
57:          cin >> choice;
58:          return choice;
59:     }
60:
61:     void DoTaskOne()
62:     {
63:        cout << "Task One!\ n";
64:     }
65:
66:     void DoTaskMany(int which)
67:     {
68:        if (which == 2)
69:             cout << "Task Two!\ n";
70:        else
71:             cout << "Task Three!\ n";
72:     }
```

РЕЗУЛЬТАТ

```
**** Menu ****
(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.

: 1
Task One!
 **** Menu ****
(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.

: 3
Task Three!
 **** Menu ****
(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.

: 5
```

АНАЛИЗ

В данной программе используются многие средства программирования, рассмотренные на этом и предыдущих занятиях. Тут же вы найдете пример использования конструкции `switch`.

Работа бесконечного цикла начинается в строке 15. Функция `menu()` обеспечивает вывод на экран команд меню и возвращает номер выбранной пользователем команды. Обработка введенного номера осуществляется в конструкции `switch` в строках 18–38.

При выборе первой команды управление передается следующему после строки `case (1):` оператору (строка 21). Далее, в строке 21, вызывается функция `DoTaskOne()`, которая выводит на экран сообщение о выборе пользователя. После завершения работы функции осуществляется возврат в точку вызова и выполняется оператор `break` (строка 22). Оператор `break` прерывает работу блока `switch` и управление передается в строку 39. Далее, в строке 40, проверяется значение переменной `exit`. Если оно истинно, бесконечный цикл прерывается оператором `break` в строке 41. В противном случае выполняется следующая итерация цикла (строка 15).

Особое внимание следует уделить оператору `continue` в строке 30. Внимательно проанализировав структуру программы, вы заметите, что этот оператор можно опустить, причем работа программы не изменится. Если бы строки с этим оператором не было, выполнялся бы оператор `break`, затем оператор `if` и, так как переменная `exit` содержала бы значение `false`, запускалась следующая итерация цикла. Использование оператора `continue` просто позволяет перейти на новую итерацию без проверки значения `exit`.

Резюме

В языке C++ существует множество способов организации циклических процессов. Оператор `while` проверяет условие и, если оно истинно, передает управление телу цикла. В конструкции `do...while` условие проверяется уже после выполнения тела цикла. Оператор `for` позволяет инициализировать переменные цикла, после чего проверяется выполнение условия. Если оно истинно, выполняется тело цикла, а затем операция, являющаяся третьей частью заголовка конструкции `for`. Перед началом каждой следующей итерации условие проверяется заново.

Оператора `goto` следует по возможности избегать, поскольку он позволяет осуществить переход в любую точку программы, что значительно усложняет ее восприятие и анализ. С помощью оператора `continue` можно осуществить переход на следующую итерацию цикла `while`, `do...while` или `for`, а `break` позволяет мгновенно завершить работу цикла.

Вопросы и ответы

Как определить, какой из операторов, `if/else` или `switch`, лучше использовать в конкретной ситуации?

Если приходится использовать более двух вложений операторов `if`, то лучше воспользоваться конструкцией с оператором `switch`.

Как выбрать между операторами `while` и `do...while`?

Если тело цикла должно выполняться хотя бы один раз, используйте цикл `do...while`. Во всех остальных случаях используйте оператор `while`.

Как выбрать между операторами `while` и `for`?

В тех случаях, когда переменная счетчика еще не инициализирована и ее значение изменяется после каждой итерации цикла на постоянную величину, используйте оператор `for`. В остальных случаях предпочтительнее `while`.

В каких случаях лучше использовать рекурсию, а в каких итерацию?

Несомненно, в большинстве случаев итеративный метод предпочтительнее, однако, если один и тот же цикл приходится повторять в разных частях программы, удобнее использовать рекурсию.

Какой из операторов, `for(;;)` или `while(true)` работает эффективнее?

Существенного различия между ними нет.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Можно ли в цикле `for` инициализировать сразу несколько переменных-счетчиков?
2. Почему следует избегать использование оператора `goto`?
3. Можно ли с помощью оператора `for` организовать цикл, тело которого не будет выполняться?
4. Можно ли организовать цикл `while` внутри цикла `for`?
5. Можно ли организовать цикл, который никогда не завершится? Приведите пример.
6. Что происходит при запуске бесконечного цикла?

Упражнения

1. Каким будет значение переменной `x` после завершения цикла `for (int x = 0; x < 100; x++)`?
2. Создайте вложенный цикл `for`, заполняющий нулями массив размером 10×10.
3. Организуйте цикл `for`, счетчик которого изменяется от 100 до 200 с шагом 2.
4. Организуйте цикл `while`, счетчик которого изменяется от 100 до 200 с шагом 2.
5. Организуйте цикл `do...while`, счетчик которого изменяется от 100 до 200 с шагом 2.
6. **Жучки:** найдите ошибку в приведенном фрагменте программы.

```
int counter = 0;
while (counter < 10)
{
     cout << "counter: " << counter;
}
```

7. **Жучки:** найдите ошибку в приведенном фрагменте программы.

```
for (int counter = 0; counter < 10; counter++);
    cout << counter << " ";
```

8. **Жучки:** найдите ошибку в приведенном фрагменте программы.

```
int counter = 100;
while (counter < 10)
{
     cout << "counter: " << counter;
     counter--;
}
```

9. **Жучки:** найдите ошибку в приведенном фрагменте программы.

```
cout << "Enter a number between 0 and 5: ";
cin >> theNumber;
switch (theNumber)
```

```
{
    case 0:
        doZero();
    case 1:          // идем дальше
    case 2:          // идем дальше
    case 3:          // идем дальше
    case 4:          // идем дальше
    case 5:
        doOneToFive();
        break;
    default:
        doDefault();
        break;
}
```

Подведение итогов

Итоги первой недели

```
1:   #include <iostream.h>
2:   intintboolfalsetrue
3:   enum CHOICE {  DrawRect = 1, GetArea,
4:     GetPerim, ChangeDimensions, Quit} ;
5:   // Объявление класса Rectangle
6:   class Rectangle
7:   {
8:    public:
9:        // constructors
10:       Rectangle(int width, int height);
11:       ~Rectangle();
12:
13:       // Методы доступа
14:       int GetHeight() const {  return itsHeight; }
15:       int GetWidth() const {  return itsWidth; }
16:       int GetArea() const {  return itsHeight * itsWidth; }
17:       int GetPerim() const {  return 2*itsHeight + 2*itsWidth; }
18:       void SetSize(int newWidth, int newHeight);
19:
20:       // Прочие методы
21:
22:
23:   private:
24:       int itsWidth;
25:       int itsHeight;
26: } ;
27:
28: // Выполнение методов класса
29: void Rectangle::SetSize(int newWidth, int newHeight)
30: {
31:    itsWidth = newWidth;
32:    itsHeight = newHeight;
33: }
```

```
34:
35:
36: Rectangle::Rectangle(int width, int height)
37: {
38:    itsWidth = width;
39:    itsHeight = height;
40: }
41:
42: Rectangle::~Rectangle() { }
43:
44: int DoMenu();
45: void DoDrawRect(Rectangle);
46: void DoGetArea(Rectangle);
47: void DoGetPerim(Rectangle);
48:
49: int main ()
50: {
51:    // Инициализация объекта rectangle значением 30,5
52:    Rectangle theRect(30,5);
53:
54:    int choice = DrawRect;
55:    int fQuit = false;
56:
57:    while (!fQuit)
58:    {
59:       choice = DoMenu();
60:       if (choice < DrawRect || choice >  Quit)
61:       {
62:            cout << "\ nInvalid Choice, please try again.\ n\ n";
63:            continue;
64:       }
65:       switch (choice)
66:       {
67:       case  DrawRect:
68:            DoDrawRect(theRect);
69:            break;
70:       case GetArea:
71:            DoGetArea(theRect);
72:            break;
73:       case GetPerim:
74:            DoGetPerim(theRect);
75:            break;
76:       case ChangeDimensions:
77:            int newLength, newWidth;
78:            cout << "\ nNew width: ";
79:            cin >> newWidth;
80:            cout << "New height: ";
81:            cin >> newLength;
82:            theRect.SetSize(newWidth, newLength);
83:            DoDrawRect(theRect);
```

```
84:              break;
85:         case Quit:
86:              fQuit = true;
87:              cout << "\ nExiting...\ n\ n";
88:              break;
89:         default:
90:              cout << "Error in choice!\ n";
91:              fQuit = true;
92:              break;
93:         }    // end switch
94:    }      // end while
95:  return 0;
96: }        // end main
97:
98: int DoMenu()
99: {
100:   int choice;
101:    cout << "\ n\ n   *** Menu *** \ n";
102:    cout << "(1) Draw Rectangle\ n";
103:    cout << "(2) Area\ n";
104:    cout << "(3) Perimeter\ n";
105:    cout << "(4) Resize\ n";
106:    cout << "(5) Quit\ n";
107:
108:    cin >> choice;
109:    return choice;
110: }
111:
112: void DoDrawRect(Rectangle theRect)
113: {
114:   int height = theRect.GetHeight();
115:   int width = theRect.GetWidth();
116:
117:   for (int i = 0; i<height; i++)
118:   {
119:     for (int j = 0; j< width; j++)
120:       cout << "*";
121:     cout << "\ n";
122:   }
123: }
124:
125:
126: void DoGetArea(Rectangle theRect)
127: {
128:   cout << "Area: " <<  theRect.GetArea() << endl;
129: }
130:
131: void DoGetPerim(Rectangle theRect)
132: {
```

```
133:    cout << "Perimeter: " <<  theRect.GetPerim() << endl;
134: }
     <Output>
*** Menu ***
(1) Draw Rectangle
(2) Area
(3) Perimeter
(4) Resize
(5) Quit
1
******************************
******************************
******************************
******************************
******************************

    *** Menu ***
(1) Draw Rectangle
(2) Area
(3) Perimeter
(4) Resize
(5) Quit
2
Area: 150

    *** Menu ***
(1) Draw Rectangle
(2) Area
(3) Perimeter
(4) Resize
(5) Quit
3
Perimeter: 70

    *** Menu ***
(1) Draw Rectangle
(2) Area
(3) Perimeter
(4) Resize
(5) Quit
4

New Width: 10
New height: 8
**********
**********
**********
**********
**********
**********
```

```
**********
**********

    *** Menu ***
(1) Draw Rectangle
(2) Area
(3) Perimeter
(4) Resize
(5) Quit
2
Area: 80

    *** Menu ***
(1) Draw Rectangle
(2) Area
(3) Perimeter
(4) Resize
(5) Quit
3
Perimeter: 36

    *** Menu ***
(1) Draw Rectangle
(2) Area
(3) Perimeter
(4) Resize
(5) Quit
5

Exiting...
```

В данной программе сведено большинство тех средств и подходов программирования, с которыми вы познакомились в течение первой недели. Вы должны не только уметь ввести программный код, скомпилировать, скомпоновать и запустить эту программу, но также и понимать, что и как в ней работает. Если все это вам удалось, значит, неделя прошла не зря.

В первых шести строках делаются объявления новых типов данных и важные определения, которые затем будут использоваться на протяжении всей программы.

В строках 6–26 объявляется класс `Rectangle`. Он содержит открытые методы доступа для возвращения и установки ширины и высоты прямоугольника, а также для вычисления его площади и периметра. Строки 29–40 содержат определения тех функций-членов класса, которые не объявлялись с ключевым словом `inline`.

Прототипы обычных функций, не являющихся членами класса, находятся в строках 44–47, а основной блок программы начинается со строки 49. Суть программы состоит в построении виртуального прямоугольника с выводом меню, предлагающего выбор из пяти опций: вывод прямоугольника на экран, определение его площади, определение периметра, изменение размера прямоугольника и выход из программы.

Флаг устанавливается в строке 55, и если пользователь установит неверное значение, то вывод меню на экран повторится. Это будет продолжатся до тех пор, пока пользователь правильно не укажет один из режимов работы либо не выберет завершение программы.

В случае выбора одного из режимов работы, за исключением ChangeDimensions, будет вызываться соответствующая функция, выбираемая с помощью оператора switch. Выбор константы ChangeDimensions не вызывает никакой функции, поскольку в этом случае пользователь должен ввести новые значения размера прямоугольника. Если предположить, что для изменения размеров прямоугольника в программе существовала бы специальная функция DoChangeDimensions(), в которую объект Rectangle передавался бы как значение, то все изменения в функции производились бы над копией существующего объекта, а сам объект в функции main() оставался бы неизменным. На занятии 8, посвященном указателям, и на занятии 10, где речь идет о разработке более сложных функций, вы узнаете, как обойти это ограничение, передавая объекты в функции как ссылки. Но пока все изменения значения объекта можно осуществлять только в функции main().

Обратите внимание, что использование перечисления сделало конструкцию оператора switch более понятной. Если бы вместо констант, о назначении которых можно судить по их именам, проверялись бы вводимые пользователем числовые значения от 1 до 5, нам бы пришлось каждый раз возвращаться к описанию меню, чтобы не запутаться в том, какой номер соответствует той или иной опции.

В строке 60 осуществляется проверка, входит ли значение, введенное пользователем, в диапазон допустимых значений. Если это не так, будет показано сообщение об ошибке и вывод меню на экран повторится. Тем не менее обратите внимание, что конструкция оператора switch содержит оператор default, хотя в этой программе он никогда не будет выполняться. Этот оператор добавлен исключительно для облегчения отладки программы, а также на случай будущих изменений в программе.

Итоги первой недели

Поздравляем вас! Вы завершили первую неделю обучения программированию на C++! Теперь вы вполне готовы не только к пониманию, но и к созданию довольно сложных программ. Конечно, еще многое нужно узнать, и следующая неделя начнется с довольно сложной и запутанной темы — использование указателей. Не расслабляйтесь, вам предстоит еще более углубиться в пучину объектно-ориентированного программирования, виртуальных функций и многих других современных и мощных средств языка программирования C++.

Немного передохните, наградите себя шоколадной медалью за проделанный путь и, перелистнув страницу, приступайте к следующей неделе.

Основные вопросы

Мы завершили первую неделю обучения и научились основным принципам и средствам программирования на C++. Для вас теперь не должно составлять труда написание и компиляция небольшой программы. Также вы должны четко представлять, что такое классы и объекты, составляющие основу объект-ориентированного программирования.

Что дальше

Вторую неделю начнем с изучения указателей. Указатели традиционно являются сложной темой для освоения начинающими программистами на C++. Но в этой книге вы найдете подробные и наглядные разъяснения того, что такое указатель и как он работает, поэтому, мы надеемся, что через день вы уже свободно будете владеть этим средством программирования. На занятии 9 вы познакомитесь со ссылками, которые являются близкими родственниками указателей. На занятии 10 вы узнаете как замещать функции, а занятие 11 будет посвящено наследованию и разъяснению фундаментальных принципов объект-ориентированного программирования. На занятии 12 вы узнаете как создавать структуры данных от простых массивов до связанных списков. Занятие 13 расширит ваши представления об объект-ориентированном программировании и познакомит с полиморфизмом, а занятие 14 завершит вторую неделю обучения рассмотрением статических функций и функций друзей класса.

День 8-й

Указатели

Возможность непосредственного доступа к памяти с помощью указателей — одно их наиболее мощных средств программирования на C++. Сегодня вы узнаете:

- Что такое указатели
- Как объявляются и используются указатели
- Как работать с памятью

При работе с указателями программисты подчас сталкиваются с довольно специфическими проблемами, поскольку в некоторых ситуациях механизм работы указателей может оказаться весьма запутанным. Кроме того, в ряде случаев нельзя однозначно ответить на вопрос о необходимости применения указателей. На этом занятии последовательно, шаг за шагом, вы освоите основные принципы работы с указателями. Однако осознать всю мощь этих средств вы сможете, только прочитав книгу до конца.

Что такое указатель

Указатель — это переменная, в которой записан адрес ячейки памяти компьютера.

Чтобы понять, как работают указатели, необходимо хотя бы в общих чертах, ознакомиться с базовыми принципами организации машинной памяти. Машинная память состоит из последовательности пронумерованных ячеек. Значение каждой переменной хранится в отдельной ячейке памяти, которая называется ее адресом. На рис. 8.1 изображена структура размещения в памяти четырехбайтового целого значения переменной `theAge`.

Для разных компьютеров характерны различные правила адресации памяти, имеющие свои особенности. Однако в большинстве случаев программисту не обязательно знать точный адрес какой-либо переменной — эту задачу выполняет компьютер. При необходимости такую информацию можно получить с помощью оператора адреса (`&`). Пример использования этого оператора приведен в листинге 8.1.

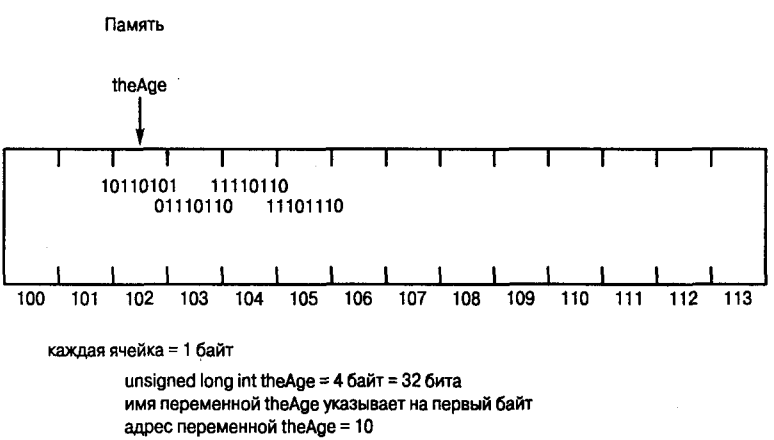


Рис. 8.1. Сохранение в памяти переменной `theAge`

Листинг 8.1. Оператор адреса

```
1:    // Листинг 8.1. Пример использования
2:    // оператора адреса
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:        unsigned short shortVar=5;
9:        unsigned long  longVar=65535;
10:       long sVar = -65535;
11:
12:       cout << "shortVar:\ t" << shortVar;
13:       cout << " Address of shortVar:\ t";
14:       cout <<  &shortVar  << "\ n";
15:
16:       cout << "longVar:\ t"  << longVar;
17:       cout  << " Address of longVar:\ t" ;
18:       cout <<  &longVar  << "\ n";
19:
20:       cout << "sVar:\ t"     << sVar;
21:       cout     << " Address of sVar:\ t" ;
22:       cout     <<  &sVar     << "\ n";
23:
24:       return 0;
25:}
```

РЕЗУЛЬТАТ

```
shortVar: 5       Address of shortVar: 0x8fc9:fff4
longVar:  65535   Address of longVar:  0x8fc9:fff2
sVar:    -65535   Address of sVar:     0x8fc9:ffee
```

(Ваши результаты могут отличаться от приведенных в листинге.)

АНАЛИЗ В начале программы объявляются и инициализируются три переменные: в строке 8 — переменная типа unsigned short, в строке 9 — типа unsigned long, а в строке 10 — типа long. Затем в строках 12–16 выводятся значения и адреса этих переменных, полученные с помощью оператора адреса (&).

При запуске программы на компьютере с процессором 80386 значение переменной shortVar равно 5, а ее адрес — 0x8fc9:fff4. Адрес размещения переменной выбирается компьютером и может изменяться при каждом последующем запуске программы. Поэтому ваши результаты могут отличаться от приведенных. Причем разница между двумя первыми адресами будет оставаться постоянной. При двухбайтовом представлении типа short эта разница составит 2 байта, а разница между третьим и четвертым адресами — 4 байта при четырехбайтовом представлении типа long. Порядок размещения этих переменных в памяти показан на рис. 8.2.

В большинстве случаев вам не придется непосредственно манипулировать адресами переменных. Важно лишь знать, какой объем памяти занимает переменная и как получить ее адрес в случае необходимости. Программист лишь указывает компилятору объем памяти, доступный для размещения статических переменных, после чего размещение переменной по определенному адресу будет выполняться автоматически. Обычно тип long имеет четырехбайтовое представление. Это означает, что для хранения переменной этого типа потребуется четыре байта машинной памяти.

Использование указателя как средства хранения адреса

Каждая переменная программы имеет свой адрес, для хранения которого можно использовать указатель на эту переменную. Причем само значение адреса знать не обязательно.

Допустим, что переменная howOld имеет тип int. Чтобы объявить указатель pAge для хранения адреса этой переменной, наберите следующий фрагмент кода:

```
int *pAge = 0;
```

Этой строкой переменная pAge объявляется указателем на тип int. Это означает, что pAge будет содержать адрес значения типа int.

Отметим, что pAge ничем не отличается от любой другой переменной. При объявлении переменной целочисленного типа (например, int) мы указываем на то, что в ней будет храниться целое число. Когда же переменная объявляется указателем на какой-либо тип, это означает, что она будет хранить адрес переменной данного типа. Таким образом, указатели являются просто отдельным типом переменных.

В данном примере переменная pAge инициализируется нулевым значением. Указатели, значения которых равны 0, называют пустыми. После объявления указателю обязательно должно присваиваться какое-либо значение. Если заранее не известно, какой адрес должен храниться в указателе, ему присваивается значение 0. Неинициализированные указатели в дальнейшем могут стать причиной больших неприятностей.

Поскольку при объявлении указателю pAge было присвоено значение 0, далее ему нужно присвоить адрес какой-либо переменной, например howOld. Как это сделать, показано ниже:

```
unsigned short int howOld = 50;      // объявляем переменную
unsigned short int *pAge = 0;        // объявляем указатель
pAge = &howOld;                      // Присвоение указателю pAge адреса переменной
howOld
```

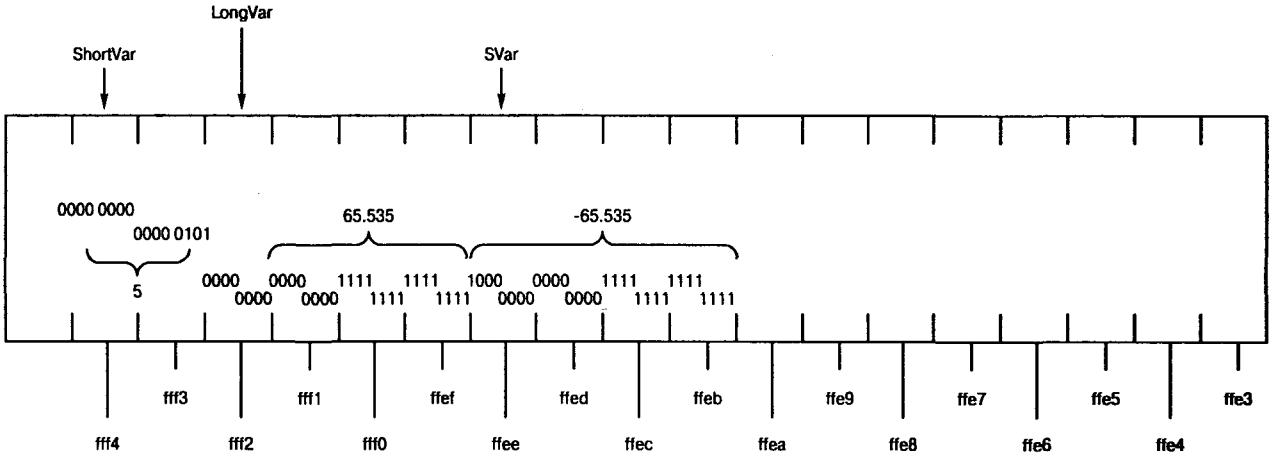


Рис. 8.2. Схема сохранения переменной в памяти

В первой строке объявлена переменная howOld типа unsigned short int и ей присвоено значение 50. Во второй строке объявлен указатель pAge на тип unsigned short int, которому присвоено значение 0. Символ "звездочка" (*), стоящий после наименования типа, указывает на то, что описанная переменная является указателем.

В последней строке указателю pAge присваивается адрес переменной howOld. На это указывает оператор адреса (&) перед именем переменной howOld. Если бы этого оператора не было, присваивался бы не адрес, а значение переменной, которое также может являться корректным адресом.

В нашем случае значением указателя pAge будет адрес переменной howOld, значение которой равно 50. Две последние строки рассмотренного фрагмента программы можно объединить в одну:

```
unsigned short int howOld = 50;          // объявляем переменную
unsigned short int * pAge = &howOld;     // объявляем указатель на переменную howOld
```

Теперь указатель pAge содержит адрес переменной howOld. С помощью этого указателя можно получить и значение переменной, на которую он указывает. В нашем примере это значение равно 50. Обращение к значению howOld посредством указателя pAge называется операцией *разыменования* или *косвенного обращения*, поскольку осуществляется неявное обращение к переменной howOld, адрес которой содержится в указателе. Далее вы узнаете, как с помощью разыменовывания возвращать значения переменных.

Косвенное обращение подразумевает получение значения переменной, адрес которой содержится в указателе, а оператор разыменования позволяет извлечь это значение.

Имена указателей

Поскольку указатели являются обычными переменными, называть их можно любыми корректными для переменных именами. Для выделения указателей среди других переменных многие программисты используют перед их именами символ "p" (от англ. pointer), например pAge или pNumber.

Оператор разыменования

Оператор косвенного обращения (или оператор разыменования) позволяет получить значение, хранящееся по адресу, записанному в указателе.

В отличие от указателя, при обращении к обычной переменной осуществляется доступ непосредственно к ее значению. Например, чтобы объявить новую переменную типа unsigned short int, а затем присвоить ей значение другой переменной, можно написать следующее:

```
unsigned short int yourAge;
yourAge = howOld;
```

При косвенном доступе будет получено значение, хранящееся по указанному адресу. Чтобы присвоить новой переменной yourAge значение howOld, используя указатель pAge, содержащий ее адрес, напишите следующее:

```
unsigned short int yourAge;
yourAge = *pAge;
```

Оператор разыменования (*) перед переменной pAge может рассматриваться как "значение, хранящееся по адресу". Таким образом, вся операция присваивания означает: "получить значение, хранящееся по адресу, записанному в pAge, и присвоить его переменной yourAge".

ПРИМЕЧАНИЕ

Оператор разыменования можно использовать с указателями двумя разными способами: для объявления указателя и для его разыменовывания. В случае объявления указателя символ звездочки сигнализирует компилятору, что это не простая переменная, а указатель, например:

```
unsigned short * pAge = 0; // объявляется указатель
                           // на переменную типа unsigned short
```

В случае разыменовывания указателя символ звездочки означает, что операция должна производиться не над самим адресом, а над значением, сохраненным по адресу, который хранится в указателе:

```
*pAge = 5; //присваивает значение 5 переменной по адресу в указателе pAge
```

Также не путайте оператор разыменовывания с оператором умножения (*). Компилятор по контексту определяет, какой именно оператор используется в данном случае.

Указатели, адреса и переменные

Чтобы овладеть навыками программирования на C++, вам в первую очередь необходимо понимать, в чем различие между указателем, адресом, хранящимся в указателе, и значением, записанным по адресу, хранящемуся в указателе. В противном случае это может привести к ряду серьезных ошибок при написании программ.

Рассмотрим еще один фрагмент программы:

```
int theVariable = 5;
int * pPointer = &theVariable ;
```

В первой строке объявляется переменная целого типа `theVariable`. Затем ей присваивается значение `5`. В следующей строке объявляется указатель на тип `int`, которому присваивается адрес переменной `theVariable`. Переменная `pPointer` является указателем и содержит адрес переменной `theVariable`. Значение, хранящееся по адресу, записанному в `pPointer`, равно `5`. На рис. 8.3 схематически показана структура этих переменных.

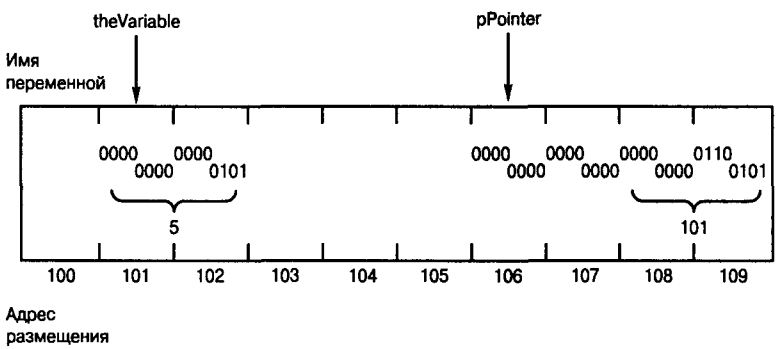


Рис. 8.3. Схема распределения памяти

Обращение к данными через указатели

После того как указателю присвоен адрес какой-либо переменной, его можно использовать для работы со значением этой переменной. В листинге 8.2 показан пример обращения к значению локальной переменной через указатель на нее.

Листинг 8.2. Обращение к данным через указатели

```
1:    // Листинг 8.2. Использование указателей
2:
3:    #include <iostream.h>
4:
5:    typedef unsigned short int USHORT;
6:    int main()
7:    {
8:            USHORT myAge;      // переменная
9:            USHORT * pAge = 0;      // указатель
10:           myAge = 5;
11:           cout << "myAge: " << myAge << "\ n";
12:           pAge = &myAge;      // заносим в pAge адрес myAge
13:           cout << "*pAge: " << *pAge << "\ n\ n";
14:           cout << "*pAge = 7\ n";
15:           *pAge = 7;      // присваиваем myAge значение 7
16:           cout << "*pAge: " << *pAge << "\ n";
17:           cout << "myAge: " << myAge << "\ n\ n";
18:           cout << "myAge = 9\ n";
19:           myAge = 9;
20:           cout << "myAge: " << myAge << "\ n";
21:           cout << "*pAge: " << *pAge << "\ n";
22:
23:           return 0;
24:   }
```

РЕЗУЛЬТАТ

```
myAge: 5
*pAge: 5

*pAge = 7
*pAge: 7
myAge: 7

myAge = 9
myAge: 9
*pAge: 9
```

АНАЛИЗ

В программе объявлены две переменные: `myAge` типа `unsigned short` и `pAge`, являющаяся указателем на этот тип. В строке 10 переменной `pAge` присваивается значение `5`, а в строке 11 это значение выводится на экран.

Затем в строке 12 указателю `pAge` присваивается адрес переменной `myAge`. С помощью операции разыменования значение, записанное по адресу, хранящемуся в указателе `pAge`, выводится на экран (строка 13). Как видим, полученный результат совпадает со значением

переменной myAge. В строке 15 переменной, адрес которой записан в pAge, присваивается значение 7. После выполнения такой операции переменная myAge будет содержать значение 7. Убедиться в этом можно после вывода этих значений (строки 16, 17).

В строке 19 значение myAge опять изменяется. Теперь этой переменной присваивается число 9. Затем в строках 20 и 21 мы обращаемся к этому значению непосредственно (через переменную) и путем разыменования указателя на нее.

Использование адреса, хранящегося в указателе

При работе с указателями в большинстве случаев не приходится иметь дело со значениями адресов, записанных в указателях. В предыдущих разделах отмечалось, что после присвоения указателю адреса переменной значением указателя будет именно этот адрес. Почему бы не проверить это утверждение? Для этого можно воспользоваться программой, приведенной в листинге 8.3.

Листинг 8.3. Что же записано в указателе?

```
1:    // Листинг 8.3. Что же хранится в указателе?
2:
3:    #include <iostream.h>
4:
5:
6:    int main()
7:    {
8:       unsigned short int myAge = 5, yourAge = 10;
9:       unsigned short int * pAge = &myAge;  // Указатель
10:      cout << "myAge:\ t" << myAge <<  "\ t yourAge:\ t" << yourAge << "\ n";
11:      cout << "&myAge:\ t" << &myAge <<  "\ t&yourAge:\ t" << &yourAge << "\ n";
12:      cout << "pAge:\ t" << pAge << "\ n";
13:      cout << "*pAge:\ t" << *pAge << "\ n";
14:      pAge = &yourAge;     // переприсвоение указателя
15:      cout << "myAge:\ t" << myAge <<  "\ t yourAge:\ t" << yourAge << "\ n";
16:      cout << "&myAge:\ t" << &myAge <<  "\ t&yourAge:\ t" << &yourAge << "\ n";
17:      cout << "pAge:\ t" << pAge << "\ n";
18:      cout << "*pAge:\ t" << *pAge << "\ n";
19:      cout << "&pAge:\ t" << &pAge << "\ n";
20:      return 0;
21:   }
```

РЕЗУЛЬТАТ

```
myAge:        5          yourAge:  10
&myAge:       0x355C     &yourAge: 0x355E
pAge:         0x355C
*pAge:        5
myAge:        5          yourAge:  10
&myAge:       0x355C     &yourAge: 0x355E
pAge:         0x355E
*pAge:        10
&pAge:        0x355A
```

(Ваши результаты могут отличаться от приведенных.)

В строке 8 объявляются две переменные типа unsigned short — myAge и yourAge. Далее, в строке 9, объявляется указатель на этот тип (pAge). Этому указателю присваивается адрес переменной myAge.

В строках 10 и 11 значения и адреса переменных pAge и myAge выводятся на экран. Обращение к значению переменной myAge путем разыменования указателя pAge выполняется в строке 13. Перед тем как перейти к дальнейшему изучению материала, подумайте, все ли вам понятно в рассмотренном примере. Еще раз проанализируйте текст программы и результат ее выполнения.

В строке 14 указателю pAge присваивается адрес переменной yourAge. После этого на экран выводятся новые значения и адреса переменных. Проанализировав результат программы, можно убедиться, что указатель pAge действительно содержит адрес переменной yourAge, а с помощью разыменования этого указателя можно получить ее значение.

Строка 19 выводит на экран значение адреса указателя pAge. Как любая другая переменная, указатель также имеет адрес, значение которого может храниться в другом указателе. О хранении в указателе адреса другого указателя речь пойдет несколько позже.

Рекомендуется

Используйте оператор разыменовывания (*) для получения доступа к данным, сохраненным по адресу, содержащемуся в указателе.

Инициализируйте указатель нулевым значением при объявлении, если заранее не известно, для указания на какую переменную он будет использоваться.

Рекомендуется

Помните о разнице между адресом в указателе и значением переменной, на которую ссылается этот указатель.

Использование указателей

Чтобы объявить указатель, запишите вначале тип переменной или объекта, на который будет ссылаться этот указатель, затем поместите символ звездочки (*), а за ним — имя нового указателя, например:

```
unsigned short int * pPointer = 0;
```

Чтобы присвоить указателю адрес переменной, установите перед именем переменной оператор адреса (&), как в следующем примере:

```
unsigned short int theVariable = 5;
unsigned short int * pPointer = & theVariable;
```

Чтобы разыменовать указатель, установите перед его именем оператор разыменовывания (*):

```
unsigned short int theValue = *pPointer
```

Для чего нужны указатели

В предыдущих разделах мы детально рассмотрели процедуру присвоения указателю адреса другой переменной. Однако на практике такое использование указателей встречается достаточно редко. К тому же, зачем задействовать еще и указатель, если значение уже хранится в другой переменной? Рассмотренные выше примеры приведены только для демонстрации механизма работы указателей. Теперь, после описания синтаксиса, используемого в C++ для работы с указателями, можно переходить к более профессиональному их применению. Наиболее часто указатели применяются в следующих случаях:

- для размещения данных в свободных областях памяти и доступа к ним;
- для доступа к переменным и функциям классов;
- для передачи параметров в функции по ссылке.

Оставшаяся часть главы посвящена динамическому управлению данными и операциям с переменными и функциями классов.

Память стековая и динамически распределяемая

Если вы помните, на занятии 5 приводились условное разделение памяти на пять областей:

- область глобальных переменных;
- свободная, или динамически распределяемая память;
- регистровая память (регистры);
- сегменты программы;
- стековая память.

Локальные переменные и параметры функций размещаются в стековой памяти. Программный код хранится в сегментах, глобальные переменные — в области глобальных переменных. Регистровая память предназначена для хранения внутренних служебных данных программы, таких как адрес вершины стека или адрес команды. Остальная часть памяти составляет так называемую свободную память — область памяти, динамически распределяемую между различными объектами.

Особенностью локальных переменных является то, что после выхода из функции, в которой они были объявлены, память, выделенная для их хранения, освобождается, а значения переменных уничтожаются.

Глобальные переменные позволяют частично решить эту проблему ценой неограниченного доступа к ним из любой точки программы, что значительно усложняет восприятие текста программы. Использование динамической памяти полностью решает обе проблемы.

Чтобы понять, что же такое динамическая память, попытайтесь представить область памяти, разделенную на множество пронумерованных ячеек, в которых записана информация. В отличие от стека переменных, ячейкам свободной памяти нельзя присвоить имя. Доступ к ним осуществляется посредством указателя, хранящего адрес нужной ячейки.

Чтобы лучше понять изложенное выше, рассмотрим пример. Допустим, вам дали номер телефона службы заказов товара по почте. Придя домой, вы занесли этот номер в память вашего телефона, а листок бумаги, на котором он был записан, выбросили. Нажимая на кнопку телефона, вы соединяетесь со службой заказа. Для вас не имеет значения номер и адрес этой службы, поскольку вы уже получили доступ к интересующей вас информации. Служба заказов в данном случае является моделью динамической памяти. Вы не знаете, где именно находится нужная вам информация, но знаете, как ее получить. Для обращения к значению используется его адрес, роль которого играет телефонный номер. Причем помнить адрес (или номер) не обязательно — достаточно лишь записать его значение в указатель (или телефон). После этого, используя указатель, можно извлечь нужное значение, даже не зная место его расположения.

Что касается стека переменных, то по завершении работы функции он очищается. В результате все локальные переменные оказываются вне области видимости и их значения уничтожаются. В отличие от стека, динамическая память не очищается до завершения работы программы, поэтому в таком случае освобождением памяти должен заниматься программист.

Важным преимуществом динамической памяти является то, что выделенная в ней облаять памяти не может использоваться до тех пор, пока явно не будет освобождена. Поэтому, если память в динамической области выделяется во время работы функции, ее можно будет использовать даже после завершения работы.

Еще одним преимуществом динамического выделения памяти перед использованием глобальных переменных является то, что доступ к данным можно получить только из функций, в которых есть доступ к указателю, хранящему нужный адрес. Такой способ доступа позволяет жестко контролировать характер манипулирования данными, а также избегать нежелательного или случайного их изменения.

Для работы с данными описанным способом прежде всего нужно создать указатель на ячейки динамической области памяти. О том, как это сделать, читайте в следующем разделе.

Оператор new

Для выделения памяти в области динамического распределения используется ключевое слово `new`. После `new` следует указать тип объекта, который будет размещаться в памяти. Это необходимо для определения размера области памяти, требуемой для хранения объекта. Написав, например, `new unsigned short int`, мы выделим два байта памяти, а строка `new long` динамически выделит четыре байта.

В качестве результата оператор `new` возвращает адрес выделенного фрагмента памяти. Этот адрес должен присваиваться указателю. Например, для выделения памяти в области динамического обмена переменной типа `unsigned short` можно использовать такую запись:

```
unsigned short int * pPointer;
pPointer = new unsigned short int;
```

Или выполнить те же действия, но в одной сороке:

```
unsigned short int * pPointer = new unsigned short int;
```

В каждом случае указатель `pPointer` будет указывать на ячейку памяти в области динамического обмена, содержащую значение типа `unsigned short`. Теперь `pPointer` можно использовать как любой другой указатель на переменную этого типа. Чтобы занести в выделенную область памяти какое-нибудь значение, напишите такую строку:

```
*pPointer = 72;
```

Эта строка означает следующее: "записать число 72 в память по адресу, хранящемуся в pPointer".

Ввиду того что память является ограниченным ресурсом, попытка выделения памяти оператором new может оказаться неудачной. В этом случае возникнет исключительная ситуация, которая рассматривается на занятии 20.

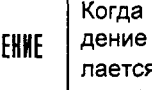
Оператор delete

Когда память, выделенная под переменную, больше не нужна, ее следует освободить. Делается это с помощью оператора delete, после которого записывается имя указателя. Оператор delete освобождает область памяти, определенную указателем. Необходимо помнить, что указатель, в отличие от области памяти, на которую он указывает, является локальной переменной. Поэтому после выхода из функции, в которой он был объявлен, этот указатель станет недоступным. Однако область памяти, выделенная оператором new, на которую сослался указатель, при этом не освобождается. В результате часть памяти окажется недоступной. Программисты называют такую ситуацию *утечкой памяти*. Такое название полностью соответствует действительности, поскольку до завершения работы программы эту память использовать нельзя, она как бы "вытекает" из вашего компьютера.

Чтобы освободить выделенную память, используйте ключевое слово delete, например:

```
delete pPointer;
```

На самом деле при этом происходит не удаление указателя, а освобождение области памяти по адресу, записанному в нем. При освобождении выделенной памяти с самим указателем ничего не происходит и ему можно присвоить другой адрес. Листинг 8.4 показывает, как выделить память для динамической переменной, использовать ее, а затем освободить.



Когда оператор delete применяется к указателю, происходит освобождение области динамической памяти, на которую этот указатель ссылается. Повторное применение оператора delete к этому же указателю приведет к зависанию программы. Рекомендуется при освобождении области динамической памяти присваивать связанному с ней указателю нулевое значение. Вызов оператора delete для нулевого указателя пройдет совершенно безболезненно для программы, например:

```
Animal *pDog = new Animal;
delete pDog;   // освобождение динамической памяти
     pDog = 0  // присвоение указателю нулевого значения
               // ...
delete pDog;   // бессмысленная, но совершенно безвредная строка
```

Листинг 8.4. Выделение, использование и освобождение динамической памяти

```
1:    // Листинг 8.4.
2:    // Выделение, использование и освобождение динамической памяти
3:
4:    #include <iostream.h>
5:    int main()
6:    {
```

```
7:          int localVariable = 5;
8:          int * pLocal= &localVariable;
9:          int * pHeap = new int;
10:     *pHeap = 7;
11:     cout << "localVariable: " << localVariable << "\ n";
12:     cout << "*pLocal: " << *pLocal << "\ n";
13:     cout << "*pHeap: " << *pHeap << "\ n";
14:     delete pHeap;
15:     pHeap = new int;
16:     *pHeap = 9;
17:     cout << "*pHeap: " << *pHeap << "\ n";
18:     delete pHeap;
19:     return 0;
20:     }
```

Результат:

```
localVariable: 5
*pLocal: 5
*pHeap: 7
*pHeap: 9
```

Анализ: В строке 7 объявляется и инициализируется локальная переменная localVariable. Затем объявляется указатель, которому присваивается адрес этой переменной (строка 8). В строке 9 выделяется память для переменной типа int и адрес выделенной области помещается в указатель pHeap. Записав по адресу, содержащемуся в pHeap, значение 7, можно удостовериться в том, что память была выделена корректно (строка 10). Если бы память под переменную не была выделена, то при выполнении этой строки появилось бы сообщение об ошибке.

Чтобы не перегружать примеры излишней информацией, мы опускаем всякого рода проверки. Однако во избежание аварийного завершения программы при решении реальных задач такой контроль обязательно должен выполняться.

В строке 10, после выделения памяти, по адресу в указателе записывается значение 7. Затем в строках 11 и 12 значения локальной переменной и указателя pLocal выводятся на экран. Вполне понятно, почему эти значения равны. Далее, в строке 13, выводится значение, записанное по адресу, хранящемуся в указателе pHeap. Таким образом, подтверждается, что значение, присвоенное в строке 10, действительно доступно для использования.

Освобождение области динамической памяти, выделенной в строке 9, осуществляется оператором delete в строке 14. Освобожденная память становится доступной для дальнейшего использования, и ее связь с указателем разрывается. После этого указатель pHeap может использоваться для хранения нового адреса. В строках 15 и 16 выполняется повторное выделение памяти и запись значения по соответствующему адресу. Затем в строке 17 это значение выводится на экран, после чего память освобождается.

Вообще говоря, строка 18 не является обязательной, так как после завершения работы программы вся выделенная в ней память автоматически освобождается. Однако явное освобождение памяти считается как бы правилом хорошего тона в программировании. Кроме того, это может оказаться полезным при редактировании программы.

Что такое утечка памяти

При невнимательной работе с указателями может возникнуть эффект так называемой утечки памяти. Это происходит, если указателю присваивается новое значение, а память, на которую он ссылался, не освобождается. Ниже показан пример такой ситуации.

```
1:    unsigned short int * pPointer = new unsigned short int;
2:    *pPointer = 72;
3:    pPointer = new unsigned short int;
4:    *pPointer = 84;
```

В строке 1 объявляется указатель и выделяется память для хранения переменной типа `unsigned short int`. В следующей строке в выделенную область записывается значение `72`. Затем в строке 3 указателю присваивается адрес другой области памяти, в которую записывается число `84` (строка 4). После выполнения таких операций память, содержащая значение `72`, оказывается недоступной, поскольку указателю на эту область было присвоено новое значение. В результате невозможно ни использовать, ни освободить зарезервированную память до завершения программы.

Правильнее было бы написать следующее:

```
1:    unsigned short int * pPointer = new unsigned short int;
2:    *pPointer = 72;
3:    delete pPointer;
4:    pPointer = new unsigned short int;
5:    *pPointer = 84;
```

В этом случае память, выделенная под переменную, корректно освобождается (строка 3).

Каждый раз, когда в программе используется оператор `new`, за ним должен следовать оператор `delete`. Очень важно следить, какой указатель ссылается на выделенную область динамической памяти, и вовремя освобождать ее.

Размещение объектов в области динамически памяти

Аналогично созданию указателя на целочисленный тип, можно динамически размещать в памяти любые объекты. Например, если вы объявили объект класса `Cat`, для манипулирования этим объектом можно создать указатель, в котором будет храниться его адрес, — ситуация, абсолютно аналогичная размещению переменных в стеке. Синтаксис этой операции такой же, как и для целочисленных типов:

```
Cat *pCat = new Cat;
```

В данном случае использование оператора `new` вызывает конструктор класса по умолчанию, т.е. конструктор, использующийся без параметров. Важно помнить, что при создании объекта класса конструктор вызывается всегда, независимо от того, размещается объект в стеке или в области динамического обмена.

Удаление объектов

При использовании оператора delete, за которым следует идентификатор указателя на объект, вызывается деструктор соответствующего класса. Это происходит еще до освобождения памяти и возвращения ее в область динамического обмена. В деструкторе, как правило, освобождается вся память, занимаемая объектом класса. Пример динамического размещения и удаления объектов показан в листинге 8.5.

Листинг 8.5. Размещение и удаление объектов в области динамического обмена

```
1:    // Листинг 8.5.
2:    // Размещение и удаление объектов в области динамического обмена
3:
4:    #include <iostream.h>
5:
6:    class SimpleCat
7:    {
8:         public:
9:              SimpleCat();
10:             ~SimpleCat();
11:    private:
12:             int itsAge;
13:    };
14:
15:    SimpleCat::SimpleCat()
16:    {
17:         cout << "Constructor called.\ n";
18:         itsAge = 1;
19:    }
20:
21:    SimpleCat::~SimpleCat()
22:    {
23:         cout << "Destructor called.\ n";
24:    }
25:
26:    int main()
27:    {
28:         cout << "SimpleCat Frisky...\ n";
29:         SimpleCat Frisky;
30:         cout << "SimpleCat *pRags = new SimpleCat...\ n";
31:         SimpleCat * pRags = new SimpleCat;
32:         cout << "delete pRags...\ n";
33:         delete pRags;
34:         cout << "Exiting, watch Frisky go...\ n";
35:         return 0;
36:    }
```

РЕЗУЛЬТАТ

```
SimpleCat Frisky...
Constructor called.
SimpleCat *pRags = new SimpleCat..
Constructor called.
delete pRags...
Destructor called.
Exiting, watch Frisky go...
Destructor called.
```

АНАЛИЗ

В строках 6–13 приведено описание простейшего класса `SimpleCat`. Описание конструктора класса находится в строке 9, а его тело — в строках 15–19. Деструктор описан в строке 10, его тело — в строках 21–24.

В строке 29 создается экземпляр описанного класса, который размещается в стеке. При этом происходит неявный вызов конструктора класса `SimpleCat`. Второй объект класса создается в строке 31. Для его хранения динамически выделяется память и адрес записывается в указатель `pRags`. В этом случае также вызывается конструктор. Деструктор класса `SimpleCat` вызывается в строке 33 как результат применения оператора `delete` к указателю `pRags`. При выходе из функции переменная `Frisky` оказывается за пределами области видимости и для нее также вызывается деструктор.

Доступ к членам класса

Для локальных переменных, являющихся объектами класса, доступ к членам класса осуществляется с помощью оператора прямого доступа (.). Для экземпляров класса, созданных динамически, оператор прямого доступа применяется для объектов, полученных разыменованием указателя. Например, для вызова функции-члена `GetAge` нужно написать:

```
(*pRags).GetAge();
```

Скобки указывают на то, что оператор разыменования должен выполняться еще до вызова функции `GetAge()`.

Такая конструкция может оказаться достаточно громоздкой. Решить эту проблему позволяет специальный *оператор косвенного обращения к члену класса*, по написанию напоминающий стрелку (->). Для набора этого оператора используется непрерывная последовательность двух символов: тире и знака "больше". В C++ эти символы рассматриваются как один оператор. Листинг 8.6 иллюстрирует пример обращения к переменным и функциям класса, экземпляр которого размещен в области динамического обмена.

Листинг 8.6. Доступ к данным объекта в области динамического обмена

```
1:    // Листинг 8.6.
2:    // Доступ к данным объекта в области динамического обмена
3:
4:    #include <iostream.h>
5:
6:    class SimpleCat
7:    {
8:         public:
```

```
9:              SimpleCat() { itsAge = 2; }
10:         ~SimpleCat() { }
11:         int GetAge() const {  return itsAge; }
12:         void SetAge(int age) {  itsAge = age; }
13:     private:
14:         int itsAge;
15:     };
16:
17:     int main()
18:     {
19:         SimpleCat * Frisky = new SimpleCat;
20:         cout << "Frisky " << Frisky->GetAge() << " years old\ n";
21:         Frisky->SetAge(5);
22:         cout << "Frisky " << Frisky->GetAge() << " years old\ n";
23:         delete Frisky;
24:         return 0;
25:     }
```

РЕЗУЛЬТАТ

```
Frisky 2 years old
Frisky 5 years old
```

АНАЛИЗ В строке 19 в области динамического обмена выделяется память для хранения экземпляра класса `SimpleCat`. Конструктор, вызываемый по умолчанию, присваивает новому объекту возраст два года. Это значение получено как результат выполнения функции-члена `GetAge()`, которая вызывается в строке 20. Поскольку мы имеем дело с указателем на объект, для вызова функции используется оператор косвенного обращения к члену класса (->). В строке 21 для установки нового значения возраста вызывается метод `SetAge()`, а повторный вызов функции `GetAge()` (строка 22) позволяет получить это значение.

Динамическое размещение членов класса

В качестве членов класса могут выступать и указатели на объекты, размещенные в области динамического обмена. В таких случаях выделение памяти для хранения этих объектов осуществляется в конструкторе или в одном из методов класса. Освобождение памяти происходит, как правило, в деструкторе (листинг 8.7.).

Листинг 8.7. Указатели как члены класса

```
1:     // Листинг 8.7.
2:     // Указатели как члены класса
3:
4:     #include <iostream.h>
5:
6:     class SimpleCat
7:     {
8:         public:
9:             SimpleCat();
```

```
10:          ~SimpleCat();
11:          int GetAge() const {  return *itsAge; }
12:          void SetAge(int age) {  *itsAge = age; }
13:
14:          int GetWeight() const {  return *itsWeight; }
15:          void setWeight (int weight) {  *itsWeight = weight; }
16:
17:     private:
18:          int * itsAge;
19:          int * itsWeight;
20:     } ;
21:
22:     SimpleCat::SimpleCat()
23:     {
24:          itsAge = new int(2);
25:          itsWeight = new int(5);
26:     }
27:
28:     SimpleCat::~SimpleCat()
29:     {
30:          delete itsAge;
31:          delete itsWeight;
32:     }
33:
34:     int main()
35:     {
36:          SimpleCat *Frisky = new SimpleCat;
37:          cout << "Frisky " << Frisky->GetAge() << " years old\ n";
38:          Frisky->SetAge(5);
39:          cout << "Frisky " << Frisky->GetAge() << " years old\ n";
40:     delete Frisky;
41:     return 0;
42:     }
```

РЕЗУЛЬТАТ

```
Frisky 2 years old
Frisky 5 years old
```

АНАЛИЗ

Объявляем класс, переменными-членами которого являются два указателя на тип int. В конструкторе класса (строки 22–26) выделяется память для хранения этих переменных, а затем им присваиваются начальные значения.

Выделенная под переменные-члены память освобождается в деструкторе (строки 28–32). После освобождения памяти в деструкторе присваивать указателям нулевые значения не имеет смысла, поскольку уничтожается и сам экземпляр класса. Такая ситуация является одним из тех случаев, когда после освобождения памяти указателю можно не присваивать значение 0.

При выполнении функции, из которой осуществляется обращение к переменным класса (в данном случае main()), вы можете и не знать, каким образом выполняется это обращение. Вы лишь вызываете соответствующие методы класса (GetAge() и SetAge()), а все операции с памятью выполняются внутренними механизмами класса.

При уничтожении объекта Frisky (строка 40) вызывается деструктор класса SimpleCat. В деструкторе память, выделенная под члены класса, освобождается. Если один из членов класса является объектом другого определенного пользователем класса, происходит вызов деструктора этого класса.

Вопросы и ответы

Если я объявляю объект класса, хранящийся в стеке, а этот объект, в свою очередь, имеет переменные-члены, хранящиеся в области динамического обмена, то какие части объекта будут находиться в стеке, а какие — в области динамического обмена?

```
#include <iostream.h>
    class SimpleCat
    {
    public:
        SimpleCat();
        ~SimpleCat();
        int GetAge() const {return *itsAge; }
        // другие методы

    private:
        int * itsAge;
        int * itsWeight;
};
SimpleCat::SimpleCat()
{
        itsAge = new int(2);
        itsWeight = new int(5);
}

SimpleCat::~SimpleCat()
{
        delete itsAge;
        delete itsWeight;
}

int main()
{
        SimpleCat Frisky;
        cout << "Frisky is " << Frisky.GetAge() << " years old\n";
        return 0;
}
```

В стеке будет находиться локальная переменная Frisky. Эта переменная содержит два указателя, каждый из которых занимает по четыре байта стековой памяти для хранения адресов целочисленных значений, размещенных в области динамического обмена. Таким образом, объект Frisky займет восемь байтов стековой памяти и восемь — в области динамического обмена.

Конечно, для данного примера динамическое размещение в памяти переменных-членов не обязательно. Однако в реальных программах такой способ хранения данных может оказаться достаточно эффективным.

Важно четко поставить задачу, которую необходимо решить. Помните, что любая программа начинается с проектирования. Допустим, например, что требуется создать класс, членом которого является объект другого класса, причем второй объект может создаваться еще до возникновения первого и оставаться после его уничтожения. В этом случае доступ ко второму объекту должен осуществляться только по ссылке, т.е. с использованием указателя.

Допустим, первым объектом является окно, а вторым — документ. Вполне понятно, что окно должно иметь доступ к документу. С другой стороны, продолжительность существования документа никак не контролируется окном. Поэтому для окна важно хранить лишь ссылку на этот документ.

Об использовании ссылок речь идет на затянии 9.

Указатель this

Каждый метод класса имеет скрытый параметр — указатель `this`. Этот указатель содержит адрес текущего объекта. Рассмотренные в предыдущем разделе функции `GetAge()` и `SetAge()` также содержат этот параметр.

В листинге 8.8 приведен пример использования указателя `this` в явном виде.

Листинг 8.8. Указатель `this`

```
1:    // Листинг 8.8.
2:    // Указатель this
3:
4:    #include <iostream.h>
5:
6:    class Rectangle
7:    {
8:         public:
9:              Rectangle();
10:             ~Rectangle();
11:             void SetLength(int length) {  this->itsLength = length; }
12:             int GetLength() const {  return this->itsLength; }
13:
14:             void SetWidth(int width) {  itsWidth = width; }
15:             int GetWidth() const {  return itsWidth; }
16:
17:        private:
18:             int itsLength;
19:             int itsWidth;
20:    };
21:
22:    Rectangle::Rectangle()
23:    {
24:         itsWidth = 5;
25:         itsLength = 10;
26:    }
27:    Rectangle::~Rectangle()
28:    { }
29:
```

```
30:     int main()
31:     {
32:          Rectangle theRect;
33:          cout << "theRect is " << theRect.GetLength() << " meters long.\ n";
34:          cout << "theRect is " << theRect.GetWidth() << " meters wide.\ n";
35:          theRect.SetLength(20);
36:          theRect.SetWidth(10);
37:          cout << "theRect is " << theRect.GetLength()<< " meters long.\ n";
38:          cout << "theRect is " << theRect.GetWidth()<< " meters wide.\ n";
39:          return 0;
40:     }
```

РЕЗУЛЬТАТ

```
theRect is 10 meters long.
theRect is 5 meters wide.
theRect is 20 meters long.
theRect is 10 meters wide.
```

АНАЛИЗ В функциях `SetLength()` и `GetLength()` при обращении к переменным класса `Rectangle` указатель `this` используется в явном виде. В функциях `SetWidth()` и `GetWidth()` такое обращение осуществляется неявно. Несмотря на различие в синтаксисе, оба варианта идентичны.

На самом деле роль указателя `this` намного важнее, чем это может показаться. Поскольку `this` является указателем, он содержит адрес текущего объекта и в этой роли может оказаться достаточно мощным инструментом.

При обсуждении проблемы перегрузки операторов (занятие 10) будет приведено несколько реальных примеров использования указателя `this`. В данный момент вам необходимо понимать, что `this` — это указатель, хранящий адрес объекта, в котором он используется.

Память для указателя `this` не выделятся и не освобождается программно. Эту задачу берет на себя компилятор.

Блуждающие, дикие или зависшие указатели

Блуждающие указатели являются достаточно распространенной ошибкой программистов, обнаружить которую довольно сложно. Блуждающий (либо, как его еще называют, дикий или зависший) указатель возникает, если после удаления объекта оператором `delete` этому указателю не присвоить значение `0`. При попытке использовать такой указатель в дальнейшем результат может оказаться непредсказуемым. В лучшем случае программа завершится сообщением об ошибке.

Возникает ситуация, подобная следующей. Почтовая служба переехала в новый офис, а вы все еще продолжаете звонить по ее старому номеру телефона. Если этот номер будет просто отключен, это не приведет ни к каким негативным последствиям. А теперь представьте себе, что этот номер отдан какому-то военному заводу...

Одним словом, будьте осторожны при использовании указателей, для которых вызывался оператор `delete`. Указатель по-прежнему будет содержать адрес области памяти, однако по этому адресу уже могут находиться другие данные. В этом случае обращение по указанному адресу может привести к аварийному завершению программы. Или, что еще хуже, программа может продолжать работать, а через несколько минут

"зависнет". Такая ситуация получила название "мины замедленного действия" и является достаточно серьезной проблемой при написании программ. Поэтому во избежание неприятностей после освобождения указателя присваивайте ему значение 0.

Пример возникновения блуждающего указателя показан в листинге 8.9.

Листинг 8.9. Пример возникновения блуждающего указателя

```
1:    // Листинг 8.9.
2:    // Пример возникновения блуждающего указателя
3:    typedef unsigned short int USHORT;
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:         USHORT * pInt = new USHORT;
9:         *pInt = 10;
10:    cout << "*pInt: " << *pInt << endl;
11:    delete pInt;
12:
13:    long * pLong = new long;
14:    *pLong = 90000;
15:    cout << "*pLong: " << *pLong << endl;
16:
17:    *pInt = 20;      // Присвоение освобожденному указателю
18:
19:    cout << "*pInt: " << *pInt  << endl;
20:    cout << "*pLong: " << *pLong  << endl;
21:    delete pLong;
22:    return 0;
23:    }
```

РЕЗУЛЬТАТ:

```
*pInt:   10
*pLong:  90000
*pInt:   20
*pLong:  65556
```

(Ваши результаты могут отличаться от приведенных.)

АНАЛИЗ: В строке 8 переменная pInt объявляется как указатель на тип USHORT. Выделяется память для хранения этого типа данных. В строке 9 по адресу в этом указателе записывается значение 10, а в строке 10 оно выводится на экран. Затем память, выделенная для pInt, освобождается оператором delete. После этого указатель оказывается зависшим, или блуждающим.

В сроке 13 объявляется новый указатель (pLong) и ему присваивается адрес выделенной оператором new области памяти. В строке 14 по адресу в указателе записывается число 90000, а в строке 15 это значение выводится на экран.

В строке 20 по адресу, занесенному в pInt, записывается значение 20. Однако, вследствие того что выделенная для этого указателя память была освобождена, такая операция является некорректной. Последствия такого присваивания могут оказаться непредсказуемыми.

В строке 19 новое значение pInt выводится на экран. Это значение, как и ожидалось, равно 20. В строке 20 выводится значение указателя pLong. К удивлению, обнаруживаем, что оно равно 65556. Возникает два вопроса.

1. Как могло измениться значение pLong, если мы его даже не использовали?
2. Куда делось число 20, присвоенное в строке 17?

Как вы, наверное, догадались, эти два вопроса связаны. При присвоении в строке 17 число 20 записывается в область памяти, на которую до этого указывал pInt. Но, так как память была освобождена в строке 11, компилятор использовал эту область для записи других данных. При объявлении указателя pLong (строка 13) была зарезервирована область памяти, на которую раньше ссылался указатель pInt. Заметим, что на некоторых компьютерах этого могло не произойти. Записывая число 20 по адресу, на который до этого указывал pInt, мы искажаем значение pLong, хранящееся по этому же адресу. Вот к чему может привести некорректное использование блуждающих указателей.

Локализовать такую ошибку достаточно сложно, поскольку искаженное значение никак не связано с блуждающим указателем. Результатом некорректного использования указателя pInt стало изменение значения pLong. В больших программах отследить возникновение подобной ситуации особенно сложно.

В качестве небольшого отступления рассмотрим, как по адресу в указателе pLong оказалось число 65556.

1. Указатель pInt ссылается на область памяти, в которую мы записали число 10.
2. Оператором delete мы позволяем компилятору использовать эту память для хранения других данных. Далее по этому же адресу записывается значение pLong.
3. Переменной *pLong присваивается значение 90000. Поскольку в компьютере использовалось четырехбайтовое представление типа long с перестановкой байтов, на машинном уровне число 90000 (00 01 5F 90) выглядело как 5F 90 00 01.
4. Затем указателю pInt присваивается значение 20, что эквивалентно 00 14 в шестнадцатеричной системе. Вследствие того что указатели ссылаются на одну и ту же область памяти, два первых байта числа 90000 перезаписываются. В результате получаем число 00 14 00 01.
5. При выводе на экран значения в указателе pLong порядок байтов изменяется на 00 01 00 14, что эквивалентно числу 65556.

Рекомендуется

Создавайте объекты в области динамического обмена.

Применяйте оператор delete для освобождения областей динамического обмена, которые больше не используются.

Проверяйте значения, возвращаемые оператором new.

Не рекомендуется

Не забывайте каждое выделение свободной памяти с помощью оператора new сопроводить освобождением памяти с помощью оператора delete.

Не забывайте присваивать освобожденным указателям нулевые значения.

Вопросы и ответы

Какая разница между пустым и блуждающим указателями?

Когда вы применяете к указателю оператор `delete`, освобождается область динамической памяти, на которую ссылался этот указатель, но сам указатель продолжает при этом существовать, становясь блуждающим.

Присваивая указателю нулевое значение, например следующим выражением: `myPtr = 0;`, вы тем самым превращаете блуждающий указатель в нулевой.

Еще одна опасность блуждающих указателей состоит в том, что, дважды применив к одному и тому же указателю оператор `delete`, вы тем самым создадите неопределенную ситуацию, которая может привести к зависанию программы. Этого не случится, если освобожденному указателю будет присвоено нулевое значение.

Присвоение освобожденному указателю — как блуждающему, так и нулевому — нового значения (т.е. использование выражения `myPtr = 5`) недопустимо, но если в случае с пустым указателем об этом вам сообщит компилятор, то в случае с блуждающим указателем вы узнаете об этом по зависанию программы в самый неподходящий момент.

Использование ключевого слова const при объявлении указателей

При объявлении указателей допускается использование ключевого слова `const` перед спецификатором типа или после него. Корректны, например, следующие варианты объявления:

```
const int * pOne;
int * const pTwo;
const int * const pThree;
```

В этом примере `pOne` является указателем на константу типа `int`. Поэтому значение, на которое он указывает, изменять нельзя.

Указатель `pTwo` является константным указателем на тип `int`. В этом случае значение, записанное по адресу в указателе, может изменяться, но сам адрес остается неизменным.

И наконец, `pThree` объявлен как константный указатель на константу типа `int`. Это означает, что он всегда указывает на одну и ту же область памяти и значение, записанное по этому адресу, не может изменяться.

В первую очередь необходимо понимать, какое именно значение объявляется константой. Если наименование типа переменной записано после ключевого слова `const`, значит, объявляемая переменная будет константой. Если же за словом `const` следует имя переменной, константой является указатель.

```
const int * p1;  // Указатель на константу типа int
int * const p2;  // Константный указатель, всегда указывающий на одну и ту же область памяти
```

Использование ключевого слова const при объявлении указателей и функций-членов

На занятии 4 мы обсудили вопрос об использовании ключевого слова const при объявлении функций-членов классов. При объявлении функции константной попытка внести изменения в данные объекта с помощью этой функции будут пресекаться компилятором.

Если указатель на объект объявлен константным, он может использоваться для вызова только тех методов, которые также объявлены со спецификатором const (листинг 8.10).

Листинг 8.10. Указатели на константные объекты

```
1:    // Листинг 8.10.
2:    // Вызов константных методов с помощью указателей
3:
4:    #include <iostream.h>
5:
6:    class Rectangle
7:    {
8:         public:
9:              Rectangle();
10:        ~Rectangle();
11:        void SetLength(int length) {  itsLength = length; }
12:        int GetLength() const {  return itsLength; }
13:        void SetWidth(int width) {  itsWidth = width; }
14:        int GetWidth() const {  return itsWidth; }
15:
16:   private:
17:        int itsLength;
18:        int itsWidth;
19:   };
20:
21:   Rectangle::Rectangle()
22:   {
23:        itsWidth = 5;,
24:        itsLength = 10;
25:   }
26:
27:   Rectangle::~Rectangle()
28:   { }
29:
30:   int main()
31:   {
32:        Rectangle* pRect =  new Rectangle;
33:        const Rectangle * pConstRect = new Rectangle;
34:        Rectangle * const pConstPtr = new Rectangle;
35:
36:   cout << "pRect width: " << pRect->GetWidth() << " meters\ n";
37:   cout << "pConstRect width: " << pConstRect-> GetWidth() << " meters\ n";
38:   cout << "pConstPtr width: " << pConstPtr-> GetWidth() << " meters\ n";
```

```
39:
40:     pRect->SetWidth(10);
41:     // pConstRect->SetWidth(10);
42:     pConstPtr->SetWidth(10);
43:
44:     cout << "pRect width: " << pRect->GetWidth() << " meters\ n";
45:     cout<< "pConstRect width:"<< pConstRect->GetWidth() << " meters\ n";
46:     cout << "pConstPtr width: "<< pConstPtr->GetWidth() << " meters\ n";
47:     return 0;
48:     }
```

```
pRect width: 5 meters
pConstRect width: 5 meters
pConstPtr width: 5 meters
pRect width: 10 meters
pConstRect width: 5 meters
pConstPtr width: 10 meters
```

В строках 6—19 приведено описание класса `Rectangle`. Обратите внимание, что метод `GetWidth()`, описанный в строке 14, имеет спецификатор `const`. Затем в строке 32 объявляется указатель на объект класса `Rectangle`, а в строке 33 — на константный объект этого же класса. Константный указатель `pConstPrt` описывается в строке 34.

В строках 36—38 значения переменных класса выводятся на экран.

Метод `SetWidth()`, вызванный для указателя `pRect` (строка 40), устанавливает значение ширины объекта. В строке 41 показан пример использования указателя `pConstRect` для вызова метода класса. Но, так как `pConstRect` является указателем на константный объект, вызов методов без спецификатора `const` для него недоступен, поэтому данная строка закомментирована. В строке 42 происходит вызов метода `SetWidth()` для указателя `pConstPrt`. Этот указатель константный и может ссылаться только на одну область памяти, однако сам объект константным не является, поэтому данная операция полностью корректна.

Рекомендуется

Проверяйте значения, возвращаемые функцией `malloc()`.

Защищайте объекты, которые не должны изменяться в программе, с помощью ключевого слова `const` в случае передачи их как ссылок.

Рекомендуется

Передавайте как ссылки те объекты, которые должны изменяться в программе.

Передавайте как значения небольшие объекты, которые не должны изменяться в программе.

Указатель const this

После объявлении константного объекта указатель `this` также будет использоваться как константный. Следует отметить, что использование указателя `const this` допускается только в методах, объявленных со спецификатором `const`.

Более подробно этот вопрос рассматривается на следующем занятии при изучении ссылок на константные объекты.

Вычисления с указателями

Один указатель можно вычитать из другого. Если, например, два указателя ссылаются на разные элементы массива, вычитание одного указателя из другого позволяет получить количество элементов массива, находящихся между двумя заданными. Наиболее эффективно эта методика используется при обработке символьных массивов (листинг 8.11).

Листинг 8.11. Выделение слов из массива символов

```
1:    #include <iostream.h>
2:    #include <ctype.h>
3:    #include <string.h>
4:    bool GetWord(char* string, char* word, int& wordOffset);
5:    // основная программа
6:    int main()
7:    {
8:         const int bufferSize = 255;
9:         char buffer[bufferSize+1];     // переменная для хранения всей строки
10:   char word[bufferSize+1];           // переменная для хранения слова
11:   int wordOffset = 0;                // начинаем с первого символа
12:
13:   cout << "Enter a string: ";
14:   cin.getline(buffer,bufferSize);
15:
16:   while (GetWord(buffer,word,wordOffset))
17:   {
18:        cout << "Got this word: " << word << endl;
19:   }
20:
21:   return 0;
22:
23:   }
24:
25:
26:   // Функция для выделения слова из строки символов.
27:   bool GetWord(char* string, char* word, int& wordOffset)
28:   {
29:
30:        if (!string[wordOffset])     // определяет конец строки?
31:        return false;
32:
33:        char *p1, *p2;
34:        p1 = p2 = string+wordOffset;       // указатель на следующее слово
35:
36:        // удаляем ведущие пробелы
37:        for (int i = 0; i<(int)strlen(p1) && !isalnum(p1[0]); i++)
38:             p1++;
39:
```

```
40:     // проверка наличия слова
41:     if (!isalnum(p1[0]))
42:     return false;
43:
44:     // указатель p1 показывает начало следующего слова
45:     // так же как и p2
46:     p2 = p1;
47:
48:     // перемещаем p2 в конец слова
49:     while (isalnum(p2[0]))
50:          p2++;
51:
52:     // p2 указывает на конец слова
53:     // а p1 - в начало
54:     // разность указателей показывает длину слова
55:     int len = int (p2 - p1);
56:
57:     // копируем слово в буфер
58:     strncpy (word,p1,len);
59:
60:     // и добавляем символ разрыва сроки
61:     word[len]='\ 0';
62:
63:     // ищем начало следующего слова
64:     for (int i = int(p2-string); i<(int)strlen(string) && !isalnum(p2[0]); i++)
65:          p2++;
66:
67:     wordOffset = int(p2-string);
68:
69:     return true;
70:     }
```

Результат

```
Enter a string: this code first appeared in C++ Report
Got this word: this
Got this word: code
Got this word: first
Got this word: appeared
Got this word: in
Got this word: C
Got this word: Report
```

Анализ

В строке 13 пользователю предлагается ввести строку. Строка считывается функцией `GetWord()`, параметрами которой является буферизированная переменная для хранения первого слова и целочисленная переменная `WordOffset`. В строке 11 переменной `WordOffset` присваивается значение `0`. По мере ввода строки (до тех пор пока `GetWord()` не возвратит значение `0`) введенные слова отображаются на экране.

При каждом вызове функции GetWord() управление передается в строку 27. Далее, в строке 30, значение string[wordOffset] проверяется на равенство нулю. Выполнение условия означает, что мы находимся за пределами строки. Функция GetWord() возвращает значение false.

В строке 33 объявляются два указателя на переменную символьного типа. В строке 34 оба указателя устанавливаются на начало следующего слова, заданное значением переменной WordOffset. Исходно значение WordOffset равно 0, что соответствует началу строки.

С помощью цикла в строках 37 и 38 указатель p1 перемещается на первый символ, являющийся буквой или цифрой. Если такой символ не найден, функция возвращает false (строки 41 и 42).

Таким образом, указатель p1 соответствует началу очередного слова. Строка 46 присваивает указателю p2 то же значение.

В строках 49 и 50 осуществляется поиск в строке первого символа, не являющегося ни цифрой, ни буквой. Указатель p2 перемещается на этот символ. Теперь p1 и p2 указывают на начало и конец слова соответственно. Вычтем из значения указателя p2 значение p1 и преобразуем результат к целочисленному типу. Результатом выполнения такой операции будет длина очередного слова (строка 55). Затем на основании данных о начале и длине полученное слово копируется в буферную переменную.

Строкой 61 в конец слова добавляется концевой нулевой символ, служащий сигналом разрыва строки. Далее указатель p2 перемещается на начало следующего слова, а переменной WordOffset присваивается значение смещения начала очередного слова относительно начала строки. Возвращая значение true, мы сигнализируем о том, что слово найдено.

Чтобы как можно лучше разобраться в работе программы, запустите ее в режиме отладки и последовательно, шаг за шагом, проконтролируйте выполнение каждой строки.

Резюме

Указатели являются мощным средством непрямого доступа к данным. Каждая переменная имеет адрес, получить который можно с помощью оператора адреса (&). Для хранения адреса используются указатели.

Для объявления указателя достаточно установить тип объекта, адрес которого он будет содержать, а затем ввести символ "*" и имя указателя. После объявления указатель следует инициализировать. Если адрес объекта неизвестен, указатель инициализируется значением 0.

Для доступа к значению, записанному по адресу в указателе, используется оператор разыменования (*). Указатель можно объявлять константным. В этом случае не допускается присвоение данному указателю нового адреса. Указатель, хранящий адрес константного объекта, не может использоваться для изменения этого объекта.

Чтобы выделить память для хранения какого-либо объекта, используется оператор new, а затем полученный адрес присваивается указателю. Для освобождения зарезервированной памяти используется оператор delete. Сам указатель при освобождении памяти не уничтожается, поэтому освобожденному указателю необходимо присвоить нулевое значение, чтобы обезопасить его.

Вопросы и ответы

В чем состоит преимущество работы с указателями?

На этом занятии вы узнали, насколько удобно использовать доступ к объекту по его адресу и передавать параметры как ссылки. На занятии 13 будет рассмотрена роль указателей в полиморфизме классов.

Чем удобно размещение объектов в динамической области памяти?

Объекты, сохраненные в области динамического обмена, не уничтожаются при выходе из функции, в которой они были объявлены. Кроме того, это дает возможность уже в процессе выполнения программы решать, какое количество объектов требуется объявить. Более подробно этот вопрос обсуждается на следующем занятии.

Зачем ограничивать права доступа к объекту, объявляя его константным?

Следует использовать все средства, позволяющие предотвратить появление ошибок. На практике достаточно сложно отследить, в какой момент и какой функцией изменяется объект. Использование спецификатора `const` позволяет решить эту проблему.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний, а также рассматривается ряд упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Какой оператор используется для получения адреса переменной?
2. Какой оператор позволяет получить значение, записанное по адресу, содержащемуся в указателе?
3. Что такое указатель?
4. В чем различие между адресом, который хранится в указателе, и значением, записанным по этому адресу?
5. В чем различие между оператором разыменования и оператором получения адреса?
6. В чем различие между следующими объявлениями: `const int * ptrOne` и `int * const ptrTwo`?

Упражнения

1. Объясните смысл следующих объявлений переменных:
   - `int * pOne`
   - `int vTwo`
   - `int * pThree = &vTwo`

2. Допустим, в программе объявлена переменная yourAge типа unsigned short. Как объявить указатель, позволяющий манипулировать этой переменной?
3. С помощью указателя присвойте переменной yourAge значение 50.
4. Напишите небольшую программу и объявите в ней переменную типа int и указатель на этот тип. Сохраните адрес переменной в указателе. Используя указатель, присвойте переменной какое-либо значение.
5. **Жучки:** найдите ошибку в следующем фрагменте программы.

```
#include <iostream.h>
int main()
{
     int *pInt;
     *pInt = 9;
     cout << "The value at pInt: " << *pInt;
     return 0;
}
```

6. **Жучки:** найдите ошибку в следующем фрагменте программы.

```
int main()
{
     int SomeVariable = 5;
     cout << "SomeVariable: " << SomeVariable << "\ n";
     int *pVar = & SomeVariable;
     pVar = 9;
     cout << "SomeVariable: " << *pVar << "\ n";
     return 0;
}
```

День 9-й

Ссылки

На предыдущем занятии вы узнали, как пользоваться указателями для управления объектами в свободной памяти и как ссылаться на эти объекты косвенным образом. Ссылки, которые обсуждаются на этом занятии, обладают почти теми же возможностями, что и указатели, но при более простом синтаксисе. Сегодня вы узнаете:

- Что представляют собой ссылки
- Чем ссылки отличаются от указателей
- Как создать ссылки и использовать их
- Какие ограничения есть у ссылок
- Как по ссылке передаются значения и объекты в функции и из функций

Что такое ссылка

Ссылка — это то же, что и псевдоним. При создании ссылки мы инициализируем ее с помощью имени другого объекта, адресата. С этого момента ссылка действует как альтернативное имя данного объекта, поэтому все, что делается со ссылкой, в действительности происходит с этим объектом.

Для объявления ссылки нужно указать типа объекта адресата, за которым следует оператор ссылки (&), а за ним — имя ссылки. Для ссылок можно использовать любое легальное имя переменной, но многие программисты предпочитают со всеми именами ссылок использовать префикс "r". Так, если у вас есть целочисленная переменная с именем `someInt`, вы можете создать ссылку на эту переменную, написав

```
int &rSomeRef = someInt;
```

Это читается следующим образом: `rSomeRef` — это ссылка на целочисленное значение, инициализированная адресом переменной `someInt`. Создание и использование ссылок показано в листинге 9.1.

Обратите внимание на то, что оператор ссылки (&) выглядит так же, как оператор адреса, который используется для возвращения адреса при работе с указателями. Однако это не одинаковые операторы, хотя очевидно, что они родственны.

Листинг 9.1. Создание и использование ссылок

```
1:    // Листинг 9.1.
2:    // Пример использования ссылок
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:         int  intOne;
9:         int &rSomeRef = intOne;
10:
11:        intOne = 5;
12:        cout << "intOne: " << intOne << endl;
13:        cout << "rSomeRef: " << rSomeRef << endl;
14:
15:        rSomeRef = 7;
16:        cout << "intOne: " << intOne << endl;
17:        cout << "rSomeRef: " << rSomeRef << endl;
18:        return 0;
19:   }
```

Результат

```
intOne: 5
rSomeRef: 5
intOne: 7
rSomeRef: 7
```

Анализ

В строке 8 объявляется локальная целочисленная переменная `intOne`, а в строке 9 — ссылка `rSomeRef` на некоторое целое значение, инициализируемая адресом переменной `intOne`. Если объявить ссылку, но не инициализировать ее, будет сгенерирована ошибка компиляции. Ссылки, в отличие от указателя, необходимо инициализировать при объявлении.

В строке 11 переменной `intOne` присваивается значение `5`. В строках 12 и 13 выводятся на экран значения переменной `intOne` и ссылки `rSomeRef` — они, конечно же, оказываются одинаковыми.

В строке 15 ссылке `rSomeRef` присваивается значение `7`. Поскольку мы имеем дело со ссылкой, а она является псевдонимом для переменной `intOne`, то число `7` в действительности присваивается переменной `intOne`, что и подтверждается выводом на экран в строках 16 и 17.

Использование оператора адреса (&) при работе со ссылками

Если использовать ссылку для получения адреса, она вернет адрес своего адресата. В этом и состоит природа ссылок. Они являются псевдонимами для своих адресатов. Именно это свойство и демонстрирует листинг 9.2.

Листинг 9.2. Взятие адреса ссылки

```
1:    // Листинг 9.2.
2:    // Пример использования ссылок
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:        int  intOne;
9:        int &rSomeRef = intOne;
10:
11:       intOne = 5;
12:       cout << "intOne: " << intOne << endl;
13:       cout << "rSomeRef: " << rSomeRef << endl;
14:
15:       cout << "&intOne: "  << &intOne << endl;
16:       cout << "&rSomeRef: " << &rSomeRef << endl;
17:
18:   return 0;
19:   }
```

РЕЗУЛЬТАТ

```
intOne: 5
rSomeRef: 5
&intOne:  0x3500
&rSomeRef: 0x3500
```

ПРИМЕЧАНИЕ

Результаты работы программы на вашем компьютере могут отличаться от приведенных в последних двух строках.

АНАЛИЗ

И вновь-таки ссылка `rSomeRef` инициализируется адресом переменной `intOne`. На этот раз выводятся адреса двух переменных, и они оказываются идентичными. В языке C++ не предусмотрено предоставление доступа к адресу самой ссылки, поскольку в этом нет смысла. С таким же успехом для этого можно было бы использовать указатель или другую переменную. Ссылки инициализируются при создании, и они всегда действуют как синонимы для своих адресатов, даже в том случае, когда применяется оператор адреса.

Например, если у вас есть класс с именем `City`, вы могли бы объявить объект этого класса следующим образом:

```
City boston;
```

Затем можно объявить ссылку на некоторый объект класса `City` и инициализировать ее, используя данный конкретный объект:

```
City  &beanTown = boston;
```

Существует только один класс City; оба идентификатора ссылаются на один и тот же объект одного и того же класса. Любое действие, которое вы предпримите относительно ссылки beanTown, будет выполнено также и над объектом boston.

Обратите внимание на различие между символом & в строке 9 листинга 9.2, который объявляет ссылку rSomeRef на значение типа int, и символами & в строках 15 и 16, которые возвращают адреса целочисленной переменной intOne и ссылки rSomeRef.

Обычно для ссылки оператор адреса не используется. Мы просто используем ссылку вместо связанной с ней переменной, как показано в строке 13.

Ссылки нельзя переназначать

Даже опытных программистов, которые хорошо знают правило о том, что ссылки нельзя переназначать и что они всегда являются псевдонимами для своих адресатов, может ввести в заблуждение происходящее при попытке переназначить ссылку. То, что кажется переназначением, оказывается присвоением нового значения адресату. Этот факт иллюстрируется в листинге 9.3.

Листинг 9.3. Присвоение значения ссылке

```
1:      // Листинг 9.3.
2:      // Присвоение значения ссылке
3:
4:      #include <iostream.h>
5:
6:      int main()
7:      {
8:            int  intOne;
9:            int &rSomeRef = intOne;
10:
11:           intOne = 5;
12:           cout << "intOne:\ t" << intOne << endl;
13:           cout << "rSomeRef:\ t" << rSomeRef << endl;
14:           cout << "&intOne:\ t"  << &intOne << endl;
15:           cout << "&rSomeRef:\ t" << &rSomeRef << endl;
16:
17:           int intTwo = 8;
18:           rSomeRef = intTwo;  // не то что вы думаете!
19:           cout << "\ nintOne:\ t" << intOne << endl;
20:           cout << "intTwo:\ t" << intTwo << endl;
21:           cout << "rSomeRef:\ t" << rSomeRef << endl;
22:           cout << "&intOne:\ t"  << &intOne << endl;
23:           cout << "&intTwo:\ t"  << &intTwo << endl;
24:           cout << "&rSomeRef:\ t" << &rSomeRef << endl;
25:           return 0;
26:     }
```

Результат

```
intOne:         5
rSomeRef:       5
&intOne:        0x213e
&rSomeRef:      0x213e

intOne:         8
intTwo:         8
rSomeRef:       8
&intOne:        0x213e
&intTwo:        0x2130
&rSomeRef:      0x213e
```

Анализ

Вновь в строках 8 и 9 объявляются целочисленная переменная и ссылка на целое значение. В строке 11 целочисленной переменной присваивается значение 5, а в строках 12–15 выводятся значения переменной и ссылки, а также их адреса.

В строке 17 создается новая переменная `intTwo`, которая тут же инициализируется значением `8`. В строке 18 программист пытается переназначить ссылку `rSomeRef` так, чтобы она стала псевдонимом переменной `intTwo`, но этого не происходит. На самом же деле ссылка `rSomeRef` продолжает действовать как псевдоним переменной `intOne`, поэтому такое присвоение эквивалентно следующей операции:

```
intOne = intTwo;
```

Это кажется достаточно убедительным, особенно при выводе на экран значений переменной `intOne` и ссылки `rSomeRef` (строки 19–21): их значения совпадают со значением переменной `intTwo`. На самом деле при выводе на экран адресов в строках 22–24 вы видите, что ссылка `rSomeRef` продолжает ссылаться на переменную `intOne`, а не на переменную `intTwo`.

Рекомендуется	Не рекомендуется
Используйте ссылки для создания псевдонимов объектов. **Инициализируйте** ссылки при объявлении.	**Не пытайтесь** переназначить ссылку. **Не путайте** оператор адреса с оператором ссылки.

На что можно ссылаться

Ссылаться можно на любой объект, включая нестандартные (определенные пользователем) объекты. Обратите внимание, что ссылка создается на объект, а не на класс. Нельзя объявить ссылку таким образом:

```
int & rIntRef = int;     // неверно
```

Ссылку `rIntRef` нужно инициализировать, используя конкретную целочисленную переменную, например:

```
int howBig = 200;
int & rIntRef = howBig;
```

Точно так же нельзя инициализировать ссылку классом `CAT`:

```
CAT & rCatRef = CAT;    // неверно
```

Ссылку rCatRef нужно инициализировать, используя конкретный объект класса CAT:

```
CAT frisky;
CAT & rCatRef = frisky;
```

Ссылки на объекты используются точно так же, как сами объекты. Доступ к данным-членам и методам осуществляется с помощью обычного оператора доступа к членам класса (.), и, подобно встроенным типам, ссылка действует как псевдоним для объекта. Этот факт иллюстрируется в листинге 9.4.

Листинг 9.4. Ссылки на объекты класса

```
1:    // Листинг 9.4.
2:    // Ссылки на объекты класса
3:
4:    #include <iostream.h>
5:
6:    class SimpleCat
7:    {
8:       public:
9:          SimpleCat (int age, int weight);
10:         ~SimpleCat() { }
11:         int GetAge() {  return itsAge; }
12:         int GetWeight() {  return itsWeight; }
13:      private:
14:         int itsAge;
15:         int itsWeight;
16:   } ;
17:
18:   SimpleCat::SimpleCat(int age, int weight)
19:   {
20:        itsAge = age;
21:        itsWeight = weight;
22:   }
23:
24:   int main()
25:   {
26:        SimpleCat Frisky(5,3);
27:        SimpleCat & rCat = Frisky;
28:
29:        cout << "Frisky: ";
30:        cout << Frisky.GetAge() << " years old. \ n»;
31:        cout << "И Frisky весит: ";
32:        cout << rCat.GetWeight() << " kilograms. \ n»;
33:        return 0;
34:   }
```

Результат

```
Frisky: 5 years old.
И Frisky весит: 3 kilograms.
```

Анализ

В строке 26 объявляется переменная Frisky в качестве объекта класса SimpleCat. В строке 27 объявляется ссылка rCat на некоторый объект класса SimpleCat, и эта ссылка инициализируется с использованием уже объявленного объекта Frisky. В строках 30 и 32 вызываются методы доступа к членам класса SimpleCat, причем сначала это делается с помощью объекта класса SimpleCat (Frisky), а затем с помощью ссылки на объект класса SimpleCat (rCat). Обратите внимание, что результаты идентичны. Снова повторюсь: ссылка — это всего лишь псевдоним реального объекта.

Объявление ссылок

Ссылка объявляется путем указания типа данных, за которым следует оператор ссылки (&) и имя ссылки. Ссылки нужно инициализировать при объявлении.

Пример 1:

```
int hisAge;
int &rAge = hisAge;
```

Пример 2:

```
CAT boots;
CAT &rCatRef = boots;
```

Нулевые указатели и нулевые ссылки

Когда указатели не инициализированы или когда они освобождены, им следует присваивать нулевое значение (0). Это не касается ссылок. На самом деле ссылка не может быть нулевой, и программа, содержащая ссылку на нулевой объект, считается некорректной. Во время работы некорректной программы может случиться все что угодно. Она может внешне вести себя вполне пристойно, но при этом удалит все файлы на вашем диске или выкинет еще какой-нибудь фокус.

Большинство компиляторов могут поддерживать нулевой объект, ничего не сообщая по этому поводу до тех пор, пока вы не попытаетесь каким-то образом использовать этот объект. Однако не советую пользоваться поблажками компилятора, поскольку они могут дорого вам обойтись во время выполнения программы.

Передача аргументов функций как ссылок

На занятии 5 вы узнали о том, что функции имеют два ограничения: аргументы передаются как значения и теряют связь с исходными данными, а возвращать функция может только одно значение.

Преодолеть эти два ограничения можно путем передачи функции аргументов как ссылок. В языке C++ передача данных как ссылок осуществляется двумя способами: с помощью указателей и с помощью ссылок. Не запутайтесь в терминах: вы можете передать аргумент как ссылку, используя либо указатель, либо ссылку.

Несмотря на то что синтаксис использования указателя отличается от синтаксиса использования ссылки, конечный эффект одинаков. Вместо копии, создаваемой в пределах области видимости функции, в функцию передается реальный исходный объект.

На занятии 5 вы узнали, что параметры, передаваемые функции, помещаются в стек. Если функции передается значение как ссылка (с помощью либо указателей, либо ссылок), то в стек помещается не сам объект, а его адрес.

В действительности в некоторых компьютерах адрес хранится в специальном регистре, а в стек ничего не помещается. В любом случае компилятору известно, как добраться до исходного объекта, и при необходимости все изменения производятся прямо над объектом, а не над его временной копией.

При передаче объекта как ссылки функция может изменять объект, просто ссылаясь на него.

Вспомните, что в листинге 5.5 (см. Занятие 5) демонстрировалось, что после обращения к функции `swap()` значения в вызывающей функции не изменялись. Исключительно ради вашего удобства этот листинг воспроизведен здесь еще раз (листинг 9.5).

Листинг 9.5. Демонстрация передачи по значению

```
1:    // Листинг 9.5. Передача параметров как значений
2:
3:    #include <iostream.h>
4:
5:    void swap(int x, int y);
6:
7:    int main()
8:    {
9:      int x = 5, y = 10;
10:
11:     cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
12:     swap(x,y);
13:     cout << "Main. After swap, x: " << x << " y: " << y << "\n";
14:        return 0;
15:   }
16:
17:   void swap (int x, int y)
18:   {
19:     int temp;
20:
21:     cout << "Swap. Before swap, x: " << x << " y: " << y << "\n";
22:
23:     temp = x;
24:     x = y;
25:     y = temp;
26:
27:     cout << "Swap. After swap, x: " << x << " y: " << y << "\n";
28:
29:   }
```

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

Эта программа инициализирует две переменные в функции main(), а затем передает их функции swap(), которая, казалось бы, должна поменять их значения. Однако после повторной проверки этих переменных в функции main() оказывается, что они не изменились.

Проблема здесь в том, что переменные x и y передаются функции swap() по значению, т.е. в данном случае локальные копии этих переменных создаются прямо в функции. Чтобы решить проблему, нужно передать значения переменных x и y как ссылки.

В языке C++ существует два способа решения этой проблемы: можно сделать параметры функции swap() указателями на исходные значения или передать ссылки на исходные значения.

Передача указателей в функцию swap()

Передавая указатель, мы передаем адрес объекта, а следовательно, функция может манипулировать значением, находящимся по этому переданному адресу. Чтобы заставить функцию swap() изменить реальные значения с помощью указателей, ее нужно объявить так, чтобы она принимала два указателя на целые значения. Затем путем разыменования указателей значения переменных x и y будут на самом деле меняться местами. Эта идея демонстрируется в листинге 9.6.

Листинг 9.6. Передача аргументов как ссылок с помощью указателей

```
1:    // Листинг 9.6. Пример передачи аргументов как ссылок
2:
3:    #include <iostream.h>
4:
5:    void swap(int *x, int *y);
6:
7:    int main()
8:    {
9:      int x = 5, y = 10;
10:
11:      cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
12:      swap(&x,&y);
13:      cout << "Main. After swap, x: " << x << " y: " << y << "\n";
14:   return 0;
15:   }
16:
17:   void swap (int *px, int *py)
18:   {
19:     int temp;
20:
21:     cout << "Swap. Before swap, *px: " << *px << " *py: " << *py << "\n";
22:
23:     temp = *px;
24:     *px = *py;
25:     *py = temp;
26:
27:     cout << "Swap. After swap, *px: " << *px << " *py: " << *py << "\n";
28:
29:   }
```

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, *px: 5 *py: 10
Swap. After swap, *px: 10 *py: 5
Main. After swap, x: 10 y: 5
```

Получилось! В строке 5 изменен прототип функции `swap()` где в качестве параметров объявляются указатели на значения типа `int`, а не сами переменные типа `int`. При вызове в строке 12 функции `swap()` в качестве параметров передаются адреса переменных `x` и `y`.

В строке 19 объявляется локальная для функции `swap()` переменная `temp`, которой вовсе не обязательно быть указателем: она будет просто хранить значение `*px` (т.е. значение переменной `x` в вызывающей функции) в течение жизни функции. После окончания работы функции переменная `temp` больше не нужна.

В строке 23 переменной `temp` присваивается значение, хранящееся по адресу `px`. В строке 24 значение, хранящееся по адресу `px`, записывается в ячейку с адресом `py`. В строке 25 значение, оставленное на время в переменной `temp` (т.е. исходное значение, хранящееся по адресу `px`), помещается в ячейку с адресом `py`.

В результате значения переменных вызывающей функции, адреса которых были переданы функции `swap()`, успешно поменялись местами.

Передача ссылок в функцию swap()

Приведенная выше программа, конечно же, работает, но синтаксис функции `swap()` несколько громоздок. Во-первых, необходимость неоднократно разыменовывать указатели внутри функции `swap()` создает благоприятную почву для возникновения ошибок, кроме того, операции разыменовывания трудно читаются. Во-вторых, необходимость передавать адреса переменных из вызывающей функции нарушает принцип инкапсуляции выполнения функции `swap()`.

Суть программирования на языке C++ состоит в сокрытии от пользователей функции деталей ее выполнения. Передача параметров с помощью указателей перекладывает ответственность за получение адресов переменных на вызывающую функцию, вместо того чтобы сделать это в теле вызываемой функции. Другое решение той же задачи предлагается в листинге 9.7, в котором показана работа функции `swap()` с использованием ссылок.

Листинг 9.7. Та же функция `swap()`, но с использованием ссылок

```
1:      // Листинг 9.7. Пример передачи аргументов по
2:      // ссылок с помощью ссылок!
3:
4:      #include <iostream.h>
5:
6:      void swap(int &x, int &y);
7:
8:      int main()
9:      {
10:         int x = 5, y = 10;
11:
12:         cout << "Main. Before swap, x: " << x << " y: " << y << "\ n";
13:         swap(x,y);
14:         cout << "Main. After swap, x: " << x << " y: " << y << "\ n";
```

```
15:          return 0;
16:      }
17:
18:      void swap (int &rx, int &ry)
19:      {
20:        int temp;
21:
22:        cout << "Swap. Before swap, rx: " << rx << " ry: " << ry << "\ n";
23:
24:        temp = rx;
25:        rx = ry;
26:        ry = temp;
27:
28:        cout << "Swap. After swap, rx: " << rx << " ry: " << ry << "\ n";
29:
30:       }
```

РЕЗУЛЬТАТ

```
Main. Before swap, x:5 y: 10
Swap. Before swap, rx:5 ry:10
Swap. After swap, rx:10 ry:5
Main. After swap, x:10, y:5
```

АНАЛИЗ

Точно так же, как и в примере с указателями, в строке 10 объявляются две переменные, а их значения выводятся на экран в строке 12. В строке 13 вызывается функция `swap()`, но обратите внимание на то, что ей передаются именно значения `x` и `y`, а не их адреса. Вызывающая функция просто передает свои переменные.

После вызова функции `swap()` выполнение программы переходит к строке 18, в которой эти переменные идентифицируются как ссылки. Их значения выводятся на экран в строке 22, но заметьте, что для этого не требуется никаких специальных операторов, поскольку мы имеем дело с псевдонимами для исходных значений и используем их в этом качестве.

В строках 24–26 выполняется обмен значений, после чего они выводятся на экран в строке 28. Управление программой вновь возвращается в вызывающую функцию, и в строке 14 эти значения опять выводятся на экран, но уже в функции `main()`. Поскольку параметры для функции `swap()` объявлены как ссылки, то и переменные из функции `main()` передаются как ссылки, следовательно, они также изменяются и в функции `main()`.

Таким образом, благодаря использованию ссылок функция приобретает новую возможность изменять исходные данные в вызывающей функции, хотя при этом сам вызов функции ничем не отличается от обычного!

Представления о заголовках функций и прототипах

В листинге 9.6 показана функция `swap()`, использующая в качестве аргументов указатели, а в листинге 9.7 — та же функция, но с использованием ссылок. Использовать функцию, которая принимает в качестве параметров ссылки, легче, да и в программе такая функция проще читается, но как вызывающей функции узнать о том, каким способом переданы параметры — по ссылке или по значению? Будучи клиентом (или пользователем) функции `swap()`, программист должен быть уверен в том, что функция `swap()` на самом деле изменит параметры.

Самое время вспомнить о прототипе функции, для которого в данном контексте нашлось еще одно применение. Изучив параметры, объявленные в прототипе, который обычно располагается в файле заголовка вместе со всеми другими прототипами, программист будет точно знать, что значения, принимаемые функцией `swap()`, передаются как ссылки, следовательно, обмен значений произойдет должным образом.

Если бы функция `swap()` была функцией-членом класса, то объявление этого класса, также расположенное в файле заголовка, обязательно содержало бы эту информацию.

В языке C++ клиенты классов и функций всю необходимую информацию черпают из файлов заголовков. Этот файл выполняет роль интерфейса с классом или функцией, действительная реализация которых скрыта от клиента. Это позволяет программисту сосредоточиться на собственных проблемах и использовать класс или функцию, не вникая в детали их работы.

Когда Колонел Джон Роблинг (Colonel John Roebling) проектировал свой Бруклинский мост (Brooklyn Bridge), он интересовался деталями процесса литья и изготовления проводов. Он глубоко вникал в подробности механических и химических процессов, которые требовалось обеспечить для создания необходимых материалов. Но в наши дни инженеры более эффективно используют свое рабочее время, доверяя информации о строительных материалах и не интересуясь подробностями их изготовления.

В этом и состоит цель языка C++ — позволить программистам полагаться на описанные классы и функции, не вникая во внутренние механизмы их действия. Эти составные части можно собрать в одну программу, подобно тому как строители из отдельных блоков, проводов, труб, кирпичей и других элементов создают дома и мосты.

Подобно инженеру, изучающему технические характеристики трубы, чтобы узнать ее пропускную способность, объем, размеры арматуры и пр., программист читает интерфейсы функций и классов, чтобы определить, какой сервис предоставляет данный компонент, какие параметры он принимает и какие значения возвращает.

Возвращение нескольких значений

Как упоминалось выше, функции могут возвращать только одно значение. Что же делать, если нужно получить от функции сразу два значения? Один путь решения этой проблемы — передача функции двух объектов как ссылок. В ходе выполнения функция присвоит этим объектам нужные значения. Факт передачи объектов как ссылок, позволяющий функции изменить исходные объекты, равносилен разрешению данной функции возвратить два значения. В этом случае мы обходимся без возвращаемого значения, которое (зачем же пропадать добру) можно использовать для сообщения об ошибках.

И вновь одинакового результата можно достичь, используя как ссылки, так и указатели. В листинге 9.8 показана функция, которая возвращает три значения: два в виде параметров-указателей и одно в виде возвращаемого значения функции.

Листинг 9.8. Возвращение значений с помощью указателей

```
1:    // Листинг 9.8.
2:    // Возвращение нескольких значений из функции с помощью указателей
3:
4:    #include <iostream.h>
5:    int
6:    short Factor(int n, int* pSquared, int* pCubed);
7:
8:    int main()
9:    {
```

```
10:     int number, squared, cubed;
11:     short error;
12:
13:     cout << "Enter a number (0 - 20): ";
14:     cin >> number;
15:
16:     error = Factor(number, &squared, &cubed);
17:
18:     if (!error)
19:     {
20:         cout << "number: " << number << "\ n";
21:         cout << "square: " << squared << "\ n";
22:         cout << "cubed: "  << cubed   << "\ n";
23:     }
24:     else
25:         cout << "Error encountered!!\ n";
26:     return 0;
27: }
28:
29: short Factor(int n, int *pSquared, int *pCubed)
30: {
31: short Value = 0;
32:     if (n > 20)
33:        Value = 1;
34:     else
35:     {
36:         *pSquared = n*n;
37:         *pCubed = n*n*n;
38:         Value = 0;
39:     }
40:     return Value;
41: }
```

РЕЗУЛЬТАТ

```
Enter a number (0-20): 3
number: 3
square: 9
cubed: 27
```

АНАЛИЗ

В строке 10 переменные `number`, `squared` и `cubed` определяются с использованием типа `int`. Переменной `number` присваивается значение, введенное пользователем. Это значение, а также адреса переменных `squared` и `cubed` передаются функции `Factor()` в виде параметров.

В функции `Factor()` анализируется первый параметр, который передается как значение. Если он больше 20 (максимальное значение, которое может обработать эта функция), то возвращаемое значение `Value` устанавливается равным единице, что служит признаком ошибки. Обратите внимание на то, что возвращаемое значение из функции `Factor()` может принимать либо значение 1, либо 0, являющееся признаком того, что все прошло нормально, а также заметьте, что функция возвращает это значение лишь в строке 40.

Итак, искомые значения (квадрат и куб заданного числа) возвращаются в вызывающую функцию не путем использования механизма возврата значений, а за счет изменения значений переменных, указатели которых переданы в функцию.

В строках 36 и 37 посредством указателей переменным в функции `main()` присваиваются возвращаемые значения. В строке 38 переменной `Value` присваивается значение возврата, означающее успешное завершение работы функции. В строке 40 это значение `Value` возвращается вызывающей функции.

Эту программу можно слегка усовершенствовать, дополнив ее следующим объявлением:

```
enum ERROR_VALUE { SUCCESS, FAILURE} ;
```

Затем вместо возврата значений `0` или `1` эта программа сможет возвращать `SUCCESS` или `FAILURE`.

Возвращение значений с помощью ссылок

Несмотря на то что листинг 9.8 прекрасно работает, его можно упростить как для чтения, так и в эксплуатации, если вместо указателей использовать ссылки. В листинге 9.9 показана та же самая программа, но вместо указателей в качестве параметров функции в ней используются ссылки, а также добавлено упомянутое выше перечисление `ERROR`.

Листинг 9.9. Возвращение значений с помощью ссылок

```
1:    // Листинг 9.9.
2:    // Возвращение нескольких значений из функции
3:    // с помощью ссылок
4:
5:    #include <iostream.h>
6:
7:    typedef unsigned short USHORT;
8:    enum ERR_CODE {  SUCCESS, ERROR } ;
9:
10:   ERR_CODE Factor(USHORT, USHORT&, USHORT&);
11:
12:   int main()
13:   {
14:         USHORT number, squared, cubed;
15:         ERR_CODE result;
16:
17:         cout << "Enter a number (0 - 20): ";
18:         cin >> number;
19:
20:         result = Factor(number, squared, cubed);
21:
22:         if (result == SUCCESS)
23:         {
24:               cout << "number: " << number << "\ n";
25:               cout << "square: " << squared << "\ n";
26:               cout << "cubed: "  << cubed   << "\ n";
```

```
27:          }
28:          else
29:          cout << "Error encountered!!\ n";
30:     return 0;
31:     }
32:
33:     ERR_CODE Factor(USHORT n, USHORT &rSquared, USHORT &rCubed)
34:     {
35:          if (n > 20)
36:               return ERROR;    // simple error code
37:          else
38:          {
39:               rSquared = n*n;
40:               rCubed = n*n*n;
41:               return SUCCESS;
42:          }
43:     }
```

Вывод

```
Enter a number (0 - 20): 3
number: 3
square: 9
cubed: 27
```

Анализ Листинг 9.9 идентичен листингу 9.8 с двумя исключениями. Перечисление `ERR_CODE` делает сообщение об ошибке более явным (см. строки 36 и 41), как, впрочем, и его обработку (строка 22).

Однако более существенные изменения коснулись функции `Factor()`. Теперь эта функция объявляется для принятия не указателей, а ссылок на переменные `squared` и `cubed`, что делает манипуляции над этими параметрами гораздо проще и легче для понимания.

Передача ссылок на переменные как средство повышения эффективности

При каждой передаче объекта в функцию как значения создается копия этого объекта. При каждом возврате объекта из функции создается еще одна копия.

На занятии 5 вы узнали о том, что эти объекты копируются в стек и на этот процесс расходуется время и память. Для таких маленьких объектов, как базовые целочисленные значения, цена этих расходов незначительна.

Однако для больших объектов, создаваемых пользователем, расходы ресурсов существенно возрастают. Размер такого объекта в стеке представляет собой сумму всех его переменных-членов. Причем каждая переменная-член может быть, в свою очередь, подобным объектом, поэтому передача такой массивной структуры путем копирования в стек может оказаться весьма дорогим удовольствием как по времени, так и по занимаемой памяти.

Кроме того, существуют и другие расходы. При создании временных копий объектов классов для этих целей компилятор вызывает специальный конструктор-копировщик. На следующем занятии вы узнаете, как работают конструкторы-копировщики и

как можно создать собственный конструктор-копировщик, но пока достаточно знать, что конструктор-копировщик вызывается каждый раз, когда в стек помещается временная копия объекта.

При разрушении временного объекта, которое происходит при возврате из функции, вызывается деструктор объекта. Если объект возвращается функцией как значение, копия этого объекта должна быть сначала создана, а затем разрушена.

При работе с большими объектами эти вызовы конструктора и деструктора могут оказать слишком ощутимое влияние на скорость работы программы и использование памяти компьютера. Для иллюстрации этой идеи в листинге 9.10 создается пользовательский объект `SimpleCat`. Реальный объект имел бы размеры побольше и обошелся бы дороже, но и этого примера вполне достаточно, чтобы показать, насколько часто вызываются конструктор-копировщик и деструктор.

Итак, в листинге 9.10 создается объект `SimpleCat`, после чего вызываются две функции. Первая функция принимает объект `Cat` как значение, а затем возвращает его как значение. Вторая же функция принимает указатель на объект, а не сам объект, и возвращает указатель на объект.

Листинг 9.10. Передача объектов как ссылок с помощью указателей

```
1:  // Листинг 9.10.
2:  // Передача указателей на объекты
3:
4:  #include <iostream.h>
5:
6:  class SimpleCat
7:  {
8:  public:
9:          SimpleCat ();           // конструктор
10:         SimpleCat(SimpleCat&); // конструктор-копировщик
11:         ~SimpleCat();          // деструктор
12: } ;
13:
14:  SimpleCat::SimpleCat()
15:  {
16:         cout << "Simple Cat Constructor...\n";
17:  }
18:
19:  SimpleCat::SimpleCat(SimpleCat&)
20:  {
21:         cout << "Simple Cat Copy Constructor...\ n";
22:  }
23:
24:  SimpleCat::~SimpleCat()
25:  {
26:         cout << "Simple Cat Destructor...\ n";
27:  }
28:
29:  SimpleCat FunctionOne (SimpleCat theCat);
30:  SimpleCat* FunctionTwo (SimpleCat *theCat);
31:
32:  int main()
```

```
33:  {
34:        cout << "Making a cat...\ n";
35:        SimpleCat Frisky;
36:        cout << "Calling FunctionOne...\ n";
37:        FunctionOne(Frisky);
38:        cout << "Calling FunctionTwo...\ n";
39:        FunctionTwo(&Frisky);
40:    return 0;
41:  }
42:
43:  // Функция FunctionOne, передача как значения
44:  SimpleCat FunctionOne(SimpleCat theCat)
45:  {
46:                  cout << "Function One. Returning...\ n";
47:                  return theCat;
48:  }
49:
50:  // Функция FunctionTwo, передача как ссылки
51:  SimpleCat* FunctionTwo (SimpleCat  *theCat)
52:  {
53:                  cout << "Function Two. Returning...\ n";
54:                  return theCat;
55:  }
```

РЕЗУЛЬТАТ

```
Making a cat...
Simple Cat Constructor...
Calling FunctionOne...
Simple Cat Copy Constructor...
Function One. Returning...
Simple Cat Copy Constructor...
Simple Cat Destructor...
Simple Cat Destructor...
Calling FunctionTwo...
Function Two. Returning...
Simple Cat Destructor...
```

Номера строк не выводятся. Мы добавили их для удобства проведения анализа программы.

АНАЛИЗ

В строках 6–12 объявляется весьма упрощенный класс `SimpleCat`. Конструктор, конструктор-копировщик и деструктор — все компоненты класса выводят на экран свои информативные сообщения, чтобы было точно известно, когда они вызываются.

В строке 34 функция `main()` выводит свое первое сообщение, которое является первым и в результатах работы программы. В строке 35 создается экземпляр объекта класса `SimpleCat`. Это приводит к вызову конструктора, что подтверждает сообщение, выводимое этим конструктором (строка 2 в результатах работы программы).

В строке 36 функция main() "докладывает" о вызове функции FunctionOne, которая выводит свое сообщение (строка 3 в результатах работы программы). Поскольку функция FunctionOne() вызывается с передачей объекта класса SimpleCat по значению, в стек помещается копия объекта SimpleCat как локального для вызываемой функции. Это приводит к вызову конструктора копии, который "вносит свою лепту" в результаты работы программы (сообщение в строке 4).

Выполнение программы переходит к строке 46, которая принадлежит телу вызванной функции, выводящей свое информативное сообщение (строка 5 в результатах работы программы). Затем эта функция возвращает управление программой вызывающей функции, и объект класса SimpleCat вновь возвращается как значение. При этом создается еще одна копия объекта за счет вызова конструктора-копировщика и, как следствие, на экран выводится очередное сообщение (строка 6 в результатах работы программы).

Значение, возвращаемое из функции FunctionOne(), не присваивается ни одному объекту, поэтому ресурсы, затраченные на создание временного объекта при реализации механизма возврата, просто выброшены на ветер, как и ресурсы, затраченные на его удаление с помощью деструктора, который заявил о себе в строке 7 в результатах работы программы. Поскольку функция FunctionOne() завершена, локальная копия объекта выходит за область видимости и разрушается, вызывая деструктор и генерируя тем самым сообщение, показанное в строке 8 в результатах работы программы.

Управление программой возвращается к функции main(), после чего вызывается функция FunctionTwo(), но на этот раз параметр передается как ссылка. При этом никакой копии объекта не создается, поэтому отсутствует и сообщение от конструктора-копировщика. В функции FunctionTwo() выводится сообщение, занимающее строку 10 в результатах работы программы, а затем выполняется возвращение объекта класса SimpleCat (снова как ссылки), поэтому нет никаких обращений к конструктору и деструктору.

Наконец программа завершается и объект Frisky выходит за область видимости, генерируя последнее обращение к деструктору, выводящему свое сообщение (строка 11 в результатах работы программы).

Проанализировав работу этой программы, можно сделать вывод, что при вызове функции FunctionOne() делается два обращения к конструктору копии и два обращения к деструктору, поскольку объект в эту функцию передается как значение, в то время как при вызове функции FunctionTwo() подобных обращений не делается.

Передача константного указателя

Несмотря на то что передача указателя функции FunctionTwo() более эффективна, чем передача по значению, она таит в себе немалую опасность. При вызове функции FunctionTwo() совершенно не имелось в виду, что разрешается изменять передаваемый ей объект класса SimpleCat, задаваемый в виде адреса объекта SimpleCat. Такой способ передачи открывает объект для изменений и аннулирует защиту, обеспечиваемую при передаче объекта как значения.

Передачу объектов как значений можно сравнить с передачей музею фотографии шедевра вместо самого шедевра. Если какой-нибудь хулиган испортит фотографию, то никакого вреда при этом оригиналу нанесено не будет. А передачу объекта как ссылки можно сравнить с передачей музею своего домашнего адреса и приглашением гостей посетить ваш дом и взглянуть в вашем присутствии на драгоценный шедевр.

Решить проблему можно, передав в функцию указатель на константный объект класса SimpleCat. В этом случае к данному объекту могут применяться только константные методы, не имеющие прав на изменение объекта SimpleCat. Эта идея демонстрируется в листинге 9.11.

Листинг 9.11. Передача константных указателей

```
1:      // Листинг 9.11.
2:      // Передача константных указателей на объекты
3:
4:        #include <iostream.h>
5:
6:        class SimpleCat
7:        {
8:        public:
9:                SimpleCat();
10:               SimpleCat(SimpleCat&);
11:               ~SimpleCat();
12:
13:               int GetAge() const {  return itsAge; }
14:               void SetAge(int age) {  itsAge = age; }
15:
16:       private:
17:               int itsAge;
18:           } ;
19:
20:           SimpleCat::SimpleCat()
21:           {
22:                  cout << "Simple Cat Constructor...\n";
23:                  itsAge = 1;
24:           }
25:
26:           SimpleCat::SimpleCat(SimpleCat&)
27:           {
28:                  cout << "Simple Cat Copy Constructor...\n";
29:           }
30:
31:           SimpleCat::~SimpleCat()
32:           {
33:                  cout << "Simple Cat Destructor...\n";
34:           }
35:
36:     const SimpleCat * const FunctionTwo (const SimpleCat * const theCat);
37:
38:           int main()
39:           {
40:                  cout << "Making a cat...\ n";
41:                  SimpleCat Frisky;
42:                  cout << "Frisky is " ;
43                   cout << Frisky.GetAge();
44:                  cout << " years old\ n";
45:                  int age = 5;
46:                  Frisky.SetAge(age);
47:                  cout << "Frisky is " ;
48                   cout << Frisky.GetAge();
```

```
49:                    cout << " years old\ n";
50:                    cout << "Calling FunctionTwo...\ n";
51:                    FunctionTwo(&Frisky);
52:                    cout << "Frisky is " ;
53:                    cout << Frisky.GetAge();
54:                    cout << " years _old\ n";
55:     return 0;
56:           }
57:
58:    // functionTwo, passes a const pointer
59:    const SimpleCat * const FunctionTwo (const SimpleCat * const theCat)
60:    {
61:             cout << "Function Two. Returning...\ n";
62:             cout << "Frisky is now " << theCat->GetAge();
63:             cout << " years old \ n";
64:             // theCat->SetAge(8);   const!
65:             return theCat;
66:            }
```

РЕЗУЛЬТАТ

```
Making a cat...
Simple Cat constructor...
Frisky is 1 years old
Frisky is 5 years old
Calling FunctionTwo...
FunctionTwo. Returning...
Frisky is now 5 years old
Frisky is 5 years old
Simple Cat Destructor...
```

АНАЛИЗ

В класс `SimpleCat` были добавлены две функции доступа к данным: метод `GetAge()` (строка 13), который является константной функцией, и метод `SetAge()` (строка 14), который не является константным. В этот класс была также добавлена переменная-член `itsAge` (строка 17).

Конструктор, конструктор-копировщик и деструктор по-прежнему определены для вывода на экран своих сообщений. Однако конструктор-копировщик ни разу не вызывался, поскольку объект был передан как ссылка и поэтому никаких копий объекта не создавалось. В строке 41 был создан объект со значением возраста, заданным по умолчанию. Это значение выводится на экран в строке 42.

В строке 46 переменная `itsAge` устанавливается с помощью метода доступа `SetAge`, а результат этой установки выводится на экран в строке 47. В этой программе функция `FunctionOne` не используется, но вызывается функция `FunctionTwo()`, которая слегка изменена. Ее объявление занимает строку 36. На этот раз и параметр, и значение возврата объявляются как константные указатели на константные объекты.

Поскольку и параметр, и возвращаемое значение передаются как ссылки, никаких копий не создается и конструктор-копировщик не вызывается. Однако указатель в функции `FunctionTwo()` теперь является константным, следовательно, к нему не может применяться неконстантный метод `SetAge()`. Если обращение к методу `SetAge()` в строке 64 не было бы закомментировано, программа не прошла бы этап компиляции.

Обратите внимание, что объект, создаваемый в функции main(), не является константным и объект Frisky может вызвать метод SetAge(). Адрес этого обычного объекта передается функции FunctionTwo(), но, поскольку в объявлении функции FunctionTwo() заявлено, что передаваемый указатель должен быть *константным* указателем на *константный* объект, с этим объектом функция обращается так, как если бы он был константным!

Ссылки в качестве альтернативы

При выполнении программы, показанной в листинге 9.11, устранена необходимость создания временных копий, что сокращает число обращений к конструктору и деструктору класса, в результате чего программа работает более эффективно. В данном примере использовался константный указатель на константный объект, что предотвращало возможность изменения объекта функцией. Однако по-прежнему имеет место некоторая громоздкость синтаксиса, свойственная передаче в функции указателей.

Поскольку, как вам известно, объект никогда не бывает нулевым, внутреннее содержание функции упростилось бы, если бы ей вместо указателя передавалась ссылка. Подтверждение этим словам вы найдете в листинге 9.12.

Листинг 9.12. Передача ссылок на объекты

```
1:   // Листинг 9.12.
2:   // Передача ссылок на объекты
3:
4:   #include <iostream.h>
5:
6:   class SimpleCat
7:   {
8:   public:
9:           SimpleCat();
10:          SimpleCat(SimpleCat&);
11:          ~SimpleCat();
12:
13:          int GetAge() const {  return itsAge; }
14:          void SetAge(int age) {  itsAge = age; }
15:
16:   private:
17:          int itsAge;
18:     } ;
19:
20:     SimpleCat::SimpleCat()
21:     {
22:           cout << "Simple Cat Constructor...\n";
23:           itsAge = 1;
24:     }
25:
26:     SimpleCat::SimpleCat(SimpleCat&)
27:     {
28:           cout << "Simple Cat Copy Constructor...\n";
```

```
29:     }
30:
31:     SimpleCat::~SimpleCat()
32:     {
33:           cout << "Simple Cat Destructor...\n";
34:     }
35:
36:     const     SimpleCat & FunctionTwo (const SimpleCat & theCat);
37:
38:     int main()
39:     {
40:           cout << "Making a cat...\ n";
41:           SimpleCat Frisky;
42:           cout << "Frisky is " << Frisky.GetAge() << " years old\ n";
43:           int age = 5;
44:           Frisky.SetAge(age);
45:           cout << "Frisky is " << Frisky.GetAge() << " years old\ n";
46:           cout << "Calling FunctionTwo...\ n";
47:           FunctionTwo(Frisky);
48:           cout << "Frisky is " << Frisky.GetAge() << " years old\ n";
49:    return 0;
50:     }
51:
52:     // functionTwo, passes a ref to a const object
53:     const SimpleCat & FunctionTwo (const SimpleCat & theCat)
54:     {
55:                    cout << "Function Two. Returning...\ n";
56:                    cout << "Frisky is now " << theCat.GetAge();
57:                    cout << " years old \ n";
58:                    // theCat.SetAge(8);   const!
59:                    return theCat;
60:     }
```

Результат:

```
Making a cat...
Simple Cat constructor...
Frisky is 1 years old
Frisky is 5 years old
Calling FunctionTwo...
FunctionTwo. Returning...
Frisky is now 5 years old
Frisky is 5 years old
Simple Cat Destructor...
```

Анализ: Результат работы этой программы идентичен результату, показанному после листинга 9.11. Единственное существенное изменение — функция `FunctionTwo()` теперь принимает и возвращает ссылки на константный объект. И вновь-таки работа со ссылками несколько проще, чем работа с указателями, хотя при этом достигается та же экономия средств и эффективность выполнения, а также обеспечивается надежность за счет использования спецификатора `const`.

Константные ссылки

Программисты, работающие с языком C++, обычно не видят разницы между константной ссылкой на объект `SimpleCat` и ссылкой на константный объект `SimpleCat`. Сами ссылки нельзя переназначать, чтобы они ссылались на другой объект, поэтому они всегда константны. Если к ссылке применено ключевое слово `const`, то это делает константным объект, с которым связана ссылка.

Когда лучше использовать ссылки, а когда – указатели

Опытные программисты безоговорочно отдают предпочтение ссылкам, а не указателям. Ссылки проще использовать, и они лучше справляются с задачей сокрытия информации, как вы видели в предыдущем примере.

Но ссылки нельзя переназначать. Если же вам нужно сначала указывать на один объект, а затем на другой, придется использовать указатель. Ссылки не могут быть нулевыми, поэтому, если существует хоть какая-нибудь вероятность того, что рассматриваемый объект может быть нулевым, вам нельзя использовать ссылку. В этом случае необходимо использовать указатель.

В качестве примера рассмотрим оператор `new`. Если оператор `new` не сможет выделить память для нового объекта, он возвратит нулевой указатель. А поскольку ссылка не может быть нулевой, вы не должны инициализировать ссылку на эту память до тех пор, пока не проверите, что она не нулевая. В следующем примере показано, как это сделать:

```
int *pInt = new int;
if (pInt != NULL)
int &rInt = *pInt;
```

В этом примере объявляется указатель `pInt` на значение типа `int`, который инициализируется областью памяти, возвращаемой оператором `new`. Адрес этой области памяти (в указателе `pInt`) тестируется, и, если он не равен значению `null`, указатель `pInt` разыменовывается. Результат разыменования переменной типа `int` представляет собой объект типа `int`, и ссылка `rInt` инициализируется этим объектом. Следовательно, ссылка `rInt` становится псевдонимом для переменной типа `int`, возвращаемой оператором `new`.

Рекомендуется

Передавайте функциям параметры как ссылке везде, где это возможно.

Обеспечивайте возврат значений как ссылок везде, где это возможно.

Используйте спецификатор `const` для защиты ссылок и указателей везде, где это возможно.

Не рекомендуется

Не используйте указатели, если вместо них можно использовать ссылки.

Не возвращайте ссылки на локальные объекты.

Коктейль из ссылок и указателей

Не будет ошибкой в списке параметров одной функции объявить как указатели, так и ссылки, а также объекты, передаваемые как значения, например:

```
CAT * SomeFunction (Person &theOwner, House *theHouse, int age);
```

Это объявление означает, что функция SomeFunction принимает три параметра. Первый является ссылкой на объект типа Person, второй — указателем на объект типа House, а третий — целочисленным значением. Сама же функция возвращает указатель на объект класса CAT.

Следует также отметить, что при объявлении соответствующих переменных можно использовать разные стили размещения операторов ссылки (&) и косвенного обращения (*). Вполне законной будет любая из следующих записей:

```
1: CAT&  rFrisky;
2: CAT & rFrisky;
3: CAT  &rFrisky;]
```

ПРИМЕЧАНИЕ

Символы пробелов в программах на языке C++ полностью игнорируются, поэтому везде, где вы видите пробел, можно ставить несколько пробелов, символов табуляции или символов разрывов строк.

Оставив в покое вопросы свободного волеизъявления, попробуем разобраться в том, какой вариант все же лучше других. Как ни странно, можно найти аргументы в защиту каждого из трех вариантов.

Аргумент в защиту первого варианта состоит в следующем. rFrisky — это переменная с именем rFrisky, тип которой можно определить как ссылку на объект класса CAT. Поэтому вполне логично, чтобы оператор & стоял рядом с типом.

Однако есть и контраргумент. CAT — это тип. Оператор & является частью объявления, которое включает имя переменной и амперсант. Но следует отметить, что слияние вместе символа & и имени типа CAT может привести к возникновению следующей ошибки:

```
CAT&  rFrisky, rBoots;
```

Поверхностный анализ этой строки может натолкнуть на мысль, что как переменная rFrisky, так и переменная rBoots являются ссылками на объекты класса CAT. Однако это не так. На самом деле это объявление означает, что rFrisky является ссылкой на объект класса CAT, а rBoots (несмотря на свое имя с характерным префиксом) — не ссылка, а обыкновенная переменная типа CAT. Поэтому последнее объявление следует переписать по-другому:

```
CAT &rFrisky, rBoots;
```

В ответ на это возражение стоит порекомендовать, чтобы объявления ссылок и обычных переменных никогда не смешивались в одной строке. Вот правильный вариант той же записи:

```
CAT& rFrisky;
CAT Boots;
```

ПРИМЕЧАНИЕ

Наконец, многие программисты не обращают внимания на приведенные аргументы и, считая, что истина находится посередине, выбирают средний вариант (средний, кстати, в двух смыслах), который иллюстрируется случаем 2:

```
2: CAT & rFrisky;
```

Безусловно, все сказанное до сих пор об операторе ссылки (&) относится в равной степени и к оператору косвенного обращения (*). Выберите стиль, который вам подходит, и придерживайтесь его на протяжении всей программы, ведь ясность текста программы — одна из основных составляющих успеха.
Многие программисты при объявлении ссылок и указателей предпочитают придерживаться двух соглашений.

1. Размещать амперсант или звездочку посередине, окаймляя этот символ пробелами с двух сторон.
2. Никогда не объявлять ссылки, указатели и переменные в одной и той же строке программы.

Не возвращайте ссылку на объект, который находится вне области видимости!

Научившись передавать аргументы как ссылки на объекты, программисты порой теряют чувство реальности. Не стоит забывать, что все хорошо в меру. Помните, что ссылка всегда служит псевдонимом некоторого объекта. При передаче ссылки в функцию или из нее не забудьте задать себе вопрос: "Что представляет собой объект, псевдонимом которого я манипулирую, и будет ли он существовать в момент его использования?"

В листинге 9.13 показан пример возможной ошибки, когда функция возвращает ссылку на объект, которого уже не существует.

Листинг 9.13. Возвращение ссылки на несуществующий объект

```
1:      // Листинг 9.13.
2:      // Возвращение ссылки на объект,
3:      // которого больше не существует
4:
5:      #include <iostream.h>
6:
7:      class SimpleCat
8:      {
9:      public:
10:           SimpleCat (int age, int weight);
11:           ~SimpleCat() { }
12:           int GetAge() {  return itsAge; }
13:           int GetWeight() {  return itsWeight; }
14:     private:
```

```
15:          int itsAge;
16:          int itsWeight;
17:      } ;
18:
19:      SimpleCat::SimpleCat(int age, int weight)
20:      {
21:        itsAge = age;
22:        itsWeight = weight;
23:      }
24:
25:      SimpleCat &TheFunction();
26:
27:      int main()
28:      {
29:           SimpleCat &rCat = TheFunction();
30:           int age = rCat.GetAge();
31:           cout << "rCat " << age << " years old!\n";
32:     return 0;
33:      }
34:
35:      SimpleCat &TheFunction()
36:      {
37:           SimpleCat Frisky(5,9);
38:           return Frisky;
39:      }
```

РЕЗУЛЬТАТ Compile error: Attempting to return a reference to a local object! (Ошибка компиляции: попытка возвратить ссылку на локальный объект!)

ПРЕДУПРЕЖДЕНИЕ

> Эта программа не компилируется на компиляторе фирмы Borland, но для нее подходят компиляторы компании Microsoft. Однако профессиональный программист никогда не станет полагаться на уступки компилятора.

АНАЛИЗ В строках 7—17 объявляется класс SimpleCat. В строке 29 инициализируется ссылка на объект класса SimpleCat с использованием результатов вызова функции TheFunction(), объявленной в строке 25. Согласно объявлению эта функция возвращает ссылку на объект класса SimpleCat.

В теле функции TheFunction() объявляется локальный объект типа SimpleCat и инициализируется его возраст и вес. Затем этот объект возвращается по ссылке. Некоторые компиляторы обладают достаточным интеллектом, чтобы распознать эту ошибку, и не позволят вам запустить данную программу на выполнение. Другие же (сразу видно, кто настоящий друг) спокойно разрешат вам выполнить эту программу с непредсказуемыми последствиями.

По возвращении функции TheFunction() локальный объект Frisky будет разрушен (надеюсь, безболезненно для самого объекта). Ссылка же, возвращаемая этой функцией, останется псевдонимом для несуществующего объекта, а это явная ошибка.

Возвращение ссылки на объект в области динамического обмена

Можно было бы попытаться решить проблему, представленную в листинге 9.13, сориентировав функцию `TheFunction()` на размещение объекта `Frisky` в области динамического обмена. В этом случае после возврата из функции `TheFunction()` объект `Frisky` будет все еще жив.

Новый подход порождает новую проблему: что делать с памятью, выделенной для объекта `Frisky`, после окончания обработки этого объекта? Эта проблема показана в листинге 9.14.

Листинг 9.14. Утечка памяти

```
1:    // Листинг 9.14.
2:     // Разрешение проблемы утечки памяти
3:     #include <iostream.h>
4:
5:     class SimpleCat
6:     {
7:     public:
8:             SimpleCat (int age, int weight);
9:            ~SimpleCat() { }
10:           int GetAge() {  return itsAge; }
11:           int GetWeight() {  return itsWeight; }
12:
13     private:
14:          int itsAge;
15:          int itsWeight;
16:    } ;
17:
18:    SimpleCat::SimpleCat(int age, int weight)
19:    {
20:      itsAge = age;
21:      itsWeight = weight;
22:    }
23:
24:    SimpleCat & TheFunction();
25:
26:    int main()
27:    {
28:         SimpleCat & rCat = TheFunction();
29:         int age = rCat.GetAge();
30:         cout << "rCat " << age << " years old!\n";
31:         cout << "&rCat: " << &rCat << endl;
32:         // Как освободить эту память?
33:         SimpleCat * pCat = &rCat;
34:         delete pCat;
```

```
35:          // Боже, на что же теперь ссылается rCat??
36:     return 0;
37:     }
38:
39:     SimpleCat &TheFunction()
40:     {
41:          SimpleCat * pFrisky = new SimpleCat(5,9);
42:          cout << "pFrisky: " << pFrisky << endl;
43:          return *pFrisky;
44:     }
```

РЕЗУЛЬТАТ

```
pFrisky: 0x00431C60
rCat 5 years old!
&rCat: 0x00431C60
```

ПРЕДУПРЕЖДЕНИЕ

Эта программа компилируется, компонуется и, кажется, работает. Но мина замедленного действия уже ожидает своего часа.

АНАЛИЗ

Функция `TheFunction()` была изменена таким образом, чтобы больше не возвращать ссылку на локальную переменную. В строке 41 выделяется некоторая область динамически распределяемой памяти и ее адрес присваивается указателю. Этот адрес выводится на экран, после чего указатель разыменовывается и объект класса `SimpleCat` возвращается по ссылке.

В строке 28 значение возврата функции `TheFunction()` присваивается ссылке на объект класса `SimpleCat`, а затем этот объект используется для получения возраста кота, и полученное значение возраста выводится на экран в строке 30.

Чтобы доказать, что ссылка, объявленная в функции `main()`, ссылается на объект, размещенный в области динамической памяти, выделенной для него в теле функции `TheFunction()`, к ссылке `rCat` применяется оператор адреса (`&`). Вполне убедителен тот факт, что адрес объекта, на который ссылается `rCat`, совпадает с адресом объекта, расположенного в свободной области памяти.

До сих пор все было гладко. Но как же теперь освободить эту область памяти, которая больше не нужна? Ведь нельзя же выполнять операции удаления на ссылках. На ум приходит одно решение: создать указатель и инициализировать его адресом, полученным из ссылки `rCat`. При этом и память будет освобождена, и условия для утечки памяти будут ликвидированы. Все же одна маленькая проблема остается: на что теперь ссылается переменная `rCat` после выполнения строки 34? Как указывалось выше, ссылка всегда должна оставаться псевдонимом реального объекта; если же она ссылается на *нулевой объект (как в данном случае), о корректности программы говорить нельзя.*

ПРИМЕЧАНИЕ

Не будет преувеличением определение программы как некорректной, если она содержит ссылку на нулевой объект (несмотря на то что она успешно компилируется), поскольку результаты ее выполнения непредсказуемы.

Для решения этой проблемы есть три пути. Первый состоит в объявлении объекта класса SimpleCat в строке 28 и возвращении этого объекта из функции TheFunction как значения. Второй — в объявлении класса SimpleCat в свободной области (в теле функции TheFunction()), но сделать это нужно так, чтобы функция TheFunction() возвращала указатель на данный объект. Затем, когда объект больше не нужен, его можно удалить в вызывающей функции с помощью оператора delete.

Третье решение (возможно, самое правильное) — объявить объект в вызывающей функции, а затем передать в функцию TheFunction()ссылку на него.

А где же указатель?

При выделении в программе памяти в области динамического обмена возвращается указатель. Важно сохранить указатель на эту область памяти, поскольку при его утрате эту память нельзя удалить, что приводит к ее утечке.

При передаче данных, хранящихся в этом блоке памяти, между функциями, необходимо следить, кому принадлежит этот указатель. Обычно ответственность за освобождение ячеек памяти в области динамического обмена ложится на ту функцию, которая их зарезервировала. Но это не догма, а лишь рекомендация для программистов.

Весьма небезопасно, если одна функция создает объект с выделением для него некоторой памяти, а другая занимается освобождением этой памяти. Неопределенность относительно владельцев указателя может привести к одной из двух проблем: можно забыть освободить память или применить оператор delete дважды к одному и тому же указателю. Любая из этих проблем может стать причиной больших неприятностей в вашей программе. Именно поэтому целесообразно придерживаться принципа, что память освобождает та функция, которая ее зарезервировала.

Если вы пишете функцию, которая требует выделения памяти в области динамического обмена, а затем возвращаете этот объект в вызывающую функцию, пересмотрите свой интерфейс. Пусть лучше вызывающая функция выделяет память, а затем передает в другую функцию этот объект как ссылку. Затем, после возвращения объекта из функции, его можно будет удалить в вызывающей функции, где он и был создан.

Рекомендуется	Не рекомендуется
Передавайте параметры функции как значения только тогда, когда в этом есть необходимость. **Возвращайте** результат работы функции как значение только тогда, когда в этом есть необходимость.	**Не используйте** ссылки на объекты, которые могут выйти в программе за пределы области видимости. **Не создавайте** ссылки на нулевые объекты.

Резюме

Сегодня вы узнали, что представляют собой ссылки и чем они отличаются от указателей. Важно уяснить для себя, что ссылки всегда инициализируют существующие объекты и их нельзя переназначить до окончания программы. Ссылка выступает псевдонимом объекта, и любое действие, выполненное над ссылкой, выполняется над ее адресатом. Доказательством этого может служить тот факт, что при взятии адреса ссылки возвращается адрес связанного с ней объекта.

Вы убедились, что передача объектов в функции как ссылок может быть более эффективной, чем передача их как значений. Передача объектов как ссылок позволяет вызываемой функции изменять значения переменных вызывающей функции.

Вы также узнали, что аргументы, передаваемые функции, и значения, возвращаемые из функций, могут передаваться как ссылки и этот процесс можно реализовать как с помощью указателей, так и с помощью ссылок.

Теперь вы научились для безопасной передачи значений между функциями использовать константные указатели на константные объекты или константные ссылки, благодаря чему достигается как эффективность, так и безопасность работы программы.

Вопросы и ответы

Зачем использовать ссылки, если указатели могут делать ту же работу?

Ссылки легче использовать, и они проще для понимания. Косвенность обращений при этом скрывается, и отсутствует необходимость в многократном разыменовании переменных.

Зачем нужны указатели, если со ссылками легче работать?

Ссылки не могут быть нулевыми, и их нельзя переназначать. Указатели предлагают большую гибкость, но их сложнее использовать.

Зачем вообще результат функции возвращать как значение?

Если возвращается объект, который является локальным в данной функции, необходимо организовать возврат его именно как значения, в противном случае возможно появление ссылки на несуществующий объект.

Если существует опасность от возвращения объекта как ссылки, почему бы тогда не сделать обязательным возврат по значению?

При возвращении объекта как ссылки достигается гораздо большая эффективность, которая заключается в экономии памяти и увеличении скорости работы программы.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний, а также ряд упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. В чем разница между ссылкой и указателем?
2. Когда нужно использовать именно указатель, а не ссылку?
3. Что возвращает оператор new, если для создания нового объекта недостаточно памяти?
4. Что представляет собой константная ссылка?
5. В чем разница между передачей объекта как ссылки и передачей ссылки в функцию?

Упражнения

1. Напишите программу, которая объявляет переменную типа int, ссылку на значение типа int и указатель на значение типа int. Используйте указатель и ссылку для управления значением переменной типа int.
2. Напишите программу, которая объявляет константный указатель на постоянное целое значение. Инициализируйте этот указатель, чтобы он указывал на целочисленную переменную varOne. Присвойте переменной varOne значение 6. Используйте указатель, чтобы присвоить переменной varOne значение 7. Создайте вторую целочисленную переменную varTwo. Переназначьте указатель, чтобы он указывал на переменную varTwo. Пока не компилируйте это упражнение.
3. Скомпилируйте программу, написанную в упражнении 2. Какие действия компилятор считает ошибочными? Какие строки генерируют предупреждения?
4. Напишите программу, которая создает блуждающий указатель.
5. Исправьте программу из упражнения 4, чтобы блуждающий указатель стал нулевым.
6. Напишите программу, которая приводит к утечке памяти.
7. Исправьте программу из упражнения 6.
8. **Жучки:** что неправильно в этой программе?

```
1:    #include <iostream.h>
2:
3:    class CAT
4:    {
5:       public:
6:          CAT(int age) {  itsAge = age; }
7:          ~CAT(){ }
8:          int GetAge() const {  return itsAge;}
9:       private:
10:         int itsAge;
11:   };
12:
13:   CAT & MakeCat(int age);
14:   int main()
15:   {
16:      int age = 7;
17:      CAT Boots = MakeCat(age);
18:      cout << "Boots is " << Boots.GetAge() << " years old!\n";
19:      return 0;
20:   }
21:
22:   CAT & MakeCat(int age)
23:   {
24:      CAT * pCat = new CAT(age);
25:      return *pCat;
26:   }
```

9. Исправьте программу из упражнения 8.

Дополнительные возможности использования функции

На занятии 5 вы познакомились с основными принципами использования функций. Теперь, когда вы знаете, как работают указатели и ссылки, перед вами открываются дополнительные возможности. Сегодня вы узнаете:

- Как перегружать функции-члены
- Как перегружать операторы
- Как создавать функции для поддержания классов с динамическим выделением памяти для переменных

Перегруженные функции-члены

На занятии 5 вы получили общие представления о полиморфизме, или перегружаемости функций. Имеется в виду объявление двух или более функций под одним именем но с разными параметрами. Функции-члены класса можно перегружать точно так же.

В классе `Rectangle` (листинг 10.1) объявляются две функции `DrawShape()`. Первая, которая не содержит списка параметров, вычерчивает прямоугольник, основываясь на текущих значениях класса. Вторая принимает два значения — ширину и длину — и в соответствии с ними создает прямоугольник, игнорируя текущие значения класса.

Листинг 10.1. Перегрузка функций-членов

```
1:   //Листинг 10.1. Перегрузка функций-членов
2:   #include <iostream.h>
3:
4:   int
5:   // Объявление класса Rectangle
6:   class Rectangle
7:   {
```

```
8:  public:
9:     // конструкторы
10:    Rectangle(int width, int height);
11:    ~Rectangle(){ }
12:
13:    // перегрузка функции-члена класса DrawShape
14:    void DrawShape() const;
15:    void DrawShape(int aWidth, int aHeight) const;
16:
17: private:
18:    int itsWidth;
19:    int itsHeight;
20: } ;
21:
22: // Применение конструктора
23: Rectangle::Rectangle(int width, int height)
24: {
25:    itsWidth = width;
26:    itsHeight = height;
27: }
28:
29:
30: // Перегруженная функция DrawShape - вариант без передачи данных
31: // Создание прямоугольника по значениям, заданным по умолчанию
32: void Rectangle::DrawShape() const
33: {
34:   DrawShape( itsWidth, itsHeight);
35: }
36:
37:
38: // Перегруженная функция DrawShape - передача двух значений
39: // Создание прямоугольника по значениям, переданным с параметрами
40: void Rectangle::DrawShape(int width, int height) const
41: {
42:    for (int i = 0; i<height; i++)
43:    {
44:      for (int j = 0; j< width; j++)
45:      {
46:      cout << "*";
47:      }
48:      cout << "\n";
49:    }
50: }
51:
52: // Выполняемая программа, демонстрирующая использование перегруженных функций
53: int main()
54: {
55:    // создание прямоугольника с размерами 30 и 5
56:    Rectangle theRect(30,5);
57:    cout << "DrawShape(): \n";
```

```
58:     theRect.DrawShape();
59:     cout << "\nDrawShape(40,2): \n";
60:     theRect.DrawShape(40,2);
61:   return 0;
62:   }
```

РЕЗУЛЬТАТ

```
DrawShape():
******************************
******************************
******************************
******************************
******************************

DrawShape(40,2):
****************************************************************
****************************************************************
```

АНАЛИЗ Листинг 10.1 представляет собой усеченную версию проекта, рассмотренного в главе подведения итогов за первую неделю. Чтобы сократить размер программы, был удален блок контроля за соответствием значений заданным типам. Основной код был упрощен до простой выполняемой программы без показа пользовательского меню.

Сейчас для нас важны строки 14 и 15, где происходит перегрузка функции `DrawShape()`. Использование перегруженных вариантов этой функции показано далее, в строках с 30 по 50. Обратите внимание, что версия функции `DrawShape()` без параметров обращается к варианту функции, содержащей два параметра, и передает в нее текущие значения переменных-членов. При программировании всегда следует избегать дублирования одинаковых программных кодов. В противном случае придется держать в памяти все созданные копии функций, чтобы при изменении программного кода в одной из них внести соответствующие изменения во все копии.

В строках программы с 52 по 62 создается прямоугольный объект и вызывается функция `DrawShape()`. В первый раз в функцию не передаются параметры, а во второй раз передается два значения типа `unsigned short integer`.

Компилятор выбирает правильное объявление функции по количеству и типу заданных параметров. Дополнительно можно задать в этой же программе еще одно объявление функции `DrawShape()`, в параметрах которой будет одно значение размера и переменная перечисления, позволяющая пользователю указать, что обозначает данный размер — ширину или длину прямоугольника.

Использование значений, заданных по умолчанию

Функции-члены класса, подобно обычным функциям, могут использовать значения, заданные по умолчанию. При объявлении функций-членов с аргументами, задаваемыми по умолчанию, используется уже знакомый вам синтаксис, как показано в листинге 10.2

Листинг 10.2. Использование значений, заданных по умолчанию

```
1:  //Листинг 10.2. Использование значений, заданных по умолчанию
2:  #include <iostream.h>
3:
4:  int
5:
6:  // Объявление класса Rectangle
7:  class Rectangle
8:  {
9:  public:
10:     // конструкторы
11:     Rectangle(int width, int height);
12:     ~Rectangle(){ }
13:     void DrawShape(int aWidth, int aHeight, bool UseCurrentVals = false) const;
14:
15:  private:
16:     int itsWidth;
17:     int itsHeight;
18:  } ;
19:
20:  //Применение конструктора
21:  Rectangle::Rectangle(int width, int height):
22:  itsWidth(width),          // инициализация
23:  itsHeight(height)
24:  { }                       // пустое тело
25:
26:
27:  // для третьего параметра используются значения по умолчанию
28:  void Rectangle::DrawShape(
29:     int width,
30:     int height,
31:     bool UseCurrentValue
32:     ) const
33:  {
34:     int printWidth;
35:     int printHeight;
36:
37:     if (UseCurrentValue == true)
38:     {
39:       printWidth = itsWidth;          // используется значение текущего класса
40:       printHeight = itsHeight;
41:     }
42:     else
43:     {
44:       printWidth = width;             // используются значения параметра
45:       printHeight = height;
46:     }
47:
48:
```

```
49:     for (int i = 0; i<printHeight; i++)
50:     {
51:       for (int j = 0; j< printWidth; j++)
52:       {
53:       cout << "*";
54:       }
55:       cout << "\n";
56:     }
57:   }
58:
59:   // Выполняемая программа показывает использование перегруженных функций
60:   int main()
61:   {
62:     // создание прямоугольника 30 на 5
63:     Rectangle theRect(30,5);
64:     cout << "DrawShape(0,0,true)...\n";
65:     theRect.DrawShape(0,0,true);
66:     cout <<"DrawShape(40,2)...\n";
67:     theRect.DrawShape(40,2);
68:     return 0;
69:   }
```

РЕЗУЛЬТАТ

```
DrawShape(0,0,true)...
******************************
******************************
******************************
******************************
******************************

DrawShape(40,2)...
****************************************************************
****************************************************************
```

АНАЛИЗ В листинге 10.2 перегруженная функция `DrawShape()` заменена простой функцией с параметрами, задаваемыми по умолчанию. Функция определена в строке 13 с тремя параметрами. Первые два, `aWidth` и `aHeigth`, относятся к типу `USHORT`, а третий представляет собой логическую переменную `UseCurrentVals`, которой по умолчанию присваивается значение `false`.

Выполнение этой немного громоздкой функции начинается со строки 28. Сначала проверяется значение переменной `UseCurrentVals`. Если эта переменная содержит значение `true`, то для присвоения значений локальным переменным `printWidth` и `printHeigth` используются соответственно переменные-члены `itsWidth` и `itsHeigth`.

Если окажется, что переменная `UseCurrentVals` содержит значение `false` либо по умолчанию, либо оно является результатом установок, сделанных пользователем, то переменным `printWidth` и `printHeigth` присваиваются значения параметров функции, заданные по умолчанию.

Обратите внимание, что если `UseCurrentVals` истинно, то значения параметров функции просто игнорируются.

Выбор между значениями по умолчанию и перегруженными функциями

В листингах 10.1 и 10.2 выполняются одни и те же задачи, но использование перегруженных функций в листинге 10.1 делает программу более естественной и читабельной. Кроме того, если в программе потребуется третий вариант функции, например, для того, чтобы пользователь мог задать только один размер геометрической фигуры, а другой оставить по умолчанию, не составит труда добавить новую перегруженную функцию.

Как решить, что следует использовать в программе — перегруженные функции или значения по умолчанию? Примите к сведению следующие положения. Использование перегруженных функций предпочтительнее, если:

- не существует стандартных общепринятых значений, которые можно было бы использовать по умолчанию;
- в программе в зависимости от ситуации необходимо использовать различные алгоритмы;
- необходимо иметь возможность изменять тип значений, передаваемых в функцию.

Конструктор, принятый по умолчанию

На шестом занятии, изучая базовые классы, вы узнали, что в случае отсутствия явного объявления конструктора класса используется конструктор по умолчанию, который не содержит параметров и никак себя не проявляет в программе. Не составляет труда создать собственный конструктор, применяемый по умолчанию, который также не будет принимать никаких параметров, но позволит управлять созданием объектов класса.

Конструктор, предоставляемый компилятором, называется заданным по умолчанию. В то же время конструктором по умолчанию называется также любой другой конструктор класса, не содержащий параметров. Это может показаться странным, но ситуация прояснится, если посмотреть на дело с точки зрения применения данного конструктора на практике.

Примите к сведению, что если в программе был создан какой-либо конструктор, то компилятор не будет предлагать свой конструктор по умолчанию. Поэтому, если вам нужен конструктор без параметров, а в программе уже создан один конструктор, то конструктор по умолчанию нужно будет создать самостоятельно!

Перегрузка конструкторов

Конструктор предназначен для создания объекта. Например, назначение конструктора `Rectangle` состоит в создании объекта прямоугольник. До запуска конструктора прямоугольник в программе отсутствует. Существует только зарезервированная для него область памяти. По завершении выполнения конструктора в программе появляется готовый для использования объект.

Конструкторы, как и все другие функции, можно перегружать. Перегрузка конструкторов — мощное средство повышения эффективности и гибкости программы.

Например, рассматриваемый нами объект `Rectangle` может иметь два конструктора. В первом задается ширина и длина прямоугольника, а второй не имеет параметров и для установки размеров использует значения по умолчанию. Эта идея реализована в листинге 10.3.

Листинг 10.3. Перегрузка конструктора

```
1:    // Листинг 10.3.
2:    // Перегрузка конструктора
3:
4:    #include <iostream.h>
5:
6:    class Rectangle
7:    {
8:    public:
9:       Rectangle();
10:      Rectangle(int width, int length);
11:      ~Rectangle() { }
12:      int GetWidth() const {   return itsWidth; }
13:      int GetLength() const {   return itsLength; }
14:   private:
15:      int itsWidth;
16:      int itsLength;
17:   } ;
18:
19:   Rectangle::Rectangle()
20:   {
21:      itsWidth = 5;
22:      itsLength = 10;
23:   }
24:
25:   Rectangle::Rectangle (int width, int length)
26:   {
27:      itsWidth = width;
28:      itsLength = length;
29:   }
30:
31:   int main()
32:   {
33:      Rectangle Rect1;
34:    cout << "Rect1 width: " << Rect1.GetWidth() << endl;
35:    cout << "Rect1 length: " << Rect1.GetLength() << endl;
36:
37:    int aWidth, aLength;
38:    cout << "Enter a width: ";
39:    cin >> aWidth;
40:    cout << "\ nEnter a length: ";
41:    cin >> aLength;
42:
```

```
43:     Rectangle Rect2(aWidth, aLength);
44:     cout << "\ nRect2 width: " << Rect2.GetWidth() << endl;
45:     cout << "Rect2 length: " << Rect2.GetLength() << endl;
46:       return 0;
47:   }
```

РЕЗУЛЬТАТ

```
Rect1 width: 5
Rect1 length: 10
Enter a width: 20

Enter a length: 50

Rect2 width: 20
Rect2 length: 50
```

АНАЛИЗ Класс `Rectangle` объявляется в строках с 6 по 17. В классе представлены два конструктора: один использует значения по умолчанию (строка 9), а второй принимает значения двух целочисленных параметров (строка 10). В строке 33 прямоугольный объект создается с использованием первого конструктора. Значения размеров прямоугольника, принятые по умолчанию, выводятся на экран в строках 34 и 35. Строки программы с 37 по 41 выводят на экран предложения пользователю ввести собственные значения ширины и длины прямоугольника. В строке 43 вызывается второй конструктор, использующий два параметра с только что установленными значениями. И наконец, значения размеров прямоугольника, установленные пользователем, выводятся на экран в строках 44 и 45.

Как и при использовании других перегруженных функций, компилятор выбирает нужное объявление конструктора, основываясь на числе и типе параметров.

Инициализация объектов

До сих пор переменные-члены объектов задавались прямо в теле конструктора. Выполнение конструктора происходит в два этапа: инициализация и выполнение тела конструктора.

Большинство переменных может быть задано на любом из этих этапов: как во время инициализации, так и во время выполнения конструктора. Но логически правильнее, а зачастую и эффективнее, инициализировать переменные-члены во время инициализации конструктора. В следующем примере показана инициализация переменных-членов:

```
CAT():          // имя конструктора и список параметров
itsAge(5),      // инициализация списка
itsWeigth(8)
{}              // тело конструктора
```

После скобки закрытия списка параметров конструктора ставится двоеточие. Затем перечисляются имена переменных-членов. Пара круглых скобок со значением за именем переменной используется для инициализации этой переменной. Если инициализируется сразу несколько переменных, то они должны быть отделены запятыми. В листинге 10.4 показана инициализация переменных конструкторов, взятых из листинга 10.3. В данном примере инициализация переменных используется вместо присвоения им значений в теле конструктора.

Листинг 10.4. Фрагмент программного кода с инициализацией переменных-членов

```
1:   Rectangle::Rectangle():
2:      itsWidth(5),
3:      itsLength(10)
4:   {
5:   }
6:
7:   Rectangle::Rectangle (int width, int length):
8:      itsWidth(width),
9:      itsLength(length)
10:  {
11:  }
```

РЕЗУЛЬТАТ Отсутствует.

Некоторые переменные можно только инициализировать и нельзя присваивать им значения: например, в случае использования ссылок и констант. Безусловно, переменной-члену можно присвоить значение прямо в теле конструктора, но для упрощения программы лучше по возможности устанавливать значения переменных-членов на этапе инициализации конструктора.

Конструктор-копировщик

Помимо конструктора и деструктора, компилятор по умолчанию предоставляет также конструктор-копировщик, который вызывается всякий раз, когда нужно создать копию объекта.

Когда объект передается как значение либо в функцию, либо из функции в виде возврата, всегда создается его временная копия. Если в программе обрабатывается объект, созданный пользователем, то для выполнения этих операций вызывается конструктор-копировщик класса, как было показано на предыдущем занятии в листинге 9.6.

Все копировщики принимают только один параметр — ссылку на объект в том же классе. Разумно будет сделать эту ссылку константной, так как конструктор не должен изменять передаваемый в него объект. Например:

```
CAT(const CAT & theCat);
```

В данном случае конструктор `CAT` принимает константную ссылку на объект класса `CAT`. Цель использования конструктора-копировщика состоит в создании копии объекта `theCat`.

Копировщик, заданный компилятором по умолчанию, просто копирует все переменные-члены из указанного в параметре объекта в переменные-члены нового объекта. Такое копирование называется поверхностным; и, хотя оно подходит для большинства случаев, могут возникнуть серьезные проблемы, если переменные-члены окажутся указателями на ячейки динамической памяти.

Поверхностное копирование создает несколько переменных-членов в разных объектах, которые ссылаются на одни и те же ячейки памяти. Глубинное копирование переписывает значения переменных по новым адресам.

Например, класс CAT содержит переменную-член theCat, которая указывает на ячейку в области динамической памяти, где сохранено некоторое целочисленное значение. Копировщик по умолчанию скопирует переменную theCat из старого класса CAT в переменную theCat в новом классе CAT. При этом оба объекта будут указывать на одну и ту же ячейку памяти (рис. 10.1).

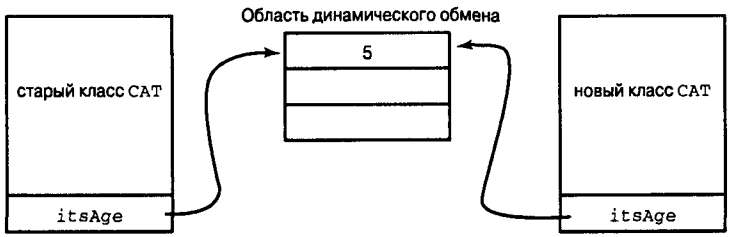


Рис. 10.1. Использование копировщика, заданного по умолчанию

Проблемы могут возникнуть, если программа выйдет за область видимости одного из классов CAT. Как уже отмечалось при изучении указателей, назначение деструктора состоит в том, чтобы очищать память от ненужных объектов. Если деструктор исходного класса CAT очистит свои ячейки памяти, а объекты нового класса CAT все так же будут ссылаться на эти ячейки, то над программной нависнет смертельная опасность. Эта проблема проиллюстрирована на рис. 10.2.

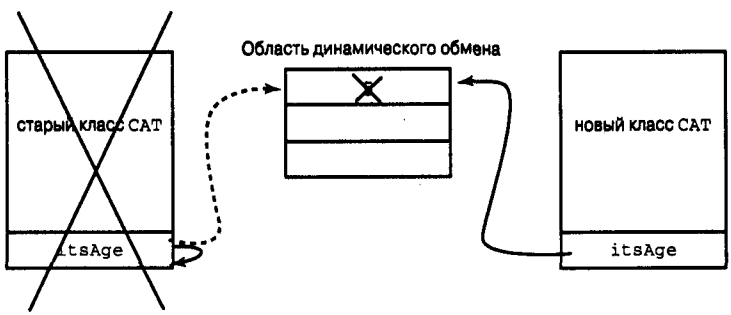


Рис. 10.2. Возникновение ошибочного указателя

Чтобы предупредить возникновение подобных проблем, нужно вместо копировщика по умолчанию создать и использовать собственный копировщик, который будет осуществлять глубинное копирование с перемещением значений переменных-членов в новые адреса памяти. Этот процесс показан в листинге 10.5

Листинг 10.5. Конструктор-копировщик

```
1:   // Листинг 10.5.
2:   // Конструктор-копировщик
3:
4:   #include <iostream.h>
5:
6:   class CAT
7:   {
8:      public:
9:        CAT();                    // конструктор по умолчанию
10:       CAT (const CAT &);        // конструктор-копировщик
```

```
11:       ~CAT();                    // деструктор
12:       int GetAge() const {   return *itsAge; }
13:       int GetWeight() const {   return *itsWeight; }
14:       void SetAge(int age) {   *itsAge = age; }
15:
16:     private:
17:       int *itsAge;
18:       int *itsWeight;
19:   } ;
20:
21:   CAT::CAT()
22:   {
23:      itsAge = new int;
24:      itsWeight = new int;
25:      *itsAge = 5;
26:      *itsWeight = 9;
27:   }
28:
29:   CAT::CAT(const CAT & rhs)
30:   {
31:      itsAge = new int;
32:      itsWeight = new int;
33:      *itsAge = rhs.GetAge();          // открытый метод доступа
34:      *itsWeight = *(rhs.itsWeight);   // закрытый метод доступа
35:   }
36:
37:   CAT::~CAT()
38:   {
39:      delete itsAge;
40:      itsAge = 0;
41:      delete itsWeight;
42:      itsWeight = 0;
43:   }
44:
45:   int main()
46:   {
47:      CAT frisky;
48:    cout << "frisky's age: " << frisky.GetAge() << endl;
49:    cout << "Setting frisky to 6...\ n";
50:    frisky.SetAge(6);
51:    cout << "Creating boots from frisky\ n";
52:    CAT boots(frisky);
53:    cout << "frisky's age: " <<   frisky.GetAge() << endl;
54:    cout << "boots' age: " << boots.GetAge() << endl;
55:    cout << "setting frisky to 7...\ n";
56:    frisky.SetAge(7);
57:    cout << "frisky's age: " <<   frisky.GetAge() << endl;
58:    cout << "boot's age: " << boots.GetAge() << endl;
59:    return 0;
60: }
```

РЕЗУЛЬТАТ

```
frisky's age: 5
Setting frisky to 6...
Creating boots from frisky
frisky's age: 6
boots' age: 6
setting frisky to 7...
frisky's age: 7
boots' age: 6
```

АНАЛИЗ

В строках программы с 6 по 19 объявляется класс CAT. Обратите внимание, что в строке 9 объявляется конструктор по умолчанию, а в строке 10 — конструктор-копировщик.

В строках 17 и 18 объявляется две переменные-члены, представляющие собой указатели на целочисленные значения. В реальной жизни трудно вообразить, для чего может понадобиться создание переменных-членов как указателей на целочисленные значения. Но в данном случае такие переменные являются отличными объектами для демонстрации методов управления переменными-членами, сохраненными в динамической области памяти.

Конструктор по умолчанию в строках с 21 по 27 выделяет для переменных области динамической памяти и инициализирует эти переменные.

Работа копировщика начинается со строки 29. Обратите внимание, что в копировщике задан параметр rhs. Использовать в параметрах копировщиков символику rhs, что означает *right-hand side* (стоящий справа), — общепринятая практика. Если вы посмотрите на строки 33 и 34, то увидите, что в выражениях присваивания имена параметров копировщика располагаются справа от оператора присваивания (знака равенства).

Вот как работает копировщик. Строки 31 и 32 выделяют свободные ячейки в области динамической памяти. Затем, в строках 33 и 34 в новые ячейки переписываются значения из существующего класса CAT.

Параметр rhs соответствует объекту классу CAT, который передается в копировщик в виде константной ссылки. Как объект класса CAT, rhs содержит все переменные-члены любого другого класса CAT.

Любой объект класса CAT может открыть доступ к закрытым переменным-членам для любого другого объекта класса CAT. В то же время для внешних обращений всегда лучше создавать открытые члены, где это только возможно. Функция-член rhs.GetAge() возвращает значение, сохраненное в переменной-члене itsAge, адрес которой представлен в rhs.

Процедуры, осуществляемые программой, продемонстрированы на рис. 10.3. Значение, на которое ссылалась переменная-член исходного класса CAT, копируется в новую ячейку памяти, адрес которой представлен в такой же переменной-члене нового класса CAT.

В строке 47 вызывается объект frisky из класса CAT. Значение возраста, заданное в frisky, выводится на экран, после чего в строке 50 переменной возраста присваивается новое значение — 6. В строке 52 методом копирования объекта frisky создается новый объект boots класса CAT. Если бы в качестве параметра передавался объект frisky, то вызов копировщика осуществлялся бы компилятором.

В строках 53 и 54 выводится возраст обеих кошек. Обратите внимание, что в обоих случаях в объектах frisky и boots записан возраст 6, тогда как если бы объект boots создавался не методом копирования, то по умолчанию было бы присвоено значение 5. В строке 56 значение возраста в объекте было изменено на 7 и вновь выведены на экран значения обоих объектов. Значение объекта frisky действительно изменилось на 7, тогда как в boots сохранилось прежнее значение возраста 6. Это доказывает, что переменная объекта frisky была скопирована в объект boots по новому адресу.

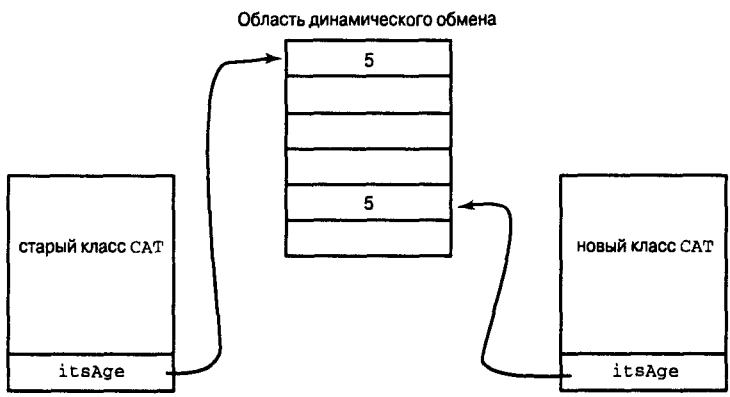


Рис. 10.3. Пример глубинного копирования

Когда выполнение программы выходит за область видимости класса `CAT`, автоматически запускается деструктор. Выполнение деструктора класса `CAT` показано в строках с 37 по 43. Оператор `delete` применяется к обоим указателям — `itsAge` и `itsWeigth`, после чего оба указателя для надежности обнуляются.

Перегрузка операторов

Язык C++ располагает рядом встроенных типов данных, включая `int`, `real`, `char` и т.д. Для работы с данными этих типов используются встроенные операторы — суммирования (+) и умножения (*). Кроме того, в C++ существует возможность добавлять и перегружать эти операторы для собственных классов.

Чтобы в деталях рассмотреть процедуру перегрузки операторов, в листинге 10.6 создается новый класс `Counter`. Объект класса `Counter` будет использоваться в других приложениях для подсчета циклов инкрементации, декрементации и других повторяющихся процессов.

Листинг 10.6. Класс `Counter`

```
1:   // Листинг 10.6.
2:   // Класс Counter
3:
4:   int
5:   #include <iostream.h>
6:
7:   class Counter
8:   {
9:     public:
10:       Counter();
11:       ~Counter(){ }
12:       int GetItsVal()const {   return itsVal; }
13:       void SetItsVal(int x) { itsVal = x; }
14:
15:     private:
16:       int itsVal;
```

```
17:
18:   } ;
19:
20:   Counter::Counter():
21:   itsVal(0)
22:   { }
23:
24:   int main()
25:   {
26:      Counter i;
27:      cout << "The value of i is " << i.GetItsVal() << endl;
28:   return 0;
29:   }
```

РЕЗУЛЬТАТ

```
The value of i is 0
```

АНАЛИЗ Судя по определению в строках программы с 7 по 18, это совершенно бесполезный класс. В нем объявлена единственная переменная-член типа `int`. Конструктор по умолчанию, который объявляется в строке 10 и выполняется в строке 20, инициализирует переменную-член нулевым значением.

В отличие от обычной переменной типа `int`, объект класса `Counter` не может использоваться в операциях приращения, прибавляться, присваиваться или подвергаться другим манипуляциям. В связи с этим выведение значения данного объекта на печать также сопряжено с рядом трудностей.

Запись функции инкремента

Ограничения использования объекта нового класса, которые упоминались выше, можно преодолеть путем перегрузки операторов. Например, существует несколько способов восстановления возможности приращения объекта класса `Counter`. Один из них состоит в том, чтобы перегрузить функцию инкрементации, как показано в листинге 10.7.

Листинг 10.7. Добавление в класс оператора инкремента

```
1:   // Листинг 10.7.
2:   // Добавление в класс Counter оператора инкремена
3:
4:   int
5:   #include <iostream.h>
6:
7:   class Counter
8:   {
9:     public:
10:        Counter();
11:        ~Counter(){ }
12:        int GetItsVal()const {   return itsVal; }
13:        void SetItsVal(int x) { itsVal = x; }
14:        void Increment() {   ++itsVal; }
```

```
15:
16:      private:
17:        int itsVal;
18:
19:   } ;
20:
21:   Counter::Counter():
22:   itsVal(0)
23:   { }
24:
25:   int main()
26:   {
27:      Counter i;
28:      cout << "The value of i is " << i.GetItsVal() << endl;
29:      i.Increment();
30:      cout << "The value of i is " << i.GetItsVal() << endl;
31:   return 0;
32:   }
```

РЕЗУЛЬТАТ

```
The value of i is 0
The value of i is 1
```

АНАЛИЗ В листинге 10.7 добавляется функция оператора инкремента, определенная в строке 14. Хотя программа работает, выглядит она довольно неуклюже. Программа из последних сил старается перегрузить `++operator`, но это можно реализовать другим способом.

Перегрузка префиксных операторов

Чтобы перегрузить префиксный оператор, можно использовать функцию следующего типа:

```
returnType Operator op (параметры)
```

В данном случае *op* — это перегружаемый оператор. Тогда для перегрузки оператора преинкремента используем функцию

```
void operator++ ()
```

Этот способ показан в листинге 10.8.

Листинг 10.8. Перегрузка оператора преинкремента

```
1:   // Листинг 10.8.
2:   // Перегрузка оператора преинкремента в классе Counter
3:
4:   int
5:   #include <iostream.h>
6:
7:   class Counter
8:   {
```

```
9:     public:
10:        Counter();
11:        ~Counter(){ }
12:        int GetItsVal()const {    return itsVal; }
13:        void SetItsVal(int x) { itsVal = x; }
14:        void Increment() {    ++itsVal; }
15:        void operator++ () {    ++itsVal; }
16:
17:      private:
18:        int itsVal;
19:
20:    } ;
21:
22:    Counter::Counter():
23:    itsVal(0)
24:    { }
25:
26:    int main()
27:    {
28:       Counter i;
29:       cout << "The value of i is " << i.GetItsVal() << endl;
30:       i.Increment();
31:       cout << "The value of i is " << i.GetItsVal() << endl;
32:       ++i;
33:       cout << "The value of i is " << i.GetItsVal() << endl;
34:    return 0;
35:    }
```

РЕЗУЛЬТАТ

```
The value of i is 0
The value of i is 1
The value of i is 2
```

АНАЛИЗ

В строке 15 перегружается `operator++`, который затем используется в строке 32. В результате объект класса `Counter` получает функции, которые можно было ожидать судя по его названию. Далее объекту сообщаются дополнительные возможности, призванные повысить эффективность его использования, в частности возможность контроля за максимальным значением, которое нельзя превышать в ходе приращения.

Но в работе перегруженного оператора инкремента существует один серьезный недостаток. В данный момент в программе не удастся выполнить следующее выражение:

```
Counter a = ++i;
```

В этой строке делается попытка создать новый объект класса `Counter` — `a`, которому присваивается приращенное значение переменной `i`. Хотя встроенный конструктор-копировщик поддерживает операцию присваивания, текущий оператор инкремента не возвращает объект класса `Counter`. Сейчас он возвращает пустое значение `void`. Невозможно присвоить значение `void` объекту класса `Counter`. (Невозможно создать что-то из ничего!)

Типы возвратов перегруженных функций операторов

Все, что нам нужно, — это возвратить объект класса Counter таким образом, чтобы его можно было присвоить другому объекту класса Counter. Как это сделать? Один подход состоит в том, чтобы создать временный объект и возвратить его. Он показан в листинге 10.9.

Листинг 10.9. Возвращение временного объекта

```
1:   // Листинг 10.9.
2:   // Возвращение временного объекта
3:
4:   int
5:   #include <iostream.h>
6:
7:   class Counter
8:   {
9:     public:
10:        Counter();
11:        ~Counter(){ }
12:        int GetItsVal()const {    return itsVal; }
13:        void SetItsVal(int x) { itsVal = x; }
14:        void Increment() {    ++itsVal; }
15:        Counter operator++ ();
16:
17:      private:
18:        int itsVal;
19:
20:   } ;
21:
22:   Counter::Counter():
23:   itsVal(0)
24:   { }
25:
26:   Counter Counter::operator++()
27:   {
28:     ++itsVal;
29:     Counter temp;
30:     temp.SetItsVal(itsVal);
31:     return temp;
32:   }
33:
34:   int main()
35:   {
36:      Counter i;
37:      cout << "The value of i is " << i.GetItsVal() << endl;
38:      i.Increment();
39:      cout << "The value of i is " << i.GetItsVal() << endl;
40:      ++i;
41:      cout << "The value of i is " << i.GetItsVal() << endl;
```

```
42:     Counter a = ++i;
43:     cout << "The value of a: " << a.GetItsVal();
44:     cout << " and i: " << i.GetItsVal() << endl;
45:     return 0;
46:  }
```

Результат

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
```

Анализ В данной версии программы operator++ объявлен в строке 15 таким образом, что может возвращать объекты класса Counter. В строке 29 создается временный объект temp, и ему присваивается значение текущего объекта Counter. Значение временной переменной возвращается и тут же, в строке 42, присваивается новому объекту a.

Возвращение безымянных временных объектов

В действительности нет необходимости присваивать имя временному объекту, как это было сделано в предыдущем листинге в строке 29. Если в классе Counter есть принимающий значение конструктор, то параметру этого конструктора можно просто присвоить значение возврата оператора инкремента. Эта идея реализована в листинге 10.10.

Листинг 10.10. Возвращение безымянного временного объекта

```
1:  // Листинг 10.10.
2:  // Возвращение безымянного временного объекта
3:
4:  int
5:  #include <iostream.h>
6:
7:  class Counter
8:  {
9:    public:
10:       Counter();
11:       Counter(int val);
12:       ~Counter(){ }
13:       int GetItsVal()const {   return itsVal; }
14:       void SetItsVal(int x) { itsVal = x; }
15:       void Increment() {   ++itsVal; }
16:       Counter operator++ ();
17:
18:     private:
19:       int itsVal;
20:
21:  } ;
22:
23:  Counter::Counter():
```

```
24:   itsVal(0)
25:   { }
26:
27:   Counter::Counter(int val):
28:   itsVal(val)
29:   { }
30:
31:   Counter Counter::operator++()
32:   {
33:      ++itsVal;
34:      return Counter (itsVal);
35:   }
36:
37:   int main()
38:   {
39:      Counter i;
40:      cout << "The value of i is" << i.GetItsVal() << endl;
41:      i.Increment();
42:      cout << "The value of i is" << i.GetItsVal() << endl;
43:      ++i;
44:      cout << "The value of i is" << i.GetItsVal() << endl;
45:      Counter a = ++i;
46:      cout << "The value of a: " << a.GetItsVal();
47:      cout << " and i: " << i.GetItsVal() << endl;
48:      return 0;
49:   }
```

РЕЗУЛЬТАТ

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
```

АНАЛИЗ

В строке 11 определен новый конструктор, который принимает значение типа `int`. Данный конструктор выполняется в строках с 27 по 29. Происходит инициализация переменной `itsVal` значением, переданным в параметре.

Выполнение оператора инкремента в данной программе упрощено. В строке 33 осуществляется приращение переменной `itsVal`. Затем в строке 34 создается временный объект класса `Counter`, которому присваивается значение переменной `itsVal`. Это значение затем возвращается как результат выполнения оператора инкремента.

Подобное решение выглядит более элегантно, но возникает вопрос, для чего вообще нужно создавать временные объекты. Напомним, что создание и удаление временного объекта в памяти компьютера требует определенных временных затрат. Кроме того, если объект `i` уже существует и имеет правильное значение, почему бы просто не возвратить его? Реализуем эту идею с помощью указателя `this`.

Использование указателя this

На прошлом занятии уже рассматривалось использование указателя this. Этот указатель можно передавать в функцию-член оператора инкремента точно так же, как в любую другую функцию-член. Указатель this связан с объектом i и в случае разыменовывания возвращает объект, переменная которого itsVal уже содержит правильное значение. В листинге 10.11 показано возвращение указателя this, что снимает необходимость создания временных объектов.

Листинг 10.11. Возвращение указателя this

```
1:  // Листинг 10.11.
2:  // Возвращение указателя this
3:
4:  int
5:  #include <iostream.h>
6:
7:  class Counter
8:  {
9:    public:
10:       Counter();
11:       ~Counter(){ }
12:       int GetItsVal()const {   return itsVal; }
13:       void SetItsVal(int x) { itsVal = x; }
14:       void Increment() {   ++itsVal; }
15:       const Counter& operator++ ();
16:
17:     private:
18:       int itsVal;
19:
20:  } ;
21:
22:  Counter::Counter():
23:  itsVal(0)
24:  { } ;
25:
26:  const Counter& Counter::operator++()
27:  {
28:     ++itsVal;
29:     return *this;
30:  }
31:
32:  int main()
33:  {
34:     Counter i;
35:     cout << "The value of i is " << i.GetItsVal() << endl;
36:     i.Increment();
37:     cout << "The value of i is " << i.GetItsVal() << endl;
38:     ++i;
39:     cout << "The value of i is " << i.GetItsVal() << endl;
```

```
40:     Counter a = ++i;
41:     cout << "The value of a: " << a.GetItsVal();
42:     cout << " and i: " << i.GetItsVal() << endl;
43:     return 0;
44: }
```

РЕЗУЛЬТАТ

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
```

АНАЛИЗ Выполнение оператора приращения в строках с 26 по 30 заменено разыменовыванием указателя `this` и возвращением текущего объекта. В результате объект класса `Counter` присваивается новому объекту `a` этого же класса. Как уже отмечалось выше, если объект класса `Counter` требует выделения памяти, необходимо заместить конструктор-копировщик. Но в данном случае конструктор-копировщик, заданный по умолчанию, отлично справляется со своими задачами.

Обратите внимание, что возвращаемое значение представляет собой ссылку класса `Counter`, благодаря чему отпадает необходимость в создании каких-либо дополнительных временных объектов. Ссылка задана как `const`, поскольку не должна меняться при использовании в функции.

Перегрузка постфиксных операторов

До сих пор рассматривалась перегрузка оператора преинкремента. Что если перегрузить оператор постинкремента? Тут перед компилятором встает проблема: как различить между собой операторы постинкремента и преинкремента. Существует договоренность, что при определении функции оператора постинкремента устанавливается целочисленный параметр. Значение параметра не имеет смысла. Он используется только как флаг, который сообщает, что перед нами оператор постинкремента.

Различия между преинкрементом и постинкрементом

Прежде чем приступить к перегрузке оператора постинкремента, следует четко понять, чем он отличается от оператора преинкремента. Подробно эта тема рассматривалась на занятии 4 (см. листинг 4.3).

Вспомните, преинкремент означает прирастить, затем возвратить значение, а постинкремент — возвратить значение, а потом прирастить.

Точно так же и в нашем примере оператор преинкремента приращивает значение, после чего возвращает объект, а оператор постинкремента возвращает объект с исходным значением. Чтобы проследить этот процесс, нужно создать временный объект, в котором будет сохранено исходное значение, затем выполнить приращение в исходном объекте и вновь вернуть его во временный объект.

Давайте все это повторим еще раз. Посмотрите на следующее выражение:

```
a = x++;
```

Если исходно переменная `x` равнялась 5, то в этом выражении переменной `a` будет присвоено значение 5, зато переменная `x` станет равной 6. Если `x` не просто переменная, а объект, то его оператор постинкремента должен сохранить исходное значение 5

во временном объекте, прирастить значение объекта x до 6, после чего возвратить значение временного объекта и присвоить его объекту a.

Обратите внимание, что, поскольку речь идет о временном объекте, его следует возвращать как значение, а не как ссылку, так как временный объект выйдет из области видимости как только функция возвратит свое значение.

В листинге 10.12 показано использование обоих операторов.

Листинг 10.12. Операторы преинкремента и постинкремента

```
1:  // Листинг 10.12.
2:  // Возвращение разыменованного указателя this
3:
4:  int
5:  #include <iostream.h>
6:
7:  class Counter
8:  {
9:  public:
10:     Counter();
11:     ~Counter(){ }
12:     int GetItsVal()const {   return itsVal; }
13:     void SetItsVal(int x) { itsVal = x; }
14:     const Counter& operator++ ();          // оператор преинкремента
15:     const Counter operator++ (int);        // оператор постинкремента
16:
17:  private:
18:     int itsVal;
19:  } ;
20:
21:  Counter::Counter():
22:  itsVal(0)
23:  { }
24:
25:  const Counter& Counter::operator++()
26:  {
27:     ++itsVal;
28:     return *this;
29:  }
30:
31:  const Counter Counter::operator++(int x)
32:  {
33:     Counter temp(*this);
34:     ++itsVal;
35:     return temp;
36:  }
37:
38:  int main()
39:  {
40:     Counter i;
41:     cout << "The value of i is " << i.GetItsVal() << endl;
```

```
42:     i++;
43:     cout << "The value of i is " << i.GetItsVal() << endl;
44:     ++i;
45:     cout << "The value of i is " << i.GetItsVal() << endl;
46:     Counter a = ++i;
47:     cout << "The value of a: " << a.GetItsVal();
48:     cout << " and i: " << i.GetItsVal() << endl;
49:     a = i++;
50:     cout << "The value of a: " << a.GetItsVal();
51:     cout << " and i: " << i.GetItsVal() << endl;
52:  return 0;
53:  }
```

РЕЗУЛЬТАТ

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of a: 3 and i: 3
The value of a: 3 and i: 4
```

АНАЛИЗ Оператор постинкремента объявляется в строке 15 и выполняется в строках с 31 по 36. Обратите внимание, что в объявлении оператора преинкремента в строке 14 не задан целочисленный параметр `x`, выполняющий роль флага. При определении оператора постинкремента используется флаг`x`, чтобы указать компилятору, что это именно постинкремент. Значение параметра `x` нигде и никогда не используется.

Синтаксис перегрузки операторов с одним операндом

Объявление перегруженных операторов выполняется так же, как и функций. Используйте ключевое слово `operator`, за которым следует сам перегружаемый оператор. В функциях операторов с одним операндом параметры не задаются, за исключением операторов постинкремента и постдекремента, в которых целочисленный параметр играет роль флага.

Пример перегрузки оператора преинкремента:

```
const Counter& Counter::operator++ ();
```

Пример перегрузки оператора постдекремента:

```
const Counter& Counter::operator-- (int);
```

Оператор суммирования

Операторы приращения, рассмотренные выше, оперируют только с одним операндом. Оператор суммирования (+) — это представитель операторов с двумя операндами. Он выполняет операции с двумя объектами. Как выполнить перегрузку оператора суммирования для класса `Counter`?

Цель состоит в том, чтобы объявить две переменные класса `Counter`, после чего сложить их, как в следующем примере:

```
Counter переменная_один, переменная_два, переменная_три;
переменная_три= переменная_один + переменная_два;
```

Начнем работу с записи функции Add(), в которой объект Counter будет выступать аргументом. Эта функция должна сложить два значения, после чего возвратить Counter с полученным результатом. Данный подход показан в листинге 10.13.

Листинг 10.13. Функция Add()

```
1:  // Листинг 10.13.
2:  // Функция Add
3:
4:  int
5:  #include <iostream.h>
6:
7:  class Counter
8:  {
9:  public:
10:     Counter();
11:     Counter(int initialValue);
12:     ~Counter(){ }
13:     int GetItsVal()const {return itsVal; }
14:     void SetItsVal(int x) {itsVal = x; }
15:     Counter Add(const Counter &);
16:
17:  private:
18:     int itsVal;
19:
20:  } ;
21:
22:  Counter::Counter(int initialValue):
23:  itsVal(initialValue)
24:  { }
25:
26:  Counter::Counter():
27:  itsVal(0)
28:  { }
29:
30:  Counter Counter::Add(const Counter & rhs)
31:  {
32:     return Counter(itsVal+ rhs.GetItsVal());
33:  }
34:
35:  int main()
36:  {
37:     Counter varOne(2), varTwo(4), varThree;
38:     varThree = varOne.Add(varTwo);
39:   cout << "varOne: " << varOne.GetItsVal()<< endl;
40:   cout << "varTwo: " << varTwo.GetItsVal() << endl;
41:   cout << "varThree: " << varThree.GetItsVal() << endl;
42:
43:  return 0;
44:  }
```

РЕЗУЛЬТАТ

```
varOne:   2
varTwo:   4
varThree: 6
```

АНАЛИЗ

Функция `Add()` объявляется в строке 15. В функции задана константная ссылка на `Counter`, представляющая число, которое нужно добавить к текущему объекту. Функция возвращает объект класса `Counter`, представляющий собой результат суммирования, который присваивается операнду слева от оператора присваивания (=), как показано в строке 38. Здесь переменная `varOne` является объектом, `varTwo` — параметр функции `Add()`, а `varThree` — адресный операнд, которому присваивается результат суммирования.

Чтобы создать объект `varThree` без исходной инициализации каким-либо значением, используется конструктор, заданный по умолчанию. Он присваивает объекту `varThree` нулевое значение, как показано в строках 22—24. Иначе эту проблему можно было решить, присвоив нулевое значение конструктору, определенному в строке 11.

Перегрузка оператора суммирования

Тело функции `Add()` показано в строках 30—33. Программа работает, но несколько замысловато. Перегрузка оператора суммирования (+) сделала бы работу класса `Counter` более гармоничной (листинг 10.14).

Листинг 10.14. Перегрузка оператора суммирования

```
1:  // Листинг 10.14.
2:  //Перегрузка оператора суммирования (+)
3:
4:  int
5:  #include <iostream.h>
6:
7:  class Counter
8:  {
9:  public:
10:     Counter();
11:     Counter(int initialValue);
12:     ~Counter(){ }
13:     int GetItsVal()const {   return itsVal; }
14:     void SetItsVal(int x) { itsVal = x; }
15:     Counter operator+ (const Counter &);
16:  private:
17:     int itsVal;
18:  } ;
19:
20:  Counter::Counter(int initialValue):
21:  itsVal(initialValue)
22:  { }
23:
24:  Counter::Counter():
25:  itsVal(0)
26:  { }
27:
```

```
28:   Counter Counter::operator+ (const Counter & rhs)
29:   {
30:      return Counter(itsVal + rhs.GetItsVal());
31:   }
32:
33:   int main()
34:   {
35:      Counter varOne(2), varTwo(4), varThree;
36:      varThree = varOne + varTwo;
37:    cout << "varOne: " << varOne.GetItsVal()<< endl;
38:    cout << "varTwo: " << varTwo.GetItsVal() << endl;
39:    cout << "varThree: " << varThree.GetItsVal() << endl;
40:
41:   return 0;
42:   }
```

РЕЗУЛЬТАТ

```
varOne: 2
varTwo: 4
varThree: 6
```

АНАЛИЗ В строке 15 объявлен оператор суммирования (`operator+`), функция которого определяется в строках 28–31. Сравните эту функцию с объявлением и определением функции `Add()` в предыдущем листинге. Они почти идентичны. В то же время далее в программе эти функции используются совершенно по разному. Посмотрите, следующая запись с оператором (+) выглядит естественней и понятнее

```
varThree = varOne + varTwo;
```

чем строка с функцией `Add()`:

```
varThree = varOne.Add(varTwo);
```

Для компилятора различия не принципиальные, но программа при этом становится более понятной и читабельной, что облегчает работу программиста.

ПРИМЕЧАНИЕ Метод, используемый для перегрузки оператора суммирования (`operator++`), можно применять также с другими операторами, например, оператором вычитания (`operator--`).

Перегрузка операторов с двумя операндами

Операторы с двумя операндами объявляются так же, как и операторы с одним операндом, за исключением того, что функции этих операторов содержат параметры. Параметры представляют собой константные ссылки на объекты таких же типов.

Пример перегрузки оператора суммирования для класса `Counter`:

```
Counter Counter::operator+ (const Counter & rhs);
```

Пример перегрузки оператора вычитания для этого же класса:

```
Counter Counter::operator- (const Counter & rhs);
```

Основные принципы перегрузки операторов

Перегруженные операторы могут быть функциями-членами, как в примерах этой главы, либо задаваться функциями-друзьями, не принадлежащими классу. Более подробно такие операторы будут рассматриваться на занятии 14 во время изучения специальных классов и функций.

Ряд операторов могут быть исключительно членами класса. Это операторы присваивания (=), индексирования ([]), вызова функции (()) и косвенного обращения к члену класса (->).

Оператор индексирования [] будет рассмотрен на следующем занятии, а оператор косвенного обращения к члену класса — на занятии 14 во время изучения дополнительных возможностей указателей.

Ограничения перегрузки операторов

Нельзя перегружать операторы стандартных типов данных (такие как int). Также нельзя изменять установленные приоритеты и ассоциативности операторов. Например, нельзя оператор с одним операндом перегрузить так, чтобы использовать его с двумя операндами. Кроме того, методом перегрузки нельзя создавать новые операторы; например, бинарный оператор умножения (**) не удастся объявить как оператор *возведения в квадрат*.

Количество операндов, которыми может манипулировать оператор, — *важная характеристика* каждого оператора. Различают операторы, используемые с одним операндом (например, оператор инкремента: myValue++), и операторы, для работы которых необходимо указать два операнда (например, оператор суммирования: a+b). Сразу тремя операндами управляет только условный оператор ?, синтаксис использования которого показан в следующем примере: (a > b ? x : y).

Что можно перегружать

Возможность перегрузки операторов — это то новое средство программирования, предоставляемое C++, которое наиболее широко используют (а часто и злоупотребляют им) начинающие программисты. Новичков захватывает азарт присвоения новых интересных функций самым обычным и заурядным операторам. В результате код программы может оказаться непонятным и нечитабельным даже для создателя, а не то что для другого программиста.

Безусловно, если в программе оператор + начнет осуществлять вычитание, а оператор * — суммирование, это может тешить самолюбие начинающего программиста, но профессионал никогда такого не допустит. Вполне можно понять желание использовать оператор + для конкатенации строк и символов, а оператор / для разделения строк, но такая перегрузка операторов таит в себе подводные рифы, на которые может совершенно неожиданно напороться программа во время выполнения. Возможно, было бы не плохо уделить больше внимания особенностям использования перегруженных операторов, но еще лучше начать с формулировки основных предостережений. Прежде всего следует помнить, что основная цель перегрузки операторов состоит в том, чтобы сделать программу эффективнее, а ее код проще и понятнее.

Рекомендуется	Не рекомендуется
Перегружайте операторы, если код программы после этого станет четче и понятнее. **Возвращайте** объекты класса из перегруженных операторов.	**Не увлекайтесь** созданием перегруженных операторов, выполняющих несвойственные им функции.

Оператор присваивания

Четвертая, и последняя, функция, предоставляемая компилятором для работы с объектами, если, конечно, вы не задали никаких дополнительных функций, это функция оператора присваивания (`operator=()`). Этот оператор используется всякий раз, когда нужно присвоить объекту новое значение, например:

```
CAT catOne(5,7);
CAT catTwo(3,4);
//   ... другие строки программы
catTwo = catOne
```

В данном примере создан объект `catOne`, переменной которого `itsAge` присвоено значение `5`, а переменной `itsWeigth` — `7`. Затем создается объект `catTwo` со значениями переменных соответственно `3` и `4`.

Через некоторое время объекту `catTwo` присваиваются значения объекта `catOne`. Что произойдет, если переменная `itsAge` является указателем, и что происходит со старыми значениями переменных объекта `catTwo`?

Работа с переменными-членами, которые хранят свои значения в области динамической памяти, рассматривалась ранее при обсуждении использования конструктора-копировщика (см. также рис. 10.1 и 10.2).

В C++ различают поверхностное и глубинное копирование данных. При поверхностном копировании происходит передача только адреса от одной переменной к другой, в результате чего оба объекта указывают на одни и те же ячейки памяти. В случае глубинного копирования действительно происходит копирование значений переменных из одной области памяти в другую. Различия между этими методами копирования показаны на рис. 10.3.

Все вышесказанное справедливо для присвоения данных. В случае использования оператора присваивания, процесс обмена данных протекает с некоторыми особенностями. Так, объект `catTwo` уже существует вместе со своими переменными, для каждой из которых выделены определенные ячейки памяти. В случае присвоения объекту новых значений предварительно необходимо освободить эти ячейки памяти. Что произойдет, если выполнить присвоение объекта `catTwo` самому себе:

```
catTwo = catTwo
```

Вряд ли такая строка в программе может иметь смысл, но в любом случае программа должна уметь поддерживать подобные ситуации. Дело в том, что присвоение объекта самому себе может произойти по ошибке в случае косвенного обращения к указателю, который ссылается на тот же объект.

Если не предусмотреть поддержку такой ситуации, то оператор присваивания сначала очистит ячейки памяти объекта `catTwo`, а затем попытается присвоить объекту `catTwo` свои собственные значения, которых уже не будет и в помине.

Чтобы предупредить подобную ситуацию, ваш оператор присваивания прежде всего должен определить, не совпадают ли друг с другом объекты по обе стороны от оператора присваивания. Это можно осуществить с помощью указателя this, как показано в листинге 10.15.

Листинг 10.15. Оператор присваивания

```
1:    // Листинг 10.15.
2:    // Конструктор-копировщик
3:
4:    #include <iostream.h>
5:
6:    class CAT
7:    {
8:       public:
9:          CAT();                  // конструктор по умолчанию
10:   // конструктор-копировщик и деструктор пропущены!
11:         int GetAge() const {   return *itsAge; }
12:         int GetWeight() const {   return *itsWeight; }
13:         void SetAge(int age) {   *itsAge = age; }
14:         CAT & operator=(const CAT &);
15:
16:      private:
17:         int *itsAge;
18:         int *itsWeight;
19:   } ;
20:
21:   CAT::CAT()
22:   {
23:       itsAge = new int;
24:      itsWeight = new int;
25:      *itsAge = 5;
26:      *itsWeight = 9;
27:   }
28:
29:
30: CAT & CAT::operator=(const CAT & rhs)
31: {
32:   if (this == &rhs)
33:      return *this;
34:   *itsAge = rhs.GetAge();
35:   *itsWeight = rhs.GetWeight();
36:   return *this;
37: }
38:
39:
40:   int main()
41:   {
42:       CAT frisky;
43:     cout << "frisky's age: " << frisky.GetAge() << endl;
44:     cout << "Setting frisky to 6...\ n";
```

```
45:     frisky.SetAge(6);
46:     CAT whiskers;
47:     cout << "whiskers' age: " << whiskers.GetAge() << endl;
48:     cout << "copying frisky to whiskers...\ n";
49:     whiskers = frisky;
50:     cout << "whiskers' age: " << whiskers.GetAge() << endl;
51:        return 0;
52:  }
```

РЕЗУЛЬТАТ

```
frisky's age: 5
Setting frisky to 6...
whiskers' age: 5
copying frisky to whiskers...
whiskers' age: 6
```

АНАЛИЗ В листинге 10.15 вновь используется класс CAT. Чтобы не повторяться, в данном коде пропущены объявления конструктора-копировщика и деструктора. В строке 14 объявляется оператор присваивания, определение которого представлено в строках 30—37.

В строке 32 выполняется проверка того, не является ли объект, которому будет присвоено значение, тем же самым объектом класса CAT, чье значение будет присвоено. Чтобы проверить это, сравниваются адреса в указателях rhs и this.

Безусловно, оператор присваивания (=) может быть произвольно перегружен таким образом, чтобы отвечать представлениям программиста, что означает равенство объектов.

Операторы преобразований

Что происходит при попытке присвоить значение переменой одного из базовых типов, таких как int или unsigned short, объекту класса, объявленного пользователем? В листинге 10.16 мы опять вернемся к классу Counter и попытаемся присвоить объекту этого класса значение переменной типа int.

ПРЕДУПРЕЖДЕНИЕ Листинг 10.16 не компилируйте!

Листинг 10.16. Попытка присвоить объекту класса Counter значение переменной типа int

```
1:    // Листинг 10.16.
2:    // Эту программу не компилируйте!
3:
4:    int
5:    #include <iostream.h>
6:
7:    class Counter
```

```
8:    {
9:       public:
10:       Counter();
11:       ~Counter(){ }
12:       int GetItsVal()const {   return itsVal; }
13:       void SetItsVal(int x) { itsVal = x; }
14:       private:
15:       int itsVal;
16:
17:     } ;
18:
19:     Counter::Counter():
20:     itsVal(0)
21:     { }
22:
23:  int main()
24:  {
25:     int theShort = 5;
26:     Counter theCtr = theShort;
27:     cout << "theCtr: " << theCtr.GetItsVal() << endl;
28:  return 0;
29:  }
```

РЕЗУЛЬТАТ Компилятор покажет сообщение об ошибке, поскольку не сможет преобразовать тип `int` в `Counter`.

АНАЛИЗ Класс `Counter`, определенный в строках 7–17, содержит только один конструктор, заданный по умолчанию. В нем не определено ни одного метода преобразования данных типа `int` в тип `Counter`, поэтому компилятор обнаруживает ошибку в строке 26. Компилятор ничего не сможет поделать, пока не получит четких инструкций, что данные типа `int` необходимо взять и присвоить переменной-члену `itsVal`.

В листинге 10.17 эта ошибка исправлена с помощью оператора преобразования типов. Определен конструктор, который создает объект класса `Counter` и присваивает ему полученное значение типа `int`.

Листинг 10.17. Преобразование `int` в `Counter`

```
1:    // Листинг 10.17.
2:    // Использование конструктора в качестве оператора преобразования типа
3:
4:    int
5:    #include <iostream.h>
6:
7:    class Counter
8:    {
9:       public:
10:       Counter();
11:       Counter(int val);
12:       ~Counter(){ }
13:       int GetItsVal()const {   return itsVal; }
```

```
14:        void SetItsVal(int x) { itsVal = x; }
15:        private:
16:        int itsVal;
17:
18:     } ;
19:
20:     Counter::Counter():
21:     itsVal(0)
22:     { }
23:
24:  Counter::Counter(int val):
25:  itsVal(val)
26:  { }
27:
28:
29:  int main()
30:  {
31:     int theShort = 5;
32:     Counter theCtr = theShort;
33:     cout << "theCtr: " << theCtr.GetItsVal() << endl;
34:     return 0;
35:  }
```

РЕЗУЛЬТАТ `the Ctr: 5`

АНАЛИЗ Важные изменения произошли в строке 11, где конструктор перегружен таким образом, чтобы принимать значения типа `int`, а также в строках 24–26, где данный конструктор применяется. В результате выполнения конструктора переменной-члену класса `Counter` присваивается значение типа `int`.

Для присвоения значения программа обращается к конструктору, в котором присваиваемое значение передается в качестве аргумента. Процесс осуществляется в несколько шагов.

Шаг 1: создание переменной класса `Counter` *с именем* `theCtr`.

Это то же самое, что записать: `int x = 5`, где создается целочисленная переменная `x` и ей присваивается значение `5`. Но в нашем случае создается объект `theCtr` класса `Counter`, который инициализируется переменной `theShort` типа `short int`.

Шаг 2: присвоение объекту `theCtr` *значения переменной* `theShort`.

Но переменная относится к типу `short`, а не `Counter`! Первое, что нужно сделать, — это преобразовать ее к типу `Counter`. Компилятор может делать некоторые преобразования автоматически, но ему нужно точно указать, чего от него хотят. Именно для инструктирования компилятора создается конструктор класса `Counter`, который содержит единственный параметр, например типа `short`:

```
class Counter
{
Counter (short int x);
//   ...
};
```

Данный конструктор создает объект класса Counter, используя временный безымянный объект этого класса, способный принимать значения типа short. Чтобы сделать этот процесс более наглядным, предположим, что для значений типа short создается не безымянный объект, а объект класса Counter с именем wasShort.

Шаг 3: присвоение значения объекта wasShort *объекту* theCtr, *что эквивалентно записи* "theCtr = wasShort";

На этом шаге временный объект wasShort, созданный при запуске конструктора, замещается на постоянный объект theCtr, принадлежащий классу Counter. Другими словами, значение временного объекта присваивается объекту theCtr.

Чтобы понять, как происходит этот процесс, следует четко уяснить принципы работы, справедливые для ВСЕХ перегруженных операторов, определенных с помощью ключевого слова operator. В случае с операторами с двумя операндами (такими как = или +) находящийся справа операнд объявляется как параметр функции оператора, заданной в конструкторе. Так, выражение

```
a = b
```

объявляется как

```
a.operator=(b);
```

Что произойдет, если изменить порядок присвоения, как в следующем примере:

```
1:   Counter theCtr(5);
2:   int theShort = theCtr;
3:   cout << "theShort : " << theShort   << endl;
```

Вновь компилятор покажет сообщение об ошибке. Хотя сейчас компилятор уже знает, как создать временный объект Counter для принятия значения типа int, но он не знает, как осуществить обратный процесс.

Операторы преобразования типов

Чтобы разрешить эту и подобные ей проблемы, в C++ есть специальные операторы преобразования типов, которые можно добавить в пользовательский класс. В результате появится возможность явного преобразования типа пользовательского класса к любому из базовых типов данных языка программирования. Реализация этой возможности показана в листинге 10.18. Только одно замечание: в операторах преобразований не задается тип возврата. Даже если их работа напоминает возврат функции, в действительности они возвращают преобразованное значение.

Листинг 10.18. Преобразование данных типа Counter **в тип** unsigned short()

```
1:   #include <iostream.h>
2:
3:   class Counter
4:   {
5:   public:
6:     Counter();
7:     Counter(int val);
8:     ~Counter(){ }
9:     int GetItsVal()const {   return itsVal; }
10:     void SetItsVal(int x) { itsVal = x; }
```

```
11:     operator unsigned short();
12:   private:
13:     int itsVal;
14:
15:   } ;
16:
17:   Counter::Counter():
18:   itsVal(0)
19:   { }
20:
21: Counter::Counter(int val):
22: itsVal(val)
23: { }
24:
25: Counter::operator unsigned short ()
26: {
27:   return ( int (itsVal) );
28: }
29:
30: int main()
31: {
32:   Counter ctr(5);
33:   int theShort = ctr;
34:   cout << "theShort: " << theShort << endl;
35:   return 0;
36: }
```

РЕЗУЛЬТАТ

```
theShort: 5
```

АНАЛИЗ

В строке 11 объявляется оператор преобразования типа. Обратите внимание, что в нем не указан тип возврата. Функция оператора преобразования выполняется в строках 25–28. В строке 27 возвращается значение объекта `itsVal`, преобразованное в тип `int`.

Теперь компилятор знает, как присвоить объекту класса значение типа `int` и как возвратить из объекта класса текущее значение, чтобы присвоить его внешней переменной типа `int`.

Резюме

Сегодня вы научились перегружать функции-члены пользовательского класса. Вы также узнали, как передавать в функции значения, заданные по умолчанию, и в каких случаях вместо значений по умолчанию лучше использовать перегруженные функции.

Перегрузка конструкторов класса позволяет более гибко управлять классами и создавать новые классы, содержащие объекты других классов. Лучше всего инициализацию объектов класса осуществлять во время инициализации конструктора, вместо того чтобы делать это в теле конструктора.

Конструктор-копировщик и оператор присваивания по умолчанию предоставляются компилятором, если в классе эти объекты не были созданы пользователем. Но при использовании копировщика и оператора присваивания, заданных по умолчанию, осуществляется только поверхностное копирование данных. В тех классах, где в числе членов класса используются указатели на области динамической памяти, вместо поверхностного копирования лучше использовать глубинное, при котором копируемые данные размещаются по новым адресам.

Хотя в языке C++ можно произвольно перегружать все операторы, настоятельно рекомендуем не создавать таких операторов, функции которых противоречат их традиционному использованию. Кроме того, невозможно изменить ассоциативность оператора, а также создавать собственные операторы, не представленные в языке C++.

Указатель `this` ссылается на текущий объект и является невидимым параметром для всех функций-членов. Разыменованный указатель `this` часто возвращается перегруженными операторами.

Операторы преобразования типов позволяют настраивать классы для использования в выражениях, осуществляющих обмен данными разных типов. Данные операторы являются исключением из правила, состоящего в том, что все функции возвращают явные значения, как, например, конструктор и деструктор. В данных операторах тип возврата не устанавливается.

Вопросы и ответы

Зачем использовать значения, заданные по умолчанию, если можно перегрузить функцию?

Проще иметь дело с одной функцией, чем с двумя. Кроме того, зачастую проще понять работу функции, использующей значения, заданные по умолчанию, чем каждый раз внимательно изучать тело функции, чтобы понять ее назначение. Кроме того, обновление одной версии функции без обновления другой версии часто бывает причиной ошибок в работе программы.

Почему бы тогда постоянно не использовать только значения, заданные по умолчанию?

Перегрузка функций предоставляет ряд возможностей, которые нельзя реализовать, используя только значения, заданные по умолчанию. Например, изменять не только число параметров в списке, но и их типы.

Какие переменные-члены следует инициализировать одновременно с инициализацией конструктора, а какие оставлять для тела конструктора?

Используйте следующее простое правило: одновременно с конструктором следует инициализировать как можно больше переменных-членов. Только некоторые из них, такие как переменные для текущих вычислений и управления выводом на печать следует инициализировать в теле конструктора.

Может ли перегруженная функция содержать параметры, заданные по умолчанию?

Конечно. Нет никакой причины, по которой не следовало бы использовать это мощное средство. Одна или несколько версий перегруженных функций могут иметь собственные значения, заданные по умолчанию. При установке значений по умолчанию для перегруженных функций нужно следовать тем же общим правилам, что и при установке значений по умолчанию для обычных функций.

Почему одни функции-члены определяются в описании класса, а другие нет?

Если функция определяется в описании класса, то далее она используется в режиме `inline`. Впрочем, встраивание кода функции по месту вызова происходит только в

том случае, если функция достаточно простая. Также следует отметить, что задать встраивание кода функции-члена в код программы можно с помощью ключевого слова `inline`, даже если эта функция была описана отдельно от класса.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Если вы перегрузили функцию-член, как потом можно будет различить разные варианты функции?
2. Какая разница между определением и объявлением?
3. Когда вызывается конструктор-копировщик?
4. Когда вызывается деструктор?
5. Чем отличается конструктор-копировщик от оператора присваивания (=)?
6. Что представляет собой указатель `this`?
7. Как отличается перегрузка операторов предварительного и последующего действия?
8. Можно ли перегрузить `operator+` для переменных типа `short int`?
9. Допускается ли в C++ перегрузка `operator++` таким образом, чтобы он выполнял в классе операцию декремента?
10. Как устанавливается тип возврата в объявлениях функций операторов преобразования типов?

Упражнения

1. Представьте объявление класса `SimpleCircle` с единственной переменой-членом `itsRadius`. В классе должны использоваться конструктор и деструктор, заданные по умолчанию, а также метод установки радиуса.
2. Используя класс, созданный в упражнении 1, с помощью конструктора, заданного по умолчанию, инициализируйте переменную `itsRadius` значением 5.
3. Добавьте в класс новый конструктор, который присваивает значение своего параметра переменной `itsRadius`.
4. Перегрузите операторы преинкремента и постинкремента для использования в вашем классе `SimpleCircle` с переменной `itsRadius`.
5. Измените `SimpleCircle` таким образом, чтобы сохранять `itsRadius` в динамической области памяти и фиксировать существующие методы.
6. Создайте в классе `SimpleCircle` конструктор-копировщик.
7. Перегрузите в классе `SimpleCircle` оператор присваивания.

8. Напишите программу, которая создает два объекта класса `SimpleCircle`. Для создания одного объекта используйте конструктор, заданный по умолчанию, а второму экземпляру при объявлении присвойте значение 9. С каждым из объектов используйте оператор инкремента и выведите полученные значения на печать. Наконец, присвойте значение одного объекта другому объекту и выведите результат на печать.

9. **Жучки**: что неправильно в следующем примере использования оператора присваивания?

```
SQUARE SQUARE ::operator=(const SQARE & rhs)
{
   itsSide = new int;
   *itsSide = rgs.GetSide();
   return *this;
}
```

10. **Жучки**: что неправильно в следующем примере использования оператора суммирования?

```
VeryShort VeryShort ::operator+ (const VeryShort& rhs)
{
   itsval += rhs.GetItsVal();
   return *this;
}
```

Наследование

Фундаментальной основой человеческого мышления является поиск, выявление и построение взаимоотношений между различными концепциями. Чтобы постичь хитросплетения отношений между вещами и явлениями, мы используем иерархические построения, матрицы, сети и прочие средства визуализации. Чтобы лучше выразить суть отношений между объектами, в C++ используется иерархическая система наследования. Сегодня вы узнаете:

- Что представляет собой наследование
- Как произвести один класс из другого
- Что такое защищенный доступ и как его использовать
- Что такое виртуальные функции

Что такое наследование

Что такое собака? Что вы видите, когда смотрите на своего питомца? Я вижу четыре лапы, обслуживающие зубастую пасть. Биолог увидит систему взаимодействующих органов, физик — стройную систему атомов и совокупность разных видов энергии, а ученый, занимающийся систематикой млекопитающих, — типичного представителя вида *Canis familiaris*.

Каждый смотрит на объект со своей точки зрения, но сегодня нас будет интересовать последнее утверждение, а именно: собака является представителем семейства волчьих, класса млекопитающих и т.д. С точки зрения систематики любой объект живой природы рассматривается в плане принадлежности одной системе иерархических таксонов: царству, типу, классу, отряду, семейству, роду и виду.

Иерархия представляет собой вид отношений подчиненности типа *принадлежности частного общему*. Так, человек *является* видом приматов. Подобный тип отношений можно видеть повсюду. Грузовик *является* видом машин, а машина, в свою очередь, *является* видом транспортных средств. Пирожное *является* видом сладких блюд, а сладкие блюда *являются* видом пищи.

Когда мы говорим, что нечто является видом чего-то, то подразумеваем большую детализацию объявления объекта. Так, отмечая, что машина — это вид транспортных средств, мы из всевозможных средств передвижения (от повозки до самолета) выбираем только четырехколесные устройства с двигателем.

Иерархия и наследование

Говоря о собаке, как представителе класса млекопитающих, мы подразумеваем, что она наследует все признаки, общие для класса млекопитающих. Поскольку собака — млекопитающее, можно предположить, что это подвижный вид животных, дышащих воздухом. Все млекопитающие по определению двигаются и дышат воздухом. Если определить некий объект как собаку, это добавит к объявлению способность вилять хвостом, грызть рукопись книги, которую я как раз собрался нести в редакцию, бегать по дому и лаять, когда я сплю... о извините, куда меня занесло! Ну так вот, продолжим.

В свою очередь, собак можно разделить на служебных, спортивных и охотничьих. Потом можно пойти дальше и описать породу собаки: спаниель, лабрадор и т.д.

Таким образом, мы можем сказать, например, что фокстерьер — это порода охотничьих собак, в которой представлены все признаки, общие для собак вообще, а также все признаки, общие для млекопитающих и т.д., включая признаки всех таксонов, к которым относится фокстерьер. Пример такой иерархии показан на рис. 11.1, где стрелками связаны категории более низкого уровня с категориям следующего порядка.

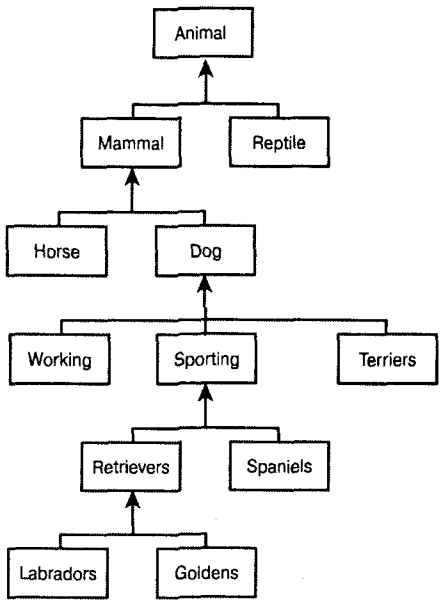


Рис. 11.1. Иерархия млекопитающих

В C++ иерархичность реализована в концепции классов, где один класс может *происходить*, или *наследоваться* от класса более высокого уровня. В наследовании классов реализуются принципы их иерархической подчиненности. Предположим, мы производим новый класс `Dog` (Собака) от класса `Mammal` (Млекопитающее). Другими словами, класс `Mammal` является базовым для класса `Dog`. Точно так же, как описание вида собака несет в себе признаки, детализирующие описание млекопитающих в целом, так и класс `Dog` содержит ряд методов и данных, дополняющих методы и данные, которые представлены в классе `Mammal`.

Как правило, с базовым классом связано несколько производных классов. Поскольку собаки, кошки и лошади являются представителями млекопитающих, то с точки зрения C++ можно сказать, что все эти классы произведены от класса Mammal.

Царство животных

Чтобы более наглядно раскрыть смысл наследования классов, рассмотрим эту тему на примере отношений между многочисленными представителями животного мира. Представим себе, что программисту поступил заказ на создание детской игры "*Ферма*".

Когда вы приступите к созданию животных, обитающих на ферме, включая лошадей, коров, собак, кошек, овец и т.д., вам потребуется снабдить каждый их класс такими методами, благодаря которым они смогут вести себя на экране так, как этого ожидает ребенок. Но на данном этапе каждый метод снабжен только функцией вывода на печать. Это общая практика программирования, когда сначала выполняется только формулировка набора методов, а детальная проработка их откладывается на потом. Вы вправе использовать все примеры программ, приведенные в этой главе, как основу для дальнейшей доработки с тем, чтобы все животные вели себя так, как вам хочется.

Синтаксис наследования классов

Для создания нового производного класса используется ключевое слово class, после которого указывается имя нового класса, двоеточие, тип объявления класса (public или какой-нибудь другой), а затем имя базового класса, как в следующем примере:

```
class Dog : public Mammal
```

Типы наследования классов рассматриваются далее в этой книге. Пока будем использовать только *открытое наследование*. Класс, из которого производится новый класс, должен быть объявлен раньше, иначе компилятор покажет сообщение об ошибке. Пример наследования класса Dog от класса Mammal показан в листинге 11.1.

Листинг 11.1. Простое наследование

```
1:  //Листинг 11.1. Простое наследование
2:
3:  #include <iostream.h>
4:  enum BREED {  GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB } ;
5:
6:  class Mammal
7:  {
8:  public:
9:    // Конструкторы
10:     Mammal();
11:     ~Mammal();
12:
13:     // Методы доступа к данным
14:     int GetAge()const;
15:     void SetAge(int);
16:     int GetWeight() const;
17:     void SetWeight();
18:
19:     // Другие методы
```

```
20:     void Speak() const;
21:     void Sleep() const;
22:
23:
24:  protected:
25:     int itsAge;
26:     int itsWeight;
27:  } ;
28:
29:  class Dog : public Mammal
30:  {
31:  public:
32:
33:     // Конструкторы
34:     Dog();
35:     ~Dog();
36:
37:     // Методы доступа к данным
38:     BREED GetBreed() const;
39:     void SetBreed(BREED);
40:
41:     // Другие методы
42:    WagTail();
43:    BegForFood();
44:
45:  protected:
46:     BREED itsBreed;
47:  } ;
```

РЕЗУЛЬТАТ Данная программа ничего не выводит на экран, так как пока содержит только объявления и установки классов. Никаких функций эта программа пока не выполняет.

АНАЛИЗ Класс `Mammal` объявляется в строках 6–27. Обратите внимание, что класс `Mammal` не производится ни от какого другого класса, хотя в реальной жизни можно сказать, что класс млекопитающих производится от класса животных. Но в C++ всегда отображается не весь окружающий мир, а лишь модель некоторой его части. Действительность слишком сложна и разнообразна, чтобы отобразить ее в одной, даже очень большой программе. Профессионализм состоит в том, чтобы с помощью относительно простой модели воспроизвести объекты, которые будут максимально соответствовать своим реальным эквивалентам.

Иерархическая структура нашего мира берет свое начало неизвестно откуда, но наша конкретная программа начинается с класса `Mammal`. В связи с этим некоторые переменные-члены, которые необходимы для работы базового класса, должны быть представлены в объявлении этого класса. Например, все животные независимо от вида и породы имеют возраст и вес. Если бы класс `Mammal` производился от класса `Animals`, то можно было бы ожидать, что он унаследует эти атрибуты. При этом атрибуты базового класса становятся атрибутами произведенного класса.

Чтобы облегчить работу с программой и ограничить ее сложность разумными рамками, в классе Mammal представлены только шесть методов: четыре метода доступа, а также функции Speak() и Sleep().

В строке 29 класс Dog наследуется из класса Mammal. Все объекты класса Dog будут иметь три переменные-члена: itsAge, itsWeight и itsBreed. Обратите внимание, что в объявлении класса Dog не указаны переменные itsAge и itsWeight. Объекты класса Dog унаследовали эти переменные из класса Mammal вместе с методами, объявленными в классе Mammal, за исключением копировщика, конструктора и деструктора.

Закрытый или защищенный

Возможно, вы заметили, что в строках 24 и 45 листинга 11.1 используется новое ключевое слово protected. До сих пор данные класса определялись с ключевым словом private. Но члены класса, объявленные как private, недоступны для наследования. Конечно, можно было в предыдущем листинге определить переменные-члены itsAge и itsWeight как public, но это нежелательно, поскольку прямой доступ к этим переменным получили бы все другие классы программы.

Нашу цель можно сформулировать следующим образом: сделать переменную-член видимой для этого класса и для всех классов, произведенных от него. Именно таковыми являются защищенные данные, определяемые ключевым словом protected. Защищенные данные доступны для всех произведенных классов, но недоступны для всех внешних классов.

Обобщим: существует три спецификатора доступа — public, protected и private. Если в функцию передаются объекты класса, то она может использовать данные всех переменных-членов и функций-членов, объявленных со спецификатором public. Функция-член класса, кроме того, может использовать все закрытые данные этого класса (объявленные как private) и защищенные данные любого другого класса, произведенного от этого класса (объявленные как protected).

Так, в нашем примере функция Dog::WagTail() может использовать значение закрытой переменной itsBreed и все переменные класса Mammal, объявленные как public и protected.

Даже если бы класс Dog был произведен не от класса Mammal непосредственно, а от какого-нибудь промежуточного класса (например, DomesticAnimals), все равно из класса Dog сохранился бы доступ к защищенным данным класса Mammal, правда только в том случае, если класс Dog и все промежуточные классы объявлялись как public. Наследование класса с ключевым словом private будет рассматриваться на занятии 15.

В листинге 11.2 показано создание объекта в классе Dog с доступом ко всем данным и функциям этого типа.

Листинг 11.2. Использование унаследованных объектов

```
1:   //Листинг 11.2. Использование унаследованных объектов
2:
3:   #include <iostream.h>
4:   enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB } ;
5:
6:   class Mammal
7:   {
```

```
8:  public:
9:    // Конструкторы
10:     Mammal():itsAge(2), itsWeight(5){ }
11:     ~Mammal(){ }
12:
13:     //Методы доступа
14:     int GetAge()const {   return itsAge; }
15:     void SetAge(int age) {   itsAge = age; }
16:     int GetWeight() const {   return itsWeight; }
17:     void SetWeight(int weight) {   itsWeight = weight; }
18:
19:     //Другие методы
20:    void Speak()const { cout << "Mammal sound!\ n"; }
21:    void Sleep()const { cout << "shhh. I'm sleeping.\ n"; }
22:
23:
24:  protected:
25:     int itsAge;
26:     int itsWeight;
27:  } ;
28:
29:  class Dog : public Mammal
30:  {
31:  public:
32:
33:     // Конструкторы
34:     Dog():itsBreed(GOLDEN){ }
35:     ~Dog(){ }
36:
37:     // Методы доступа
38:     BREED GetBreed() const {   return itsBreed; }
39:     void SetBreed(BREED breed) {   itsBreed = breed; }
40:
41:     // Другие методы
42:    void WagTail() const { cout << "Tail wagging...\ n"; }
43:    void BegForFood() const { cout << "Begging for food...\ n"; }
44:
45:  private:
46:    BREED itsBreed;
47:  } ;
48:
49:  int main()
50:  {
51:    Dog fido;
52:    fido.Speak();
53:    fido.WagTail();
54:    cout << "Fido is " << fido.GetAge() << " years old\ n";
55:   return 0;
56:   }
```

РЕЗУЛЬТАТ

```
Mammal sound!
Tail wagging...
Fido is 2 years old
```

АНАЛИЗ

В строках 6–27 объявляется класс `Mammal` (для краткости тела функций вставлены по месту их вызовов). В строках 29–47 из класса `Mammal` производится класс `Dog`. В результате объекту `Fido` этого класса доступны как функция производного класса `WagTail()`, так и функции базового класса `Speak()` и `Sleep()`.

Конструкторы и деструкторы

Объекты класса `Dog` одновременно являются объектами класса `Mammal`. В этом суть иерархических отношений между классами. Когда в классе `Dog` создается объект `Fido`, то для этого из класса `Mammal` вызывается базовый конструктор, называемый первым. Затем вызывается конструктор класса `Dog`, который завершает создание объекта. Поскольку объект `Fido` не снабжен никакими параметрами, в обоих случаях вызывается конструктор, заданный по умолчанию. Объект `Fido` не существует до тех пор, пока полностью не будет завершено его создание с использованием обоих конструкторов класса `Mammal` и класса `Dog`.

При удалении объекта `Fido` из памяти компьютера сначала вызывается деструктор класса `Dog`, а затем деструктор класса `Mammal`. Каждый деструктор удаляет ту часть объекта, которая была создана соответствующим конструктором производного или базового классов. Не забудьте удалить из памяти объект, если он больше не используется, как показано в листинге 11.3.

Листинг 11.3. Вызов конструктора и деструктора

```
1:   //Листинг 11.3. Вызов конструктора и деструктора.
2:
3:   #include <iostream.h>
4:   enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB } ;
5:
6:   class Mammal
7:   {
8:   public:
9:     // конструкторы
10:      Mammal();
11:      ~Mammal();
12:
13:      //Методы доступа
14:      int GetAge() const {   return itsAge; }
15:      void SetAge(int age) {   itsAge = age; }
16:      int GetWeight() const {   return itsWeight; }
17:      void SetWeight(int weight) {   itsWeight = weight; }
18:
19:      //Другие методы
20:    void Speak() const { cout << "Mammal sound!\ n"; }
21:    void Sleep() const { cout << "shhh. I'm sleeping.\ n"; }
22:
```

```
23:
24:    protected:
25:       int itsAge;
26:       int itsWeight;
27:    } ;
28:
29:    class Dog : public Mammal
30:    {
31:    public:
32:
33:       // Конструкторы
34:       Dog();
35:       ~Dog();
36:
37:       // Методы доступа
38:       BREED GetBreed() const {    return itsBreed; }
39:       void SetBreed(BREED breed) {    itsBreed = breed; }
40:
41:       // Другие методы
42:     void WagTail() const { cout << "Tail wagging...\ n"; }
43:     void BegForFood() const { cout << "Begging for food...\ n"; }
44:
45:    private:
46:       BREED itsBreed;
47:    } ;
48:
49:    Mammal::Mammal():
50:    itsAge(1),
51:    itsWeight(5)
52:    {
53:     cout << "Mammal constructor...\ n";
54:   }
55:
56:   Mammal::~Mammal()
57:   {
58:     cout << "Mammal destructor...\ n";
59:   }
60:
61:   Dog::Dog():
62:   itsBreed(GOLDEN)
63:   {
64:     cout << "Dog constructor...\ n";
65:   }
66:
67:   Dog::~Dog()
68:   {
69:     cout << "Dog destructor...\ n";
70:   }
71:   int main()
72:   {
```

```
73:    Dog fido;
74:    fido.Speak();
75:    fido.WagTail();
76:    cout << "Fido is " << fido.GetAge() << " years old\ n";
77:   return 0;
78:   }
```

РЕЗУЛЬТАТ

```
Mammal constructor...
Dog constructor...
Mammal sound!
Tail wagging...
Fido is 1 years old
Dog destructor...
Mammal destructor...
```

АНАЛИЗ Листинг 11.3 напоминает листинг 11.2 за тем исключением, что вызов конструктора и деструктора сопровождается сообщением об этом на экране. Сначала вызывается конструктор класса `Mammal`, затем класса `Dog`. После этого объект класса `Dog` полноценно существует и можно использовать все его методы. Когда выполнение программы выходит за область видимости объекта `Fido`, вызывается пара деструкторов, сначала из класса `Dog`, а затем из класса `Mammal`.

Передача аргументов в базовые конструкторы

Предположим, нужно перегрузить конструкторы, заданные по умолчанию в классах `Mammal` и `Dog`, таким образом, чтобы первый из них сразу присваивал новому объекту определенный возраст, а второй — породу. Как передать в конструктор класса `Mammal` значения возраста и веса животного? Что произойдет, если вес не будет установлен конструктором класса `Mammal`, зато его установит конструктор класса `Dog`?

Чтобы выполнить инициализацию базового класса, необходимо записать имя класса, после чего указать параметры, ожидаемые базовым классом, как показано в листинге 11.4.

Листинг 11.4. Перегрузка конструкторов в производных классах

```
1:   //Листинг 11.4. Перегрузка конструкторов в производных классах
2:
3:   #include <iostream.h>
4:   enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB } ;
5:
6:   class Mammal
7:   {
8:   public:
9:     // Конструкторы
10:      Mammal();
11:      Mammal(int age);
12:      ~Mammal();
13:
14:      // Методы доступа
15:      int GetAge() const {   return itsAge; }
```

```
16:     void SetAge(int age) {   itsAge = age; }
17:     int GetWeight() const {   return itsWeight; }
18:     void SetWeight(int weight) {   itsWeight = weight; }
19:
20:     //Другие методы
21:    void Speak() const { cout << "Mammal sound!\ n"; }
22:    void Sleep() const { cout << "shhh. I'm sleeping.\ n"; }
23:
24:
25:   protected:
26:     int itsAge;
27:     int itsWeight;
28:   } ;
29:
30:   class Dog : public Mammal
31:   {
32:   public:
33:
34:     // Конструкторы
35:     Dog();
36:     Dog(int age);
37:     Dog(int age, int weight);
38:     Dog(int age, BREED breed);
39:     Dog(int age, int weight, BREED breed);
40:     ~Dog();
41:
42:     // Методы доступа
43:     BREED GetBreed() const {   return itsBreed; }
44:     void SetBreed(BREED breed) {   itsBreed = breed; }
45:
46:     // Другие методы
47:    void WagTail() const { cout << "Tail wagging...\ n"; }
48:    void BegForFood() const { cout << "Begging for food...\ n"; }
49:
50:   private:
51:     BREED itsBreed;
52:   } ;
53:
54:   Mammal::Mammal():
55:   itsAge(1),
56:   itsWeight(5)
57:   {
58:    cout << "Mammal constructor...\ n";
59:  }
60:
61:  Mammal::Mammal(int age):
62:  itsAge(age),
63:  itsWeight(5)
64:  {
65:    cout << "Mammal(int) constructor...\ n";
```

```
66:  }
67:
68:  Mammal::~Mammal()
69:  {
70:    cout << "Mammal destructor...\ n";
71:  }
72:
73:  Dog::Dog():
74:  Mammal(),
75:  itsBreed(GOLDEN)
76:  {
77:    cout << "Dog constructor...\ n";
78:  }
79:
80:  Dog::Dog(int age):
81:  Mammal(age),
82:  itsBreed(GOLDEN)
83:  {
84:    cout << "Dog(int) constructor...\ n";
85:  }
86:
87:  Dog::Dog(int age, int weight):
88:  Mammal(age),
89:  itsBreed(GOLDEN)
90:  {
91:    itsWeight = weight;
92:    cout << "Dog(int, int) constructor...\ n";
93:  }
94:
95:  Dog::Dog(int age, int weight, BREED breed):
96:  Mammal(age),
97:  itsBreed(breed)
98:  {
99:    itsWeight = weight;
100:   cout << "Dog(int, int, BREED) constructor...\ n";
101:  }
102:
103:  Dog::Dog(int age, BREED breed):
104:  Mammal(age),
105:  itsBreed(breed)
106:  {
107:   cout << "Dog(int, BREED) constructor...\ n";
108:  }
109:
110:  Dog::~Dog()
111:  {
112:   cout << "Dog destructor...\ n";
113:  }
114:  int main()
115:  {
```

```
116:    Dog fido;
117:    Dog rover(5);
118:    Dog buster(6,8);
119:    Dog yorkie (3,GOLDEN);
120:    Dog dobbie (4,20,DOBERMAN);
121:    fido.Speak();
122:    rover.WagTail();
123:    cout << "Yorkie is " << yorkie.GetAge() << " years old\ n";
124:    cout << "Dobbie weighs ";
125:    cout << dobbie.GetWeight() << " pounds\ n";
126:    return 0;
127:  }
```

ПРИМЕЧАНИЕ

Для удобства дальнейшего анализа строки вывода программы на экран пронумерованы.

РЕЗУЛЬТАТ

```
1: Mammal constructor...
2: Dog constructor...
3: Mammal(int) constructor...
4: Dog(int) constructor...
5: Mammal(int) constructor...
6: Dog(int, int) constructor...
7: Mammal(int) constructor...
8: Dog(int, BREED) constructor....
9: Mammal(int) constructor...
10: Dog(int, int, BREED) constructor...
11: Mammal sound!
12: Tail wagging...
13: Yorkie is 3 years old.
14: Dobbie weighs 20 pounds.
15: Dog destructor...
16: Mammal destructor...
17: Dog destructor...
18: Mammal destructor...
19: Dog destructor...
20: Mammal destructor...
21: Dog destructor...
22: Mammal destructor...
23: Dog destructor...
24: Mammal destructor...
```

АНАЛИЗ

В листинге 11.4 конструктор класса `Mammal` перегружен в строке 11 таким образом, чтобы принимать целочисленные значения возраста животного. В строках 61–66 происходит инициализация переменной `itsAge` значением `5`, переданным в параметре конструктора.

В классе `Dog` в строках 35–39 создается пять перегруженных конструкторов. Первый — это конструктор, заданный по умолчанию. Второй принимает возраст и ис-

пользует для этого тот же параметр, что и конструктор класса Mammal. Третий принимает возраст и вес, четвертый — возраст и породу, а пятый — возраст, вес и породу.

Обратите внимание, что в строке 74 конструктор по умолчанию класса Dog вызывает конструктор по умолчанию класса Mammal. Хотя в этом нет необходимости, но данная запись лишний раз документирует намерение вызвать именно базовый конструктор, не содержащий параметров. Базовый конструктор будет вызван в любом случае, но в данной строке это было сделано явно.

В строках 80–85 выполняется конструктор класса Dog, который принимает одно целочисленное значение. Во время инициализации (строки 81 и 82) возраст принимается из базового класса в виде параметра, после чего присваивается значение породы.

Другой конструктор класса Dog выполняется в строках 87–93. Этот конструктор принимает два параметра. Первое значение вновь инициализируется обращением к соответствующему конструктору базового класса, тогда как второе берется из переменной базового класса itsWeight самим конструктором класса Dog. Обратите внимание, что присвоение значения переменной базового класса не может осуществляться на стадии инициализации конструктора произведенного класса. Поскольку в классе Mammal нет конструктора, присваивающего значение этой переменной, то присвоение значения должно выполняться в теле конструктора класса Dog.

Самостоятельно проанализируйте работу остальных конструкторов в программе, чтобы закрепить полученные знания. Обращайте внимание, какие переменные можно инициализировать одновременно с инициализацией конструктора, а в каких случаях инициализацию следует выполнять в теле конструктора.

Для удобства анализа работы программы строки вывода были пронумерованы. Первые две строки вывода соответствуют инициализации объекта Fido с помощью конструкторов, заданных по умолчанию.

Строки 3 и 4 соответствуют созданию объекта rover, а строки 5 и 6 — объекта buster. Обратите внимание, что в последнем случае из конструктора класса Dog с двумя целочисленными параметрами происходит вызов конструктора класса Mammal, содержащего один целочисленный параметр.

После создания всех объектов программа использует их и наконец выходит за область видимости этих объектов. Удаление каждого объекта сопровождается обращением к деструктору класса Dog, после чего следует обращение к деструктору класса Mammal.

Замещение функций

Объект класса Dog имеет доступ ко всем функциям-членам класса Mammal, а также к любой функции-члену, чье объявление добавлено в класс Dog, например к функции WagTaill(). Но кроме этого, базовые функции могут быть замещены в производном классе. Под замещением базовой функции понимают изменение ее выполнения в производном классе для объектов, созданных в этом классе.

Если в производном классе создается функция с таким же возвратом и сигнатурой как и в базовом классе, но выполняемая особым образом, то имеет место замещение метода.

В случае замещения функций должно сохраняться соответствие между типом возврата и сигнатурой функций в базовом классе. Под *сигнатурой* понимают установки, заданные в прототипе функции, включая ее имя, список параметров и, в случае использования, ключевое слово const.

В листинге 11.5 показано замещение в классе Dog функции Speak(), объявленной в классе Mammal. Для экономии места знакомые по предыдущим листингам объявления методов доступа в этом примере были опущены.

Листинг 11.5. Замещение метода базового класса в производном классе

```
1:  //Листинг 11.5. Замещение метода базового класса в производном классе
2:
3:  #include <iostream.h>
4:  enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB } ;
5:
6:  class Mammal
7:  {
8:  public:
9:    // Конструкторы
10:   Mammal() { cout << "Mammal constructor...\ n"; }
11:   ~Mammal() { cout << "Mammal destructor...\ n"; }
12:
13:     //Другие методы
14:   void Speak()const { cout << "Mammal sound!\ n"; }
15:   void Sleep()const { cout << "shhh. I'm sleeping.\ n"; }
16:
17:
18:  protected:
19:     int itsAge;
20:     int itsWeight;
21:  } ;
22:
23:  class Dog : public Mammal
24:  {
25:  public:
26:
27:     // Конструкторы
28:   Dog(){ cout << "Dog constructor...\ n"; }
29:   ~Dog(){ cout << "Dog destructor...\ n"; }
30:
31:     // Другие методы
32:   void WagTail() const { cout << "Tail wagging...\ n"; }
33:   void BegForFood() const { cout << "Begging for food...\ n"; }
34:   void Speak() const { cout << "Woof!\ n"; }
35:
36:  private:
37:     BREED itsBreed;
38:  } ;
39:
40:  int main()
41:  {
42:     Mammal bigAnimal;
43:     Dog fido;
44:     bigAnimal.Speak();
45:     fido.Speak();
46:  return 0;
47:  }
```

```
Mammal constructor...
Mammal constructor...
Dog constructor...
Mammal sound!
Woof!
Dog destructor...
Mammal destructor...
Mammal destructor...
```

В строке 34 в классе `Dog` происходит замещение метода базового класса `Speak()`, в результате чего в случае вызова этой функции объектом класса `Dog` на экран выводится `Woof!`. В строке 42 создается объект `bigAnimal` класса `Mammal`, в результате чего вызывается конструктор класса `Mammal` и на экране появляется первая строка. В строке 43 создается объект `Fido` класса `Dog`, что сопровождается последовательным вызовом сначала конструктора класса `Mammal`, а затем конструктора класса `Dog`. Соответственно на экран выводится еще две строки.

В строке 44 объект класса `Mammal` вызывает метод `Speak()`, а в строке 45 уже объект класса `Dog` обращается к этому методу. На экран при этом выводится разная информация, так как метод `Speak()` в классе `Dog` замещен. Наконец выполнение программы выходит за область видимости объектов и для их удаления вызываются соответствующие пары деструкторов.

Перегрузка или замещение

Эти схожие подходы приводят почти к одинаковым результатам. При перегрузке метода создается несколько вариантов этого метода с одним и тем же именем, но с разными сигнатурами. При замещении в производном классе используется метод с тем же именем и сигнатурой, что и в базовом классе, но с изменениями в теле функции.

Сокрытие метода базового класса

В предыдущем примере при обращении к методу `Speak()` из объекта класса `Dog` программа выполнялась не так, как было указано при объявлении метода `Speak()` в базовом классе. Казалось бы, это то, что нам нужно. Если в классе `Mammal` есть некоторый метод `Move()`, который замещается в классе `Dog`, то можно сказать, что метод `Move()` класса `Dog` скрывает метод с тем же именем в базовом классе. Однако в некоторых случаях результат может оказаться неожиданным.

Усложним ситуацию. Предположим, что в классе `Mammal` метод `Move()` трижды перегружен. В одном варианте метод не требует параметров, в другом используется один целочисленный параметр (дистанция), а в третьем — два целочисленных параметра (скорость и дистанция). В классе `Dog` замещен метод `Move()` без параметров. Тем не менее попытка обратиться из объекта класса `Dog` к двум другим вариантам перегруженного метода класса `Mammal` окажется неудачной. Суть проблемы раскрывается в листинге 11.6.

Листинг 11.6. Сокрытие методов

```
1:  //Листинг 11.6. Сокрытие методов
2:
3:  #include <iostream.h>
4:
5:  class Mammal
6:  {
7:  public:
8:   void Move() const { cout << "Mammal move one step\ n"; }
9:   void Move(int distance) const
10:   {
11:    cout << "Mammal move ";
12:    cout << distance <<" steps.\ n";
13:     }
14:  protected:
15:     int itsAge;
16:     int itsWeight;
17:  } ;
18:
19:  class Dog : public Mammal
20:  {
21:  public:
22: // Возможно, последует сообщение, что функция скрыта!
23:   void Move() const { cout << "Dog move 5 steps.\ n"; }
24:  } ;
25:
26:  int main()
27:  {
28:     Mammal bigAnimal;
29:     Dog fido;
30:     bigAnimal.Move();
31:     bigAnimal.Move(2);
32:     fido.Move();
33:     // fido.Move(10);
34:  return 0;
35:  }
```

РЕЗУЛЬТАТ

```
Mammal move one step
Mammal move 2 steps.
Dog move 5 steps.
```

АНАЛИЗ В данном примере из программы были удалены все другие методы и данные, рассмотренные нами ранее. В строках 8 и 9 в объявлении класса `Mammal` перегружаются методы `Move()`. В строке 23 происходит замещение метода `Move()` без параметров в классе `Dog`. Данный метод вызывается для объектов разных классов в строках 30 и 32, и информация, выводимая на экран, подтверждает, что замещение метода прошло правильно.

Однако строка 33 заблокирована, так как она вызовет ошибку компиляции. Хотя логично было предположить, что в классе Dog свободно можно использовать метод Move(int), поскольку замещен был только метод Move(), но в действительности в данной ситуации, чтобы использовать Move(int), его также нужно заместить в классе Dog. В случае замещения одного из перегруженных методов *скрытыми* оказываются все варианты этого метода в базовом классе. Если вы хотите использовать в производном классе другие варианты перегруженного метода, то их также нужно заместить в этом классе.

Часто случается ошибка, когда после попытки заместить метод в производном классе данный метод оказывается недоступным для класса из-за того, что программист забыл установить ключевое слово const, используемое при объявлении метода в базовом классе. Вспомните, что слово const является частью сигнатуры, а несоответствие сигнатур ведет к скрытию базового метода, а не к его замещению.

Замещение и сокрытие

В следующем разделе главы будут рассматриваться виртуальные методы. Замещение виртуальных методов ведет к полиморфизму, а сокрытие методов разрушает полиморфизм. Скоро вы узнаете об этом больше.

Вызов базового метода

Даже если вы заместили базовый метод, то все равно можете обратиться к нему, указав базовый класс, где хранится исходное объявление метода. Для этого в обращении к методу нужно явно указать имя базового класса, за которым следуют два символа двоеточия и имя метода. Например: Mammal::Move().

Если в листинге 11.6 переписать строку 32 так, как показано ниже, то ошибка во время компиляции больше возникать не будет:

```
32:      fido.Mammal::Move();
```

Такая запись, реализованная в листинге 11.7, называется явным обращением к методу базового класса.

Листинг 11.7. Явное обращение к методу базового класса

```
1:  //Листинг 11.7. Явное обращение к методу базового класса
2:
3:  #include <iostream.h>
4:
5:  class Mammal
6:  {
7:  public:
8:   void Move() const { cout << "Mammal move one step\ n"; }
9:   void Move(int distance) const
10:   {
11:     cout << "Mammal move " << distance;
12:     cout << " steps.\ n";
13:   }
14:
15: protected:
```

```
16:    int itsAge;
17:    int itsWeight;
18:  } ;
19:
20:  class Dog : public Mammal
21:  {
22:  public:
23:    void Move()const;
24:
25:  } ;
26:
27:  void Dog::Move() const
28:  {
29:    cout << "In dog move...\ n";
30:      Mammal::Move(3);
31:   }
32:
33:   int main()
34:   {
35:      Mammal bigAnimal;
36:      Dog fido;
37:      bigAnimal.Move(2);
38:      fido.Mammal::Move(6);
39:   return 0;
40:   }
```

РЕЗУЛЬТАТ

```
Mammal move 2 steps.
Mammal move 6 steps.
```

АНАЛИЗ

В строке 35 создается объект `bigAnimal` класса `Mammal`, а в строке 36 — объект `fido` класса `Dog`. В строке 37 вызывается метод `Move(int)` из базового класса для объекта класса `Dog`.

В предыдущей версии программы мы столкнулись с проблемой из-за того, что в классе `Dog` доступен только один замещенный метод `Move()`, в котором не задаются параметры. Проблема была разрешена явным обращением к методу `Move(int)` базового класса в строке 38.

Рекомендуется

Повышайте функциональные возможности класса путем создания новых производных классов.

Изменяйте выполнение отдельных функций в производных классах с помощью замещения методов.

Не рекомендуется

Не допускайте сокрытие функций базового класса из-за несоответствия сигнатур.

Виртуальные методы

В этой главе неоднократно подчеркивалось, что объекты класса Dog одновременно являются объектами класса Mammal. До сих пор под этим подразумевалось, что объекты класса Dog наследуют все атрибуты (данные) и возможности (методы) базового класса. Но в языке C++ принципы иерархического построения классов несут в себе еще более глубинный смысл.

Полиморфизм в C++ развит настолько, что допускается присвоение указателям на базовый класс адресов объектов производных классов, как в следующем примере:

```
Mammal* pMammal = new Dog;
```

Данное выражение создает в области динамической памяти новый объект класса Dog и возвращает указатель на этот объект, который является указателем класса Mammal. Это вполне логично, так как собака — представитель млекопитающих.

В этом суть полиморфизма. Например, можно объявить множество окон разных типов, включая диалоговые, прокручиваемые окна и поля списков, после чего создавать их в программе с помощью единственного виртуального метода draw(). Создав указатель на базовое окно и присваивая этому указателю адреса объектов производных классов, можно обращаться к методу draw() независимо от того, с каким из объектов в данный момент связан указатель. Причем всегда будет вызываться вариант метода, специфичный для класса выбранного объекта.

Затем этот указатель можно использовать для вызова любого метода класса Mammal. Причем если метод был замещен, скажем, в классе Dog, то при обращении к методу через указатель будет вызываться именно вариант, указанный в данном производном классе. В этом суть использования виртуальных функций. Листинг 11.8 показывает, как работает виртуальная функция и что происходит с невиртуальной функцией.

Листинг 11.8. Использование виртуальных методов

```
1:  //Листинг 11.8. Использование виртуальных методов
2:
3:  #include <iostream.h>
4:
5:  class Mammal
6:  {
7:  public:
8:   Mammal():itsAge(1) { cout << "Mammal constructor...\ n"; }
9:   virtual ~Mammal() { cout << "Mammal destructor...\ n"; }
10:   void Move() const { cout << "Mammal move one step\ n"; }
11:   virtual void Speak() const { cout << "Mammal speak!\ n"; }
12: protected:
13:   int itsAge;
14:
15: } ;
16:
```

```
17: class Dog : public Mammal
18: {
19: public:
20:   Dog() { cout << "Dog Constructor...\ n"; }
21:   virtual ~Dog() { cout << "Dog destructor...\ n"; }
22:   void WagTail() { cout << "Wagging Tail...\ n"; }
23:   void Speak()const { cout << "Woof!\ n"; }
24:   void Move()const { cout << "Dog moves 5 steps...\ n"; }
25:  } ;
26:
27:  int main()
28:  {
29:
30:     Mammal *pDog = new Dog;
31:     pDog->Move();
32:     pDog->Speak();
33:
34:  return 0;
35:  }
```

РЕЗУЛЬТАТ

```
Mammal constructor...
Dog Constructor...
Mammal move one step
Woof!
```

АНАЛИЗ В строке 11 объявляется виртуальный метод `Speak()` класса `Mammal`. Предполагается, что данный класс должен быть базовым для других классов. Вероятно также, что данная функция может быть замещена в производных классах.

В строке 30 создается указатель класса `Mammal` (`pDog`), но ему присваивается адрес нового объекта производного класса `Dog`. Поскольку собака является млекопитающим, это вполне логично. Данный указатель затем используется для вызова функции `Move()`. Поскольку `pDog` известен компилятору как указатель класса `Mammal`, результат получается таким же, как при обычном вызове метода `Move()` из объекта класса `Mammal`.

В строке 32 через указатель `pDog` делается обращение к методу `Speak()`. В данном случае метод `Speak()` объявлен как виртуальный, поэтому вызывается вариант функции `Speak()`, замещенный в классе `Dog`.

Это кажется каким-то волшебством. Хотя компилятор знает, что указатель `pDog` принадлежит классу `Mammal`, тем не менее происходит вызов версии функции, объявленной в другом производном классе. Если создать массив указателей базового класса, каждый из которых указывал бы на объект своего производного класса, то, обращаясь попеременно к указателям данного массива, можно управлять выполнением всех вариантов замещенного метода. Эта идея реализована в листинге 11.9.

Листинг 11.9. Произвольное обращение к набору виртуальных функций

```
1:    //Листинг 11.9. Произвольное обращение к набору виртуальных функций
2:
3:    #include <iostream.h>
4:
```

```
5:  class Mammal
6:  {
7:  public:
8:    Mammal():itsAge(1) {   }
9:    virtual ~Mammal() {   }
10:   virtual void Speak() const { cout << "Mammal speak!\ n"; }
11: protected:
12:   int itsAge;
13: } ;
14:
15: class Dog : public Mammal
16: {
17: public:
18:   void Speak()const { cout << "Woof!\ n"; }
19: } ;
20:
21:
22: class Cat : public Mammal
23: {
24: public:
25:   void Speak()const { cout << "Meow!\ n"; }
26: } ;
27:
28:
29: class Horse : public Mammal
30: {
31: public:
32:   void Speak()const { cout << "Whinny!\ n"; }
33: } ;
34:
35: class Pig : public Mammal
36: {
37: public:
38:   void Speak()const { cout << "Oink!\ n"; }
39: } ;
40:
41: int main()
42: {
43:   Mammal* theArray[5];
44:   Mammal* ptr;
45:   int choice, i;
46:   for ( i = 0; i<5; i++)
47:   {
48:    cout << "(1)dog (2)cat (3)horse (4)pig: ";
49:    cin >> choice;
50:    switch (choice)
51:    {
52:      case 1: ptr = new Dog;
53:      break;
54:      case 2: ptr = new Cat;
```

```
55:       break;
56:       case 3: ptr = new Horse;
57:       break;
58:       case 4: ptr = new Pig;
59:       break;
60:       default: ptr = new Mammal;
61:       break;
62:     }
63:     theArray[i] = ptr;
64:    }
65:    for (i=0;i<5;i++)
66:     theArray[i]->Speak();
67:   return 0;
68:  }
```

Результат

```
(1)dog (2)cat (3)horse (4)pig: 1
(1)dog (2)cat (3)horse (4)pig: 2
(1)dog (2)cat (3)horse (4)pig: 3
(1)dog (2)cat (3)horse (4)pig: 4
(1)dog (2)cat (3)horse (4)pig: 5
Woof!
Meow!
Whinny!
Oink!
Mammal speak!
```

Анализ

Чтобы идея использования виртуальных функций была понятнее, в данной программе этот метод раскрыт наиболее явно и четко. Сначала определяется четыре класса — `Dog`, `Cat`, `Horse` и `Pig`, которые являются производными от базового класса `Mammal`.

В строке 10 объявляется виртуальная функция `Speak()` класса `Mammal`. В строках 18, 25, 32 и 38 указанная функция замещается во всех соответствующих производных классах.

Пользователю предоставляется возможность выбрать объект любого производного класса, и в строках 46—64 создается и добавляется в массив указатель класса `Mammal` на вновь созданный объект.

Вопросы и ответы

Если функция-член была объявлена как виртуальная в базовом классе, следует ли повторно указывать виртуальность при объявлении этого метода в производном классе?

Нет. Если метод уже был объявлен как виртуальный, то он будет оставаться таким, несмотря на замещение его в производном классе. В то же время для повышения читабельности программы имеет смысл (но не требуется) и в производных классах продолжать указывать на виртуальность данного метода с помощью ключевого слова `virtual`.

ПРИМЕЧАНИЕ

Во время компиляции неизвестно, объект какого класса захочет создать пользователь и какой именно вариант метода `Speak()` будет использоваться. Указатель `ptr` связывается со своим объектом только во время выполнения программы. Такое связывание указателя с объектом называется *динамическим*, в отличие от *статического* связывания, происходящего во время компиляции программы.

Как работают виртуальные функции

При создании объекта в производном классе, например в классе `Dog`, сначала вызывается конструктор базового, а затем — производного класса. Схематично объект класса `Dog` показан на рис. 11.2. Обратите внимание, что объект производного класса состоит как бы из двух частей, одна из которых создается конструктором базового класса, а другая — конструктором производного класса.

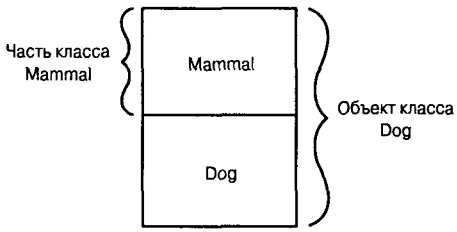


Рис. 11.2. Созданный объект класса `Dog`

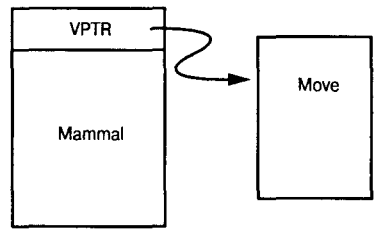


Рис. 11.3. Таблица виртуальных функций класса `Mammal`

Если в каком-то из объектов создается обычная невиртуальная функция, то всю полноту ответственности за эту функцию берет на себя объект. Большинство компиляторов создают *таблицы виртуальных функций*, называемые также *v-таблицами*. Такие таблицы создаются для каждого типа данных, и каждый объект любого класса содержит указатель на таблицу виртуальных функций (`vptr`, или v-указатель).

Хотя детали реализации выполнения виртуальных функций меняются в разных компиляторах, сами виртуальные функции будут работать совершенно одинаково, независимо от компилятора.

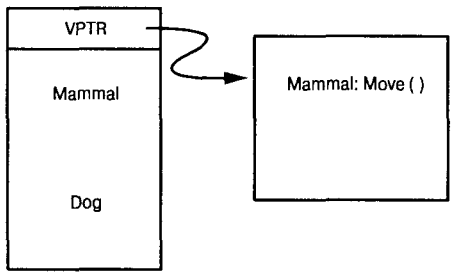


Рис. 11.4. Таблица виртуальных функций класса `Dog`

Итак, в каждом объекте есть указатель `vptr`, который ссылается на таблицу виртуальных функций, содержащую, в свою очередь, указатели на все виртуальные функции. (Более подробно указатели на функции рассматриваются на занятии 14.) Указа-

тель vptr для объекта класса Dog инициализируется при создании части объекта, принадлежащей базовому классу Mammal, как показано на рис. 11.3.

После вызова конструктора класса Dog указатель vptr настраивается таким образом, чтобы указывать на замещенный вариант виртуальной функции (если такой есть), существующий для класса Dog (рис. 11.4).

В результате при использовании указателя на класс Mammal указатель vptr попрежнему ссылается на тот вариант виртуальной функции, который соответствует реальному типу объекта. Поэтому при обращении к методу Speak() в предыдущем примере выполнялась та функция, которая была задана в соответствующем производном классе.

Нельзя брать там, находясь здесь

Если для объекта класса Dog объявлен метод WagTail(), который не принадлежит классу Mammal, то невозможно получить доступ к этому методу, используя указатель класса Mammal (если только этот указатель не будет явно преобразован в указатель класса Dog). Поскольку функция WagTail() не является виртуальной и не принадлежит классу Mammal, то доступ к ней можно получить только из объекта класса Dog или с помощью указателя этого класса.

Поскольку любые преобразования чреваты ошибками, создатели C++ допустили только явные преобразования типов. Всегда можно преобразовать любой указатель класса Mammal в указатель класса Dog, но есть более надежный и безопасный способ вызова метода WagTail(). Чтобы разобраться в тонкостях упомянутого метода, необходимо освоить множественное наследование, о котором речь пойдет на следующем занятии, или научиться работе с шаблонами, что будет темой занятия 20.

Дробление объекта

Следует обратить внимание, что вся магия виртуальных функций проявляется только при обращении к ним с помощью указателей и ссылок. Если передать объект как значение, то виртуальную функцию вызвать не удастся. Эта проблема показана в листинге 11.10.

Листинг 11.10. Дробление объекта при передаче его как значения

```
1:    //Листинг 11.10. Дробление объекта при передачи его как значения
2:
3:    #include <iostream.h>
4:
5:    class Mammal
6:  {
7:  public:
8:    Mammal():itsAge(1) {   }
9:    virtual ~Mammal() {   }
10:   virtual void Speak() const { cout << "Mammal speak!\ n"; }
11: protected:
12:   int itsAge;
13: } ;
14:
15: class Dog : public Mammal
16: {
```

```
17:  public:
18:    void Speak()const { cout << "Woof!\ n"; }
19:  } ;
20:
21:  class Cat : public Mammal
22:  {
23:  public:
24:    void Speak()const { cout << "Meow!\ n"; }
25:  } ;
26:
27   void ValueFunction (Mammal);
28:  void PtrFunction  (Mammal*);
29:  void RefFunction (Mammal&);
30:  int main()
31:  {
32:    Mammal* ptr=0;
33:    int choice;
34:    while (1)
35:    {
36:     bool fQuit = false;
37:     cout << "(1)dog (2)cat (0)Quit: ";
38:     cin >> choice;
39:     switch (choice)
40:     {
41:       case 0: fQuit = true;
42:       break;
43:       case 1: ptr = new Dog;
44:       break;
45:       case 2: ptr = new Cat;
46:       break;
47:       default: ptr = new Mammal;
48:       break;
49:     }
50:      if (fQuit)
51:       break;
52:      PtrFunction(ptr);
53:      RefFunction(*ptr);
54:      ValueFunction(*ptr);
55:    }
56:    return 0;
57:  }
58:
59:  void ValueFunction (Mammal MammalValue)
60:  {
61:    MammalValue.Speak();
62:  }
63:
64:  void PtrFunction (Mammal * pMammal)
65:  {
66:    pMammal->Speak();
```

```
67:  }
68:
69:  void RefFunction (Mammal & rMammal)
70:  {
71:    rMammal.Speak();
72:  }
```

Результат

```
(1)dog (2)cat (0)Quit: 1
Woof
Woof
Mammal Speak!
(1)dog (2)cat (0)Quit: 2
Meow!
Meow!
Mammal Speak!
(1)dog (2)cat (0)Quit: 0
```

Анализ В строках 5–25 определяются классы `Mammal`, `Dog` и `Cat`. Затем объявляются три функции — `PtrFunction()`, `RefFunction()` и `ValueFunction()`. Они принимают соответственно указатель класса `Mammal`, ссылку класса `Mammal` и объект класса `Mammal`. После чего выполняют одну и ту же операцию — вызывают метод `Speak()`.

Пользователю предлагается выбрать объект класса `Dog` или класса `Cat`, после чего в строках 43–46 создается указатель соответствующего типа.

Судя по информации, выведенной программой на экран, пользователь первый раз выбрал объект класса `Dog`, который был создан в свободной области памяти 43-й строкой программы. Затем объект класса `Dog` передается в три функции с помощью указателя, с помощью ссылки и как значение.

В том случае, когда в функцию передавался адрес объекта с помощью указателя или ссылки, успешно выполнялась функция-член `Dog->Speak()`. На экране компьютера дважды появилось сообщение, соответствующее выбранному пользователем объекту.

Разыменованный указатель передает объект как значение. В этом случае функция распознает принадлежность переданного объекта классу `Mammal`, компилятор разбивает объект класса `Dog` пополам и использует только ту часть, которая была создана конструктором класса `Mammal`. В таком случае вызывается версия метода `Speak()`, которая была **объявлена** для класса `Mammal`, что и отобразилось в информации, выведенной программой на экран.

Те же действия и с тем же результатом были выполнены затем и для объекта класса `Cat`.

Виртуальные деструкторы

В том случае, когда ожидается указатель на объект базового класса, вполне допустима и часто используется на практике передача указателя на объект производного класса. Что произойдет при удалении указателя, ссылающегося на объект производного класса? Если деструктор будет объявлен как виртуальный, то все пройдет отлично — будет вызван деструктор соответствующего производного класса. Затем деструктор производного класса автоматически вызовет деструктор базового класса, и указанный объект будет удален целиком.

Отсюда следует правило: если в классе объявлены виртуальные функции, то и деструктор должен быть виртуальным.

Виртуальный конструктор-копировщик

Конструкторы не могут быть виртуальными, из чего можно сделать вывод, что не может быть также виртуального конструктора-копировщика. Но иногда требуется, чтобы программа могла передать указатель на объект базового класса и правильно скопировать его в объект производного класса. Чтобы добиться этого, необходимо в базовом классе создать виртуальный метод `Clone()`. Метод `Clone()` должен создавать и возвращать копию объекта текущего класса.

Поскольку в производных классах метод `Clone()` замещается, при вызове его создаются копии объектов, соответствующие выбранному классу. Программа, использующая этот метод, показана в листинге 11.11.

Листинг 11.11. Виртуальный конструктор-копировщик

```
1:  //Листинг 11.11. Виртуальный конструктор-копировщик
2:
3:  #include <iostream.h>
4:
5:  class Mammal
6:  {
7:  public:
8:    Mammal():itsAge(1) { cout << "Mammal constructor...\ n"; }
9:    virtual ~Mammal() { cout << "Mammal destructor...\ n"; }
10:   Mammal (const Mammal & rhs);
11:   virtual void Speak() const { cout << "Mammal speak!\ n"; }
12:   virtual Mammal* Clone() { return new Mammal(*this); }
13:   int GetAge()const { return itsAge; }
14: protected:
15:   int itsAge;
16: } ;
17:
18: Mammal::Mammal (const Mammal & rhs):itsAge(rhs.GetAge())
19: {
20:   cout << "Mammal Copy Constructor...\ n";
21: }
22:
23: class Dog : public Mammal
24: {
25: public:
26:   Dog() { cout << "Dog constructor...\ n"; }
27:   virtual ~Dog() { cout << "Dog destructor...\ n"; }
28:   Dog (const Dog & rhs);
29:   void Speak()const { cout << "Woof!\ n"; }
30:   virtual Mammal* Clone() { return new Dog(*this); }
31: } ;
32:
33: Dog::Dog(const Dog & rhs):
34: Mammal(rhs)
35: {
36:   cout << "Dog copy constructor...\ n";
```

```
37: }
38:
39: class Cat : public Mammal
40: {
41: public:
42:   Cat() { cout << "Cat constructor...\ n"; }
43:   ~Cat() { cout << "Cat destructor...\ n"; }
44:   Cat (const Cat &);
45:   void Speak()const { cout << "Meow!\ n"; }
46:   virtual Mammal* Clone() { return new Cat(*this); }
47: } ;
48:
49: Cat::Cat(const Cat & rhs):
50: Mammal(rhs)
51: {
52:   cout << "Cat copy constructor...\ n";
53: }
54:
55: enum ANIMALS { MAMMAL, DOG, CAT} ;
56: const int NumAnimalTypes = 3;
57: int main()
58: {
59:   Mammal *theArray[NumAnimalTypes];
60:   Mammal* ptr;
61:   int choice, i;
62:   for ( i = 0; i<NumAnimalTypes; i++)
63:   {
64:    cout << "(1)dog (2)cat (3)Mammal: ";
65:    cin >> choice;
66:    switch (choice)
67:    {
68:      case DOG: ptr = new Dog;
69:      break;
70:      case CAT: ptr = new Cat;
71:      break;
72:      default: ptr = new Mammal;
73:      break;
74:    }
75:    theArray[i] = ptr;
76:   }
77:   Mammal *OtherArray[NumAnimalTypes];
78:   for (i=0;i<NumAnimalTypes;i++)
79:   {
80:    theArray[i]->Speak();
81:    OtherArray[i] = theArray[i]->Clone();
82:   }
83:   for (i=0;i<NumAnimalTypes;i++)
84:    OtherArray[i]->Speak();
85:  return 0;
86: }
```

```
1: (1)dog (2)cat (3)Mammal: 1
2: Mammal constructor...
3: Dog constructor...
4: (1)dog (2)cat (3)Mammal: 2
5: Mammal constructor...
6: Cat constructor...
7: (1)dog (2)cat (3)Mammal: 3
8: Mammal constructor...
9: Woof!
10: Mammal Copy Constructor...
11: Dog copy constructor...
12: Meow!
13: Mammal Copy Constructor...
14: Cat copy constructor...
15: Mammal speak!
16: Mammal Copy Constructor...
17: Woof!
18: Meow!
19: Mammal speak!
```

Листинг 11.11 похож на два предыдущих листинга, однако в данной программе в классе `Mammal` добавлен один новый виртуальный метод — `Clone()`. Этот метод возвращает указатель на новый объект класса `Mammal`, используя конструктор-копировщик, параметр которого представлен указателем `*this`.

Метод `Clone()` замещается в обоих производных классах — `Dog` и `Cat` — соответствующими версиями, после чего копии данных передаются на конструкторы-копировщики производных классов. Поскольку `Clone()` является виртуальной функцией, то в результате будут созданы виртуальные конструкторы-копировщики, как показано в строке 81.

Пользователю предлагается выбрать объект класса `Dog`, `Cat` или `Mammal`. Объект выбранного типа создается в строках 62–74. В строке 75 указатель на новый объект добавляется в массив данных.

Затем осуществляется цикл, в котором для каждого объекта массива вызываются методы `Speak()` и `Clone()` (см. строки 80 и 81). В результате выполнения функции возвращается указатель на копию объекта, которая сохраняется в строке 81 во втором массиве.

В строке 1 вывода на экран показан выбор пользователем опции `1` — создание объекта класса `Dog`. В создание этого объекта вовлекаются конструкторы базового и производного классов. Эта операция повторяется для объектов классов `Cat` и `Mammal` в строках вывода 4–8.

В строке 9 вывода показано выполнение метода `Speak()` для объекта класса `Dog`. Поскольку функция `Speak()` также объявлена как виртуальная, то при обращении к ней вызывается та ее версия, которая соответствует типу объекта. Затем следует обращение еще к одной виртуальной функции `Clone()`, виртуальность которой проявляется в том, что при вызове из объекта класса `Dog` запускаются конструктор класса `Mammal` и конструктор-копировщик класса `Dog`.

То же самое повторяется для объекта класса `Cat` (строки вывода с 12–14) и объекта класса `Mammal` (строки вывода 15 и 16). В результате создается массив объектов, для каждого из которых вызывается своя версия функции `Speak()`.

Цена виртуальности методов

Поскольку объекты с виртуальными методами должны поддерживать v-таблицу, то использование виртуальных функций всегда ведет к некоторому повышению затрат памяти и снижению быстродействия программы. Если вы работаете с небольшим классом, который не собираетесь делать базовым для других классов, то в этом случае нет никакого смысла использовать виртуальные методы.

Объявляя виртуальный метод в программе, заплатить придется не только за v-таблицу (хотя добавление последующих записей потребует не так уж много места), но и за создание виртуального деструктора. Поэтому следует подумать, имеет ли смысл преобразовывать методы программы в виртуальные, а если да, то какие именно.

Рекомендуется	Не рекомендуется
Используйте виртуальные методы только в том случае, если программа содержит базовый и производные классы.	**Не пытайтесь** создать виртуальный конструктор.
Используйте виртуальный деструктор, если в программе были созданы виртуальные методы.	

Резюме

Сегодня вы узнали, как наследовать новые классы от базового класса. В этой главе рассматривалось наследование с ключевым словом `public` и использование виртуальных функций. *Во время наследования в производные классы передаются все открытые* и защищенные данные и функции из базового класса.

Защищенные данные базового класса открыты для всех производных классов, но закрыты для всех других классов программы. Но даже производные классы не могут получить доступ к закрытым данным и функциям базового класса.

Конструкторы могут инициализироваться до выполнения тела конструктора. При этом вызывается конструктор базового класса, и туда могут быть переданы данные в виде параметров.

Функции, объявленные в базовом классе, могут быть замещены в производных классах. Если при этом функция объявлена как виртуальная, а обращение к функции от объекта осуществляется с помощью указателя на объект или ссылки, то вызываться будет тот замещенный вариант функции, который соответствует типу текущего объекта.

Методы базового класса можно вызывать явным обращением, когда в строке вызова сначала указывается имя базового класса с двумя символами двоеточия после него. Например, если класс `Dog` произведен от класса `Mammal`, то к методу базового класса напрямую можно обратиться следующим выражением: `Mammal::walk()`.

Если в классе используются виртуальные методы, то следует объявить также и виртуальный деструктор. Он необходим для того, чтобы быть уверенным в удалении части объекта, относящейся к производному классу, если удаление объекта осуществлялось с помощью указателя базового класса. Нельзя создать виртуальный конструктор. В то же время можно создать виртуальный конструктор-копировщик и эффективно его использовать с помощью виртуальной функции, вызывающей конструктор-копировщик.

Вопросы и ответы

Наследуются ли данные и функции-члены базового класса в последующие поколения производных классов? Скажем, если класс `Dog` произведен от класса `Mammal`, а класс `Mammal` произведен от класса `Animals`, унаследует ли класс `Dog` данные и функции класса `Animals`?

Да. Если последовательно производить ряд классов, последний класс в этом ряду унаследует всю сумму данных и методов предыдущих базовых классов.

Если в предыдущем примере в классе `Mammal` будет замещена функция, описанная в классе `Animals`, то какой вариант функции получит класс `Dog`?

Если класс `Dog` наследуется от класса `Mammal`, то он получит функцию в том виде, в каком она существует в классе `Mammal`, т.е. замещенную.

Можно ли в производном классе описать как `private` функцию, которая перед этим была описана в базовом классе как `public`?

Можно. Функция может быть не только защищена в производном классе, но и закрыта. Она останется закрытой для всех последующих классов, произведенных от этого.

В каких случаях не следует делать функции класса виртуальными?

Описание первой виртуальной функции вызовет создание v-таблицы, что потребует времени и дополнительной памяти. Последующее добавление виртуальных функций будет тривиальным. Многие программисты увлекаются созданием виртуальных функций и полагают, что если в программе есть уже одна виртуальная функция, то и все другие должны быть виртуальными. В действительности это не так. Создание виртуальных функций всегда должно отвечать решению конкретных задач.

Предположим, что некоторая функция без параметров была описана в базовом классе как виртуальная, а затем перегружена таким образом, чтобы принимать один и два целочисленных параметра. Затем в производном классе был замещен вариант функции с одним целочисленным параметром. Что произойдет, если с помощью указателя, связанного с объектом производного класса, вызвать вариант функции с двумя параметрами?

Замещение в производном классе варианта функции с одним параметром скроет от объектов этого класса все остальные варианты функции. Поэтому в случае обращения, описанного в вопросе, компилятор покажет сообщение об ошибке.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Тест

1. Что такое v-таблица?
2. Что представляет собой виртуальный деструктор?
3. Можно ли объявить виртуальный конструктор?

4. Как создать виртуальный конструктор-копировщик?
5. Как вызвать функцию базового класса из объекта производного класса, если в производном классе эта функция была замещена?
6. Как вызвать функцию базового класса из объекта производного класса, если в производном классе эта функция не была замещена?
7. Если в базовом классе функция объявлена как виртуальная, а в производном классе виртуальность функции указана не была, сохранится ли функция как виртуальная в следующем произведенном классе?
8. С какой целью используется ключевое слово `protected`?

Упражнения

1. Объявите виртуальную функцию, которая принимает одно целочисленное значение и возвращает `void`.
2. Запишите объявление класса `Square`, произведенного от класса `Rectangle`, который, в свою очередь, произведен от класса `Shape`.
3. Предположим, что в предыдущем примере объект класса `Shape` не использует параметры, объект класса `Rectangle` принимает два параметра (`length` и `width`), а объект класса `Square` — один параметр (`length`); запишите конструктор для класса `Square`.
4. Запишите виртуальный конструктор-копировщик для класса `Square`, взятого из упражнения 3.
5. **Жучки**: что неправильно в следующем программном коде?

```
void SomeFunction(Shape);
Shape * pRect = new Rectangle;
SomeFunction(*pRect);
```

6. **Жучки**: что неправильно в следующем программном коде?

```
class Shape()
{
public:
   Shape();
   virtual -Shape();
   virtual Shape(const Shape&);
};
```

День 12-й

Массивы и связанные листы

В программах, представленных в предыдущей главе, объявлялись одиночные объекты типов `int`, `char` и др. Но часто возникает необходимость создать коллекцию объектов, например 20 значений типа `int` или кучу объектов типа `CAT`. Сегодня вы узнаете:

- Что представляет собой массив и как его объявить
- Что такое строки и как их создавать с помощью массивов символов
- Какие существуют отношения между массивами и указателями
- Каковы особенности математических операций с указателями, связанными с массивами

Что такое массивы

Массивы представляют собой коллекции данных одного типа, сохраненные в памяти компьютера. Каждая единица данных называется *элементом массива*.

Чтобы объявить массив, нужно указать его тип, имя и размер. Размер задается числом, взятым в квадратные скобки, и указывает, сколько элементов можно сохранить в данном массиве, например:

```
long LongArray[25];
```

В этом примере объявляется массив под именем `LongArray`, который может содержать 25 элементов типа `long int`. Обнаружив подобную запись, компилятор резервирует в памяти компьютера место, чтобы сохранить 25 элементов указанного типа. Поскольку для сохранения одного значения типа `long int` требуется четыре байта памяти, то для заданного массива компилятор выделит цельную область памяти размером 100 байт (рис. 12.1).

Элементы массива

Адресация элементов массива определяется по сдвигу относительно адреса первого элемента, сохраненного в имени массива. Первый элемент массива имеет нулевой сдвиг. Таким образом, к первому элементу массива можно обратиться следующим об-

разом: arrayName[0]. Если использовать пример массива, приведенный в предыдущем разделе, то обращение к первому элементу массива будет выглядеть так: LongArray[0], а ко второму — LongArray[1] и т.д. В общем виде, если объявлен массив *Массив*[n], то к его элементам можно обращаться, указывая индекс от *Массив*[0]до *Массив*[n-1].

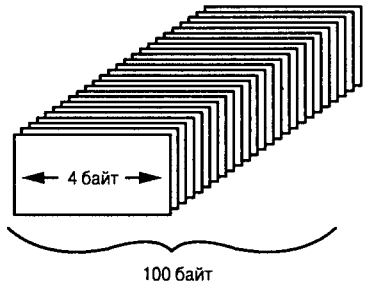


Рис. 12.1. Объявление массива

Так, в нашем примере массива LongArray[25] для обращения к элементам используются индексы от LongArray[0]до LongArray[24]. В листинге 12.1 показано объявление массива целых чисел из пяти элементов и заполнение его данными.

Листинг 12.1. Использование массива целых чисел

```
1:   //Листинг 12.1. Массивы
2:   #include <iostream.h>
3:
4:   int main()
5:   {
6:     int myArray[5];
7:     int i;
8:     for ( i=0; i<5; i++)   // 0-4
9:     {
10:      cout << "Value for myArray[" << i << "]: ";
11:    cin >> myArray[i];
12:    }
13:    for (i = 0; i<5; i++)
14:     cout << i << ": " << myArray[i] << "\ n";
15:   return 0;
16:  }
```

РЕЗУЛЬТАТ

```
Value for myArray[0]: 3
Value for myArray[1]: 6
Value for myArray[2]: 9
Value for myArray[3]: 12
Value for myArray[4]: 15
0: 3
1: 6
2: 9
3: 12
4: 15
```

АНАЛИЗ В строке 6 объявляется массив myArray, который может содержать пять целочисленных значений. В строке 8 начинается цикл от 0 до 4, в котором задаются допустимые индексы созданного массива. Пользователю предлагается ввести свое значение для текущего элемента массива, после чего это значение сохраняется в компьютере по адресу, отведенному компилятором для данного элемента массива.

Первое значение сохраняется по адресу, указанному в имени массива с нулевым сдвигом, — myArray[0], второе — в ячейке myArray[1]и т.д. Второй цикл программы выводит сохраненные значения на экран.

Следует запомнить, что отсчет элементов массива начинается с 0, а не с 1. Это источник частых ошибок новичков в программах на C++. Если используется массив, состоящий из 10 элементов, то для обращения к элементам массива используются индексы от ArrayName[0] до ArrayName[9]. Обращение ArrayName[10] будет ошибочным.

Ввод данных за пределы массива

При записи данных в массив компилятор вычисляет адрес соответствующего элемента, основываясь на размере элемента и указанном сдвиге относительно первого элемента. Предположим, что некоторое значение записывается в шестой элемент рассмотренного нами ранее массива LongArray, для чего используется индекс LongArray[5]. Компилятор умножит указанное значение сдвига 5 на размер элемента (в нашем примере 4 байт) и получит 20 байт. Затем компилятор вычислит адрес шестого элемента массива, добавив к адресу массива 20 байт сдвига и запишет введенное значение по этому адресу.

Если при записи данных будет указан индекс LongArray[50], то компилятор не сможет самостоятельно определить, что такого элемента массива просто не существует. Компилятор вычислит, что такой элемент должен находиться по адресу, сдвинутому на 200 байт относительно адреса первого элемента массива, и запишет в эту ячейку памяти введенное значение. В связи с тем, что выбранная область памяти может принадлежать любой другой переменной, результат такой операции для работы программы непредсказуем. Если вам повезет, то программа зависнет сразу же. Если вы неудачник, то программа продолжит работу и через некоторое время выдаст вам совершенно неожиданный результат. Такие ошибки очень сложно локализовать, поскольку строка, где проявляется ошибка, и строка, где ошибка была допущена в программе, могут далеко отстоять друг от друга.

Компилятор ведет себя, как слепой человек, отмеряющий расстояние от дома к дому шагами. Он стоит возле первого дома на улице с адресом ГлавнаяУлица[0] и спрашивает вас, куда ему идти. Если будет дано указание следовать до шестого дома, то наш человек-компилятор станет размышлять следующим образом: "Чтобы добраться до шестого дома, от этого дома нужно пройти еще пять домов. Чтобы пройти один дом, нужно сделать четыре больших шага. Следовательно, нужно сделать 20 больших шагов." Если вы поставите задачу идти до дома ГлавнаяУлица[100], а на этой улице есть только 25 домов, то компилятор послушно начнет отмерять шаги и даже не заметит, что улица закончилась и началась проезжая часть с несущимися машинами. Поэтому, посылая компилятор по адресу, помните, что вся ответственность за последствия лежит только на вас.

Возможный результат ошибочной записи за пределы массива показан в листинге 12.2.

ПРЕДУПРЕЖДЕНИЕ Ни в коем случае не запускайте эту программу у себя на компьютере. Она может привести к поломке системы.

Листинг 12.2. Запись за пределы массива

```
1:  //Листинг 12.2.
2:  // Пример того, что может произойти при записи
3:  // за пределы массива
4:
5:  #include <iostream.h>
6:  int main()
7:  {
8:    // часовые
9:    long sentinelOne[3];
10:     long TargetArray[25]; // массив для записи данных
11:     long sentinelTwo[3];
12:     int i;
13:     for (i=0; i<3; i++)
14:       sentinelOne[i] = sentinelTwo[i] = 0;
15:
16:     for (i=0; i<25; i++)
17:       TargetArray[i] = 0;
18:
19:   cout << "Test 1: \ n"; // test current values (should be 0)
20:   cout << "TargetArray[0]: " << TargetArray[0] << "\ n";
21:   cout << "TargetArray[24]: " << TargetArray[24] << "\ n\ n";
22:
23:   for (i = 0; i<3; i++)
24:   {
25:    cout << "sentinelOne[" << i << "]: ";
26:    cout << sentinelOne[i] << "\ n";
27:    cout << "sentinelTwo[" << i << "]: ";
28:     cout << sentinelTwo[i]<< "\ n";
29:   }
30:
31:   cout << "\ nAssigning...";
32:   for (i = 0; i<=25; i++)
33:    TargetArray[i] = 20;
34:
35:   cout << "\ nTest 2: \ n";
36:   cout << "TargetArray[0]: " << TargetArray[0] << "\ n";
37:   cout << "TargetArray[24]: " << TargetArray[24] << "\ n";
38:   cout << "TargetArray[25]: " << TargetArray[25] << "\ n\ n";
39:   for (i = 0; i<3; i++)
40:   {
41:    cout << "sentinelOne[" << i << "]: ";
42:    cout << sentinelOne[i]<< "\ n";
43:    cout << "sentinelTwo[" << i << "]: ";
```

```
44:     cout << sentinelTwo[i]<< "\ n";
45:    }
46:
47:   return 0;
48:  }
```

РЕЗУЛЬТАТ

```
Test 1:
TargetArray[0]: 0
TargetArray[24]: 0

SentinelOne[0]: 0
SentinelTwo[0]: 0
SentinelOne[1]: 0
SentinelTwo[1]: 0
SentinelOne[2]: 0
SentinelTwo[2]: 0

Assigning...
Test 2:
TargetArray[0]: 20
TargetArray[24]: 20
TargetArray[25]: 20

SentinelOne[0]: 20
SentinelTwo[0]: 0
SentinelOne[1]: 0
SentinelTwo[1]: 0
SentinelOne[2]: 0
SentinelTwo[2]: 0
```

АНАЛИЗ

В строках 9 и 11 объявляются два массива типа `long` по три элемента в каждом, которые выполняют роль часовых вокруг массива `TargetArray`. Изначально значения этих массивов устанавливаются в 0. Если будет записано значение в массив `TargetArray` по адресу, выходящему за пределы этого массива, то значения массивов-часовых изменятся. Одни компиляторы ведут отсчет по возрастающей от адреса массива, другие — по убывающей. Именно поэтому используется два вспомогательных массива, расположенных по обе стороны от целевого массива `TargetArray`.

В строках 19–29 проверяется равенство нулю значений элементов массивов-часовых (`Test 1`). В строке 33 элементу массива `TargetArray` присваивается значение 20, но при этом указан индекс 25, которому не соответствует ни один элемент массива `TargetArray`.

В строках 36–38 выводятся значения элементов массива `TargetArray` (`Test 2`). Обратите внимание, что обращение к элементу массива `TargetArray[25]` проходит вполне успешно и возвращается присвоенное ранее значение 20. Но когда на экран выводятся значения массивов-часовых `SentinelOne` и `SentinelTwo`, вдруг обнаруживается, что значение элемента массива `SentinelOne` изменилось. Дело в том, что обращение к массиву `TargetArray[25]` ссылается на ту же ячейку памяти, что и элемент массива `SentinelOne[0]`. Таким образом, записывая значения в несуществующий элемент массива `TargetArray`, программа изменяет значение элемента совсем другого массива.

Если далее в программе значения элементов массива SentinelOne будут использоваться в каких-то расчетах, то причину возникновения ошибки будет сложно определить. В этом состоит коварство ввода значений за пределы массива.

В нашем примере размеры массивов были заданы значениями 3 и 25 в объявлении массивов. Гораздо безопаснее использовать для этого константы, объявленные где-нибудь в одном месте программы, чтобы программист мог легко контролировать размеры всех массивов в программе.

Еще раз отметим, что, поскольку разные компиляторы по-разному ведут отсчет адресов памяти, результат выполнения показанной выше программы может отличаться на вашем компьютере.

Ошибки подсчета столбов для забора

Ошибки с записью данных за пределы массива случаются настолько часто, что для них используется особый термин — *ошибки подсчета столбов для забора*. Такое странное название было придумано по аналогии с одной житейской проблемой — подсчетом, сколько столбов нужно вкопать, чтобы установить 10-метровый забор, если расстояние между столбами должно быть 1 м. Многие не задумываясь отвечают — десять. Вот если бы задумались и посчитали, то нашли бы правильный ответ — одиннадцать. Если вы не поняли почему, посмотрите на рис. 12.2.

Ошибка на единицу при установке индекса в обращении к массиву может стоить программе жизни. Нужно время, чтобы начинающий программист привык, что при обращении к массиву, состоящему из 25 элементов, индекс не может превышать значение 24 и отсчет начинается с 0, а не с 1. (Некоторые программисты потом настолько к этому привыкают, что в лифте нажимают на кнопку 4, когда им нужно подняться на пятый этаж.)

Иногда элемент массива *ИмяМассива*[0] называют нулевым, а не первым. Но и в этом случае легко запутаться. Если элемент *ИмяМассива*[0] нулевой, то каким тогда будет элемент *ИмяМассива*[1]? Первым или вторым? И так далее... Каким будет элемент *ИмяМассива*[24] — двадцать четвертым или двадцать пятым? Правильно будет считать элемент *ИмяМассива*[0] первым, имеющим нулевой сдвиг.

Инициализация массива

Инициализацию массива базового типа (например, int или char) можно проводить одновременно с его объявлением. Для этого за выражением объявления массива нужно установить знак равенства (=) и в фигурных скобках список значений элементов массива, разделенных запятыми. Например:

```
int IntegerArray[5] = {10, 20, 30, 40, 50};
```

В этом примере объявляется массив целых чисел IntegerArray и элементу IntegerArray[0] присваивается значение 10, элементу IntegerArray[1] — 20 и т.д.

Если вы опустите установку размера массива, то компилятор автоматически вычислит размер массива по списку значений элементов. Поэтому справедливой является следующая запись:

```
int IntegerArray[] = {10, 20, 30, 40, 50};
```

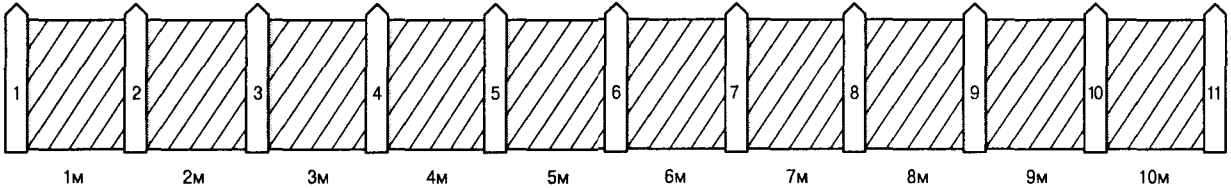


Рис. 12.2. Ошибка подсчета столбов для забора

В результате получим тот же массив значений, что и в предыдущем примере.

Если вам потребуется затем установить размер массива, обратитесь к компилятору, используя следующее выражение:

```
const USHORT IntegerArrayLength;
IntegerArrayLength = sizeof(IntegerArray)/sizeof(IntegerArray[0]);
```

В этом примере число элементов массива определяется как отношение размера массива в байтах к размеру одного элемента. Результат отношения сохраняется в переменной `IntegerArrayLength` типа `const USHORT`, которая была объявлена строкой выше.

Нельзя указывать в списке больше значений, чем заданное количество элементов массива. Так, следующее выражение вызовет показ компилятором сообщения об ошибке, поскольку массиву, состоящему из пяти элементов, пытаются присвоить шесть значений:

```
int IntegerArray[5] = {10, 20, 30, 40, 50, 60};
```

В то же время следующее выражение не будет ошибочным:

```
int IntegerArray[5] = {10, 20};
```

Значения тех элементов массива, которые не были инициализированы при объявлении, не устанавливаются. Обычно считают, что значения неинициализированных элементов массива нулевые. В действительности они могут содержать любой мусор — данные, которые когда-то ранее были занесены в эти ячейки памяти, что, в свою очередь, может оказаться источником ошибки.

Рекомендуется	Не рекомендуется
Позвольте компилятору самостоятельно вычислять размер массива. **Присваивайте** массивам информативные имена, раскрывающие их назначение. **Помните**, что для обращения к первому элементу массива следует указать индекс 0.	**Не записывайте** данные за пределы массива.

Объявление массивов

Массиву можно присвоить любое имя, но оно должно отличаться от имени всех других переменных и массивов в пределах видимости этого массива. Так, нельзя объявить массив `myCats[5]`, если в программе ранее уже была объявлена переменная `myCats`.

Размер массива при объявлении можно задать как числом, так и с помощью константы или перечисления, как показано в листинге 12.3.

Листинг 12.3. Использование константы и перечисления при объявлении массива

```
1:  // Листинг 12.3.
2:  // Установка размера массива с помощью константы и перечисления
3:
4:  #include <iostream.h>
5:  int main()
```

```
6:   {
7:     enum WeekDays {   Sun, Mon, Tue,
8:        Wed, Thu, Fri, Sat, DaysInWeek } ;
9:     int ArrayWeek[DaysInWeek] = {   10, 20, 30, 40, 50, 60, 70 } ;
10:
11:      cout << "The value at Tuesday is " << ArrayWeek[Tue];
12:   return 0;
13:   }
```

РЕЗУЛЬТАТ

```
The value at Tuesday is 30
```

АНАЛИЗ

В строке 7 объявляется перечисление `WeekDays`, содержащее восемь членов. Воскресенью (`Sunday`) соответствует значение `0`, а константе `DaysInWeek` — значение `7`.

В строке 11 константа перечисления `Tue` используется в качестве указателя на элемент массива. Поскольку константе `Tue` соответствует значение `2`, то в строке 11 возвращается и выводится на печать значение третьего элемента массива `ArrayWeek[2]`.

Массивы

Чтобы объявить массив, сначала нужно указать тип объектов, которые будут в нем сохранены, затем определить имя массива и задать размер массива. Размер определяет, сколько объектов заданного типа можно сохранить в данном массиве.

Пример 1:

```
int MyIntegerArray[90];
```

Пример 2:

```
long * ArrayOfPointersToLogs[100];
```

Чтобы получить доступ к элементам массива, используется оператор индексирования.

Пример 1:

```
Int theNinethInteger = MyIntegerArray[8];
```

Пример 2:

```
long * pLong = ArrayOfPointersToLogs[8];
```

Отсчет индексов массива ведется с нуля. Поэтому, для обращения к массиву, содержащему n элементов, используются индексы от `0` до `n-1`.

Массивы объектов

Любой объект, встроенный или созданный пользователем, может быть сохранен в массиве. Но для этого сначала нужно объявить массив и указать компилятору, для объектов какого типа этот массив создан и сколько объектов он может содержать. Компилятор вычислит, сколько памяти нужно отвести для массива, основываясь на размере объекта, заданном при объявлении класса. Если класс содержит конструктор, заданный по умолчанию, в котором не устанавливаются параметры, то объект класса может быть создан и сохранен в массиве одновременно с объявлением массива.

Получение доступа к данным переменных-членов объекта, сохраненного в массиве, идет в два этапа. Сначала с помощью оператора индексирования ([]) нужно указать элемент массива, а затем обратиться к конкретной переменной-члену с помощью оператора прямого обращения к члену класса (.). В листинге 12.4 показано создание массива для пяти объектов типа CAT.

Листинг 12.4. Создание массива объектов

```
1:  // Листинг 12.4. Массив объектов
2:
3:  #include <iostream.h>
4:
5:  class CAT
6:  {
7:    public:
8:       CAT() {   itsAge = 1; itsWeight=5; }
9:       ~CAT() { }
10:       int GetAge() const {   return itsAge; }
11:       int GetWeight() const {   return itsWeight; }
12:       void SetAge(int age) {   itsAge = age; }
13:
14:    private:
15:       int itsAge;
16:       int itsWeight;
17: } ;
18:
19: int main()
20: {
21:    CAT Litter[5];
22:    int i;
23:    for (i = 0; i < 5; i++)
24:       Litter[i].SetAge(2*i +1);
25:
26:    for (i = 0; i < 5; i++)
27:    {
28:       cout << "cat #" << i+1<< ": ";
29:       cout << Litter[i].GetAge() << endl;
30:    }
31: return 0;
32: }
```

Результат

```
cat #1: 1
cat #2: 3
cat #3: 5
cat #4: 7
cat #5: 9
```

Анализ

В строках 5–17 объявляется класс CAT. Чтобы объекты класса CAT могли создаваться при объявлении массива, в этом классе должен использовать-

ся конструктор, заданный по умолчанию. Вспомните, что если в классе создан какой-нибудь другой конструктор, то конструктор по умолчанию не будет предоставляться компилятором и вам придется создавать его самим.

Первый цикл `for` (строки 23 и 24) заносит значения возраста кошек в объекты класса, сохраненные в массиве. Следующий цикл `for` (строки 26–30) обращается к каждому объекту, являющемуся элементом массива, и вызывает для выбранного объекта метод `GetAge()`.

Чтобы применить метод `GetAge()` для объекта, являющегося элементом массива, используются последовательно операторы индексации (`[]`) и прямого доступа к члену класса (`.`), а также вызов функции-члена.

Многомерные массивы

Можно создать и использовать массив, содержащий более одного измерения. Доступ к каждому измерению открывается своим индексом. Так, чтобы получить доступ к элементу двухмерного массива, нужно указать два индекса; к элементу трехмерного массива — *три индекса и т.д. Теоретически можно создать массив любой мерности*, но, как правило, в программах используются одномерные и двухмерные массивы.

Хорошим примерным двухмерного массива является шахматная доска, состоящая из клеток, собранных в восемь рядов и восемь столбцов (рис. 12.3).

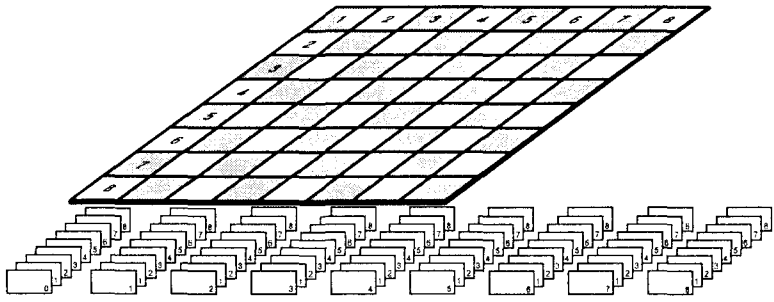


Рис. 12.3. Шахматная доска и двухмерный массив

Предположим, что в программе объявлен класс `SQUARE`. Объявление двухмерного массива `Board` для сохранения объектов этого класса будет выглядеть следующим образом:

```
SQUARE Board[8][8];
```

Эти же объекты можно было сохранить в одномерном массиве с 64 элементами:

```
SQUARE Board[64];
```

Использование двухмерного массива может оказаться предпочтительнее, если такой массив лучше отражает положение вещей в реальном мире, например при создании программы игры в шахматы. Так, в начале игры король занимает четвертую позицию в первом ряду. С учетом нулевого сдвига позиция этой фигуры будет представлена объектом массива:

```
Board[0][3];
```

В этом примере предполагается, что первый индекс будет контролировать нумерацию рядов, а второй — столбцов. Соответствие элементов массива клеткам шахматной доски наглядно показано на рис. 12.3.

Инициализация многомерного массива

Многомерный массив также можно инициализировать одновременно с объявлением. При этом следует учитывать, что сначала весь цикл значений проходит индекс, указанный последним, после чего изменяется предпоследний индекс. Таким образом, если есть массив

```
int theArray[5][3];
```

то первые три значения будут записаны в массив `theArray[0]`, вторые три значения — в массив `theArray[1]` и т.д.

Указанный массив можно инициализировать следующей строкой:

```
int theArray[5][3] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

Чтобы не запутаться в числах, значения можно сгруппировать с помощью дополнительных фигурных скобок, например:

```
int theArray[5][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,9},
                      {10,11,12},
                      {13,14,15}};
```

Компилятор проигнорирует все внутренние фигурные скобки.

Все значения должны быть разделены запятыми независимо от того, используете вы дополнительные фигурные скобки или нет. Весь список значений должен быть заключен во внешние фигурные скобки, и после закрывающей скобки обязательно устанавливается символ точки с запятой.

Пример создания двухмерного массива показан в листинге 12.5. Первый ряд двухмерного массива содержит целые числа от 0 до 4, а второй — удвоенные значения соответствующих элементов первого ряда.

Листинг 12.5. Создание многомерного массива

```
1:   #include <iostream.h>
2:   int main()
3:   {
4:     int SomeArray[5][2] = {   { 0,0} , { 1,2} , { 2,4} , { 3,6} , { 4,8} } ;
5:     for (int i = 0; i<5; i++)
6:        for (int j=0; j<2; j++)
7:        {
8:          cout << "SomeArray[" << i << "][" << j << "]: ";
9:          cout << SomeArray[i][j]<< endl;
10:       }
11:
12:   return 0;
13:   }
```

```
SomeArray[0][0]: 0
SomeArray[0][1]: 0
SomeArray[1][0]: 1
SomeArray[1][1]: 2
SomeArray[2][0]: 2
SomeArray[2][1]: 4
SomeArray[3][0]: 3
SomeArray[3][1]: 6
SomeArray[4][0]: 4
SomeArray[4][1]: 8
```

АНАЛИЗ В строке 4 объявляется двухмерный массив. Первый ряд содержит пять целочисленных значений, а второй ряд представлен двумя значениями. В результате создается конструкция из десяти элементов (5×2), как показано на рис. 12.4.

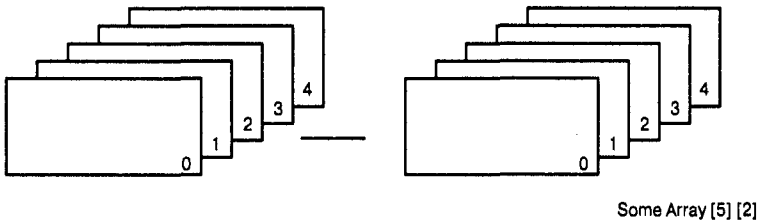


Рис. 12.4. Массив 5×2

Данные вводятся в массив попарно, хотя их можно было записать одной строкой. Затем осуществляется вывод данных с помощью двух вложенных циклов `for`. Внешний цикл последовательно генерирует индексы первого ряда, а внутренний — индексы второго ряда. В такой последовательности данные выводятся на экран: сначала идет элемент `SomeArray[0][0]`, затем элемент `SomeArray[0][1]`. Приращение индекса первого ряда происходит после того, как индекс второго ряда становится равным `1`, после чего вновь дважды выполняется внутренний цикл.

Несколько слов о памяти

При объявлении массива компилятору точно указывается, сколько объектов планируется в нем сохранить. Компилятор зарезервирует память для всех объектов массива, даже если далее в программе они не будут заданы. Если вы заранее точно знаете, сколько элементов должен хранить массив, то никаких проблем не возникнет. Например, шахматная доска всегда имеет только 64 клетки, а от кошки можно ожидать, что она не родит более 10 котят. Если же изначально неизвестно, сколько элементов будет в массиве, то для решения этой проблемы нужно использовать более гибкие средства управления памятью.

В этой книге рассматриваются только некоторые дополнительные средства программирования, такие как массивы указателей, массивы с резервированием памяти в области динамического обмена и ряд других возможностей. Больше информации о средствах программирования, открывающих дополнительные возможности, можно прочитать в моей книге *C++ Unleashed*, выпущенной издательством *Sams Publishing*. И вообще, всегда следует помнить, что каким бы хорошим программистом вы ни были, всегда остается то, чему следовало бы научиться, и всегда есть источники, откуда можно почерпнуть новую свежую информацию.

Массивы указателей

Все массивы, рассмотренные нами до сих пор, хранили значения своих элементов в стеках памяти. Использование стековой памяти связано с рядом ограничений, которых можно избежать, если обратиться к более гибкой области динамической памяти. Это можно сделать, если сначала сохранить все объекты массива в области динамической памяти, а затем собрать в массиве указатели на эти объекты. Этот подход значительно сократит потребление программой стековой памяти компьютера. В листинге 12.6 показан тот же массив, с которым мы работали в листинге 12.4, но теперь все его объекты сохранены в области динамической памяти. Чтобы показать возросшую эффективность использования памяти программой, в этом примере размер массива был увеличен с 5 до 500 и его название изменено с `Litter` (помет) на `Family` (семья).

Листинг 12.6. Сохранение массива в области динамической памяти

```
1:  // Листинг 12.6. Массив указателей на объекты
2:
3:  #include <iostream.h>
4:
5:  class CAT
6:  {
7:    public:
8:       CAT() {   itsAge = 1; itsWeight=5; }
9:       ~CAT() { }                          // destructor
10:       int GetAge() const {   return itsAge; }
11:       int GetWeight() const {   return itsWeight; }
12:       void SetAge(int age) {   itsAge = age; }
13:
14:     private:
15:       int itsAge;
16:       int itsWeight;
17: } ;
18:
19: int main()
20: {
21:    CAT * Family[500];
22:    int i;
23:    CAT * pCat;
24:    for (i = 0; i < 500; i++)
25:    {
26:      pCat = new CAT;
27:      pCat->SetAge(2*i +1);
28:      Family[i] = pCat;
29:    }
30:
31:    for (i = 0; i < 500; i++)
32:    {
33:      cout << "Cat #" << i+1 << ": ";
34:      cout << Family[i]->GetAge() << endl;
35:    }
36: return 0;
37: }
```

РЕЗУЛЬТАТ

```
Cat #1: 1
Cat #2: 3
Cat #3: 5
...
Cat #499: 997
Cat #500: 999
```

АНАЛИЗ

Объявление класса CAT в строках 5—17 идентично объявлению этого класса в листинге 12.4. Но, в отличие от предыдущего листинга, в строке 21 объявляется массив Family, в котором можно сохранить 500 указателей на объекты класса CAT.

В цикле инициализации (строки 24—29) в области динамической памяти создается 500 новых объектов класса CAT, каждому из которых присваивается значение переменной itsAge, равное удвоенному значению индекса плюс один. Таким образом, первому объекту класса CAT присваивается значение 1, второму — 3, третьему — 5 и т.д. В этом же цикле каждому элементу массива присваивается указатель на вновь созданный объект.

Поскольку тип массива был объявлен как CAT*, в нем сохраняются именно указатели, а не их разыменованные значения.

Следующий цикл (строки 31—35) выводит на экран все значения объектов, на которые делаются ссылки в массиве. Обращение к указателю выполняется с помощью индекса: Family[i]. После того как элемент массива установлен, следует вызов метода GetAge().

В данном примере программы все элементы массива сохраняются в стековой памяти. Но в этот раз элементами являются указатели, тогда как сами объекты хранятся в области динамического обмена.

Объявление массивов в области динамического обмена

Существует возможность поместить весь массив в область динамического обмена. Для этого используется ключевое слово new и оператор индексирования, как показано в следующем примере, где результатом этой операции является указатель на массив, сохраненный в области динамического обмена:

```
CAT *Family = new CAT[500];
```

Указатель Family будет содержать адрес в динамической области первого элемента массива из пятисот объектов класса CAT. Другими словами, в указателе представлен адрес объекта Family[0].

Еще одно преимущество подобного объявления массива состоит в том, что в программе с переменной Family теперь можно будет выполнять математические действия как с любым другим указателем, что открывает дополнительные возможности в управлении доступом к элементам массива. Например, можно выполнить следующие действия:

```
CAT *Family = new CAT[500];
CAT *pCat = Family;    // pCat указывает на Family[0]
pCat->SetAge(10);      // присваивает Family[0] значение 10
pCat++;                // переход к Family[1]
pCat->SetAge(20);      // присваивает Family[1] значение 20
```

В данном примере объявляется новый массив из 500 объектов класса CAT и возвращается указатель на первый элемент этого массива. Затем, используя это указатель и метод SetAge(), объявленный в классе CAT, первому объекту массива присваивается

значение 10. Переход к следующему объекту массива осуществляется за счет приращения адреса в указателе на массив, после чего тем же способом присваивается значение 20 второму объекту массива.

Указатель на массив или массив указателей

Рассмотрим следующие три объявления:

```
1:   Cat  FamilyOne[500];
2:   CAT * FamilyTwo[500];
3:   CAT * FamilyThree = new CAT[500];
```

В первом случае объявляется массив FamilyOne, содержащий 500 объектов типа CAT. Во втором случае — массив FamilyTwo, содержащий 500 указателей на объекты класса CAT, и в третьем случае — указатель FamilyThree, ссылающийся на массив из 500 объектов класса CAT.

В зависимости от того, какое объявление используется в программе, принципиально меняются способы управления массивом. Как ни странно, но указатель FamilyThree по сути своей гораздо ближе к массиву FamilyOne, но принципиально отличается от массива указателей FamilyTwo.

Чтобы разобраться в этом, следует внимательно рассмотреть, что содержат в себе все эти переменные. Указатель на массив FamilyThree содержит адрес первого элемента массива, но ведь это именно то, что содержит имя массива FamilyOne.

Имена массивов и указателей

В C++ имя массива представляет собой константный указатель на первый элемент массива. Другими словами, в объявлении

```
CAT Family[50];
```

создается указатель Family на адрес первого элемента массива &Family[0].

В программе допускается использование имен массивов как константных указателей и наоборот. Таким образом, выражению Family + 4 соответствует обращение к пятому элементу массива Family[4].

Компилятор выполняет с именами массивов те же математические действия сложения, инкремента и декремента, что и с указателями. В результате операция Family + 4 будет означать не прибавление четырех байтов к текущему адресу, а сдвиг на четыре объекта. Если размер одного объекта равен четырем байтам, то к адресу в имени массива будут добавлены не 4, а 16 байт. Если в нашем примере каждый объект класса CAT содержит четыре переменные-члена типа long по четыре байта каждая и две переменные-члена типа short по два байта каждая, то размер одного элемента массива будет равен 20 байт и операция Family + 4 сдвинет адрес в имени указателя на 80 байт.

Объявление массива в динамической области памяти и его использование показано в листинге 12.7.

Листинг 12.7. Создание массива с использованием ключевого слова new

```
1:  // Листинг 12.7. Массив в динамической области памяти
2:
3:  #include <iostream.h>
4:
5:  class CAT
6:  {
7:    public:
8:       CAT() {   itsAge = 1; itsWeight=5; }
9:       ~CAT();
10:       int GetAge() const {   return itsAge; }
11:       int GetWeight() const {   return itsWeight; }
12:       void SetAge(int age) {   itsAge = age; }
13:
14:     private:
15:       int itsAge;
16:       int itsWeight;
17: } ;
18:
19: CAT :: ~CAT()
20: {
21:   // cout << "Destructor called!\ n";
22: }
23:
24: int main()
25: {
26:    CAT * Family = new CAT[500];
27:    int i;
28:
29:    for (i = 0; i < 500; i++)
30:    {
31:      Family[i].SetAge(2*i +1);
32:    }
33:
34:    for (i = 0; i < 500; i++)
35:    {
36:      cout << "Cat #" << i+1 << ": ";
37:      cout << Family[i].GetAge() << endl;
38:    }
39:
40:    delete [] Family;
41:
42: return 0;
43: }
```

РЕЗУЛЬТАТ

```
Cat #1: 1
Cat #2: 3
Cat #3: 5
...
Cat #499: 997
Cat #500: 999
```

АНАЛИЗ

В строке 26 объявляется массив `Family` для пятисот объектов класса `CAT`. Благодаря использованию выражения `new CAT[500]` весь массив сохраняется в области динамической памяти.

Удаление массива из области динамической памяти

Куда деваются при удалении массива все эти объекты класса `CAT`, показанные в предыдущем разделе? Не происходит ли здесь утечка памяти? Удаление массива `Family` с помощью оператора `delete[]` (не забудьте установить квадратные скобки) освобождает все ячейки памяти, отведенные для него. Компилятор достаточно сообразительный, чтобы удалить из памяти все объекты удаляемого массива и освободить динамическую память для нового использования.

Чтобы убедиться в этом, измените размер массива в предыдущей программе с 500 на 10 в строках 26, 29 и 34. Затем разблокируйте выражение в строке 21 с оператором `cout` и запустите программу. Когда будет выполнена строка 43, последует десять вызовов деструктора для удаления каждого объекта класса `CAT` в массиве `Family`.

Создавая какой-либо объект в области динамической памяти с помощью ключевого слова `new`, всегда удаляйте его из памяти с помощью оператора `delete`, если этот объект больше не используется в программе. В случае создания массива в области динамического обмена выражением `new <class>[size]` удалять его из памяти нужно оператором `delete[]`. Квадратные скобки указывают, что удаляется весь массив.

Если вы забудете установить квадратные скобки, то из памяти будет удален только первый объект массива. В этом можно убедиться, если в нашем примере программы удалить квадратные скобки в строке 38. Если уже были внесены изменения в строку 21, как указывалось выше, то при выполнении программы на экране отобразится вызов только одного деструктора объекта, который удалит первый объект массива. Поздравляем вас! Вы потеряли огромный блок памяти для дальнейшего использования программой.

Рекомендуется

Помните, что для обращения к массиву из `n` элементов используются индексы от `0` до `n-1`.

Используйте свойства математических операций с указателями для управления доступом к элементам массива.

Не рекомендуется

Не записывайте данные за пределы массива.

Не путайте массив указателей с указателем на массив.

Массивы символов

Строка текста представляет собой набор символов. Все строки, которые до сих пор использовались нами в программах, представляли собой безымянные строковые константы, используемые в выражениях с оператором `cout`, такие как:

```
cout << "hello world.\n";
```

Но в C++ строку можно представить как массив символов, заканчивающийся *концевым нулевым символом строки*. Такой массив можно объявить и инициализировать точно так же, как любой другой массив, например:

```
char Greeting[] = { 'H', 'e', 'l', 'l', 'o', ' ', 'W','o','r','l','d', '\ 0' };
```

В последний элемент массива заносится нулевой концевой символ строки (`\0`), который многие функции C++ распознают как символ разрыва строки. Хотя метод ввода строки текста в массив символ за символом работает нормально, это довольно утомительная процедура, чреватая ошибками. К счастью, C++ допускает упрощенный метод ввода строк текста в массивы:

```
char Greeting[] = "hello world";
```

Обратите внимание на два момента данного синтаксиса.

- Вместо одиночных кавычек вокруг каждого символа, запятых между символами и фигурных скобок вокруг всей строки в данном примере используются только двойные кавычки вокруг строки и ничего более. Нет даже обычных для инициализации массивов фигурных скобок.
- Нет необходимости добавлять концевой нулевой символ, так как компилятор сделает это автоматически.

Строка `Hello World` займет 12 байт. Пять байтов пойдет на слово `Hello`, пять на слово `World` и по одному байту на пробел и концевой нулевой символ.

Инициализацию строкового массива можно оставить на потом. При этом, также как и с другими массивами, нужно следить, чтобы затем в массив не было записано символов больше, чем отводилось для этого места.

В листинге 12.8 показан пример использования массива символов, который инициализируется строкой, вводимой пользователем с клавиатуры.

Листинг 12.8. Заполнение массива символами

```
1:   //Листинг 12.8. Заполнение массива символами
2:
3:   #include <iostream.h>
4:
5:   int main()
6:   {
7:     char buffer[80];
8:    cout << "Enter the string: ";
9:    cin >> buffer;
10:    cout << "Here is' the buffer: " << buffer << endl;
11:   return 0;
12:   }
```

```
Enter the string: Hello World
Here's the buffer: Hello
```

В строке 7 объявляется массив `buffer`, рассчитанный на 80 символов. Такой массив может содержать строку из 79 букв, включая пробелы, плюс нулевой концевой символ строки.

В строке 8 пользователю предлагается ввести строку текста, которая копируется в массив `buffer` в строке 9. Метод `cin` автоматически добавит нулевой концевой символ в конце введенной строки.

Но при выполнении программы, показанной в листинге 12.8, возникает ряд проблем. Во-первых, если пользователь введет строку, содержащую более 79 символов, то оператор `cin` введет их за пределами массива. Во-вторых, оператор `cin` воспринимает пробел как окончание строки, после чего прекращает ввод данных.

Чтобы решить эти проблемы, нужно использовать метод `get()`, применяемый вместе с оператором `cin`: `cin.get()`. Для выполнения метода нужно задать три параметра.

- Буфер ввода.
- Максимальное число символов.
- Разделительный символ прерывания ввода.

По умолчанию в качестве разделительного задается символ разрыва строки. Использование этого метода показано в листинге 12.9.

Листинг 12.9. Заполнение массива

```
1:   //Листинг 12.9. Использование метода cin.get()
2:
3:   #include <iostream.h>
4:
5:   int main()
6:   {
7:     char buffer[80]>
8:     cout << "Enter the string: ";
9:     cin.get(buffer, 79); // ввод завершается после 79 символа или символа разрыва строки
10:    cout << "Here's the buffer: " << buffer << endl;
11:    return 0;
12:  }
```

```
Enter the string: Hello World
Here's the buffer: Hello World
```

В строке 9 осуществляется вызов метода `cin.get()`. Буфер ввода, заданный в строке 7, передается в функцию как первый аргумент. Второй аргумент задает максимальную длину строки, равную 79 символам. Допускается ввод только 79 символов, поскольку последний элемент массива отводится на концевой нулевой символ строки. Устанавливать третий аргумент не обязательно. В большинстве случаев в качестве разделительного символа подходит задаваемый по умолчанию символ разрыва строки.

Функции strcpy() и strncpy()

Язык C++ унаследовал от C библиотечные функции, выполняющие операции над строками. Среди множества доступных функций есть две, которые осуществляют копирование одной строки в другую. Это функции `strcpy()` и `strncpy()`. Функция `strcpy()` копирует строку целиком в указанный буфер, как показано в листинге 12.10.

Листинг 12.10. Использование функции `strcpy()`

```
1:   #include <iostream.h>
2:   #include <string.h>
3:   int main()
4:   {
5:    char String1[] = "No man is an island";
6:    char String2[80];
7:
8:    strcpy(String2,String1);
9:
10:    cout << "String1: " << String1 << endl;
11:    cout << "String2: " << String2 << endl;
12:   return 0;
13:   }
```

РЕЗУЛЬТАТ

```
String1: No man is an island
String2: No man is an island
```

АНАЛИЗ Файл заголовка `strin.h` включается в программу в строке 2. Этот файл содержит прототип функции `strcpy()`. В качестве аргументов функции указываются два массива символов, первый из которых является целевым, а второй — массивом источника данных. Если массив-источник окажется больше целевого массива, то функция `strcpy()` введет данные за пределы массива.

Чтобы предупредить подобную ошибку, в этой библиотеке функций содержится еще одна функция копирования строк: `strncpy()`. Эта функция копирует ряд символов, не превышающий длины строки, заданной в целевом массиве. Функция `strncpy()` также прерывает копирование, если ей повстречается символ разрыва строки.

Использование функции `strncpy()` показано в листинге 12.11.

Листинг 12.11. Использование функции `strncpy()`

```
1:   #include <iostream.h>
2:   #include <string.h>
3:   int main()
4:   {
5:     const int MaxLength = 80;
6:    char String1[] = "No man is an island";
7:    char String2[MaxLength+1];
8:
9:
```

```
10:    strncpy(String2,String1,MaxLength);
11:
12:    cout << "String1: " << String1 << endl;
13:    cout << "String2: " << String2 << endl;
14:   return 0;
15:   }
```

РЕЗУЛЬТАТ

```
String1: No man is an island
String2: No man is an island
```

АНАЛИЗ В строке 10 программа вместо функции `strcpy()` используется функцию `strncpy()`, третий параметр `MaxLength` которой задает максимальную длину копируемой строки. Размер массива `String2` задан как `MaxLength+1`. Дополнительный элемент потребовался для концевого нулевого символа строки, который добавляется автоматически обеими функциями — `strcpy()` и `strncpy()`.

Классы строк

Многие компиляторы C++ содержат библиотеки классов, с помощью которых можно решать различные прикладные задачи. Одним из представителей встроенных классов является класс `String`.

Язык C++ унаследовал от C концевой нулевой символ окончания строки и библиотеку строковых функций, куда входит функция `strcpy()`. Но все эти функции нельзя использовать в объектно-ориентированном программировании. Класс `String` предлагает набор встроенных функций-членов и переменных-членов, а также методов доступа, которые позволяют автоматически решать многие задачи, связанные с обработкой текстовых строк, получая команды от пользователя.

Если в вашем компиляторе нет встроенного класса `String`, а иногда и в тех случаях, когда он есть, бывает необходимо создать собственный класс работы со строками. Далее в этой главе рассматривается процедура создания и применения класса `String` и пользовательских классов работы со строками.

Как минимум, класс `String` должен преодолеть ограничения, свойственные использованию массивов символов. Подобно другим массивам, массивы символов статичны. Вам приходится задавать их размер при объявлении или инициализации. Они всегда занимают все отведенное для них пространство памяти, даже если вы используете только половину элементов массива. Запись данных за пределы массива ведет к катастрофе.

Хорошо написанный класс работы со строковыми данными выделяет столько памяти, сколько необходимо для текущего сеанса работы с программой, и всегда предусматривает возможность добавления новых данных. Если с выделением дополнительной памяти возникнут проблемы, предусмотрены элегантные пути их решения.

Первый пример использования класса `String` показан в листинге 12.12.

Листинг 12.12. Использование класса `String`

```
1:   //Листинг. 12.12
2:
3:   #include <iostream.h>
4:   #include <string.h>
```

```
5:
6:  // Рудиментарный класс string
7:  class String
8:  {
9:    public
10:       // Конструкторы
11:       String()
12:       String(const char *const),
13:       String(const String &),
14:       ~String()
15:
16:       // Перегруженные операторы
17:       char & operator[](unsigned short offset),
18:       char operator[](unsigned short offset) const,
19:       String operator+(const String&),
20:       void operator+=(const String&)
21:       String & operator= (const String &),
22:
23:       // Основные методы доступа
24:       unsigned short GetLen()const {   return itsLen, }
25:       const char * GetString() const {   return itsString, }
26:
27:   private
28:       String (unsigned short), // Закрытый конструктор
29:       char * itsString,
30:       unsigned short itsLen
31:  }
32:
33:  // Конструктор, заданный по умолчанию, создает строку нулевой длины
34:  String  String()
35:  {
36:     itsString = new char[1]
37:     itsString[0] = '\ 0 ,
38:     itsLen=0,
39:  }
40:
41:  // Закрытый (вспомогательный) конструктор
42:  // используется только методами класса для создания
43:  // строк требуемой длины с нулевым наполнением
44:  String  String(unsigned short len)
45:  {
46:     itsString = new char[len+1]
47:     for (unsigned short i = 0  i<=len, i++)
48:       itsString[i] =  \ 0 ,
49:     itsLen=len,
50:  }
51:
52:  // Преобразование массива символов в строку
53:  String  String(const char * const cString)
54:  {
```

```
55:     itsLen = strlen(cString);
56:     itsString = new char[itsLen+1];
57:     for (unsigned short i = 0; i<itsLen; i++)
58:       itsString[i] = cString[i];
59:     itsString[itsLen]='\ 0';
60:   }
61:
62:   // Конструктор-копировщик
63:   String::String (const String & rhs)
64:   {
65:     itsLen=rhs.GetLen();
66:     itsString = new char[itsLen+1];
67:     for (unsigned short i = 0; i<itsLen;i++)
68:       itsString[i] = rhs[i];
69:     itsString[itsLen] = '\ 0';
70:   }
71:
72:   // Деструктор для освобождения памяти
73:   String::~String ()
74:   {
75:     delete [] itsString;
76:     itsLen = 0;
77:   }
78:
79:   // Оператор присваивания освобождает память
80:   // и копирует туда string и size
81:   String& String::operator=(const String & rhs)
82:   {
83:     if (this == &rhs)
84:       return *this;
85:     delete [] itsString;
86:     itsLen=rhs.GetLen();
87:     itsString = new char[itsLen+1];
88:     for (unsigned short i = 0; i<itsLen;i++)
89:       itsString[i] = rhs[i];
90:     itsString[itsLen] = '\ 0';
91:     return *this;
92:   }
93:
94:   //неконстантный оператор индексирования
95:   // возвращает ссылку на символ так, что его
96:   // можно изменить!
97:   char & String::operator[](unsigned short offset)
98:   {
99:     if (offset > itsLen)
100:      return itsString[itsLen-1];
101:    else
102:     return itsString[offset];
103:  }
104:
```

```
105:  // константный оператор индексирования для использования
106:  // с константными объектами (см. конструктор-копировщик!)
107:  char String::operator[](unsigned short offset) const
108:  {
109:    if (offset > itsLen)
110:     return itsString[itsLen-1];
111:    else
112:     return itsString[offset];
113:  }
114:
115:  // создание новой строки путем добавления
116:  // текущей строки к rhs
117:  String String::operator+(const String& rhs)
118:  {
119:    unsigned short   totalLen = itsLen + rhs.GetLen();
120:    String temp(totalLen);
121:     unsigned short i;
122:    for ( i= 0; i<itsLen; i++)
123:     temp[i] = itsString[i];
124:    for (unsigned short j = 0; j<rhs.GetLen(); j++, i++)
125:     temp[i] = rhs[j];
126:    temp[totalLen]='\ 0';
127:    return temp;
128:  }
129:
130:  // изменяет текущую строку и возвращает void
131:  void String::operator+=(const String& rhs)
132:  {
133:    unsigned short rhsLen = rhs.GetLen();
134:    unsigned short totalLen = itsLen + rhsLen;
135:    String   temp(totalLen);
136:       unsigned short i;
137:    for (i = 0; i<itsLen; i++)
138:     temp[i] = itsString[i];
139:    for (unsigned short j = 0; j<rhs.GetLen(); j++, i++)
140:     temp[i] = rhs[i-itsLen];
141:    temp[totalLen]='\ 0';
142:    *this = temp;
143:  }
144:
145:  int main()
146:  {
147:    String s1("initial test");
148:    cout << "S1:\ t" << s1.GetString() << endl;
149:
150:    char * temp = "Hello World";
151:    s1 = temp;
152:    cout << "S1:\ t" << s1.GetString() << endl;
153:
154:    char tempTwo[20];
```

```
155:    strcpy(tempTwo,"; nice to be here!");
156:    s1 += tempTwo;
157:    cout << "tempTwo:\ t" << tempTwo << endl;
158:    cout << "S1:\ t" << s1.GetString() << endl;
159:
160:    cout << "S1[4]:\ t" << s1[4] << endl;
161:    s1[4]='o';
162:    cout << "S1:\ t" << s1.GetString() << endl;
163:
164:    cout << "S1[999]:\ t" << s1[999] << endl;
165:
166:    String s2(" Another string");
167:    String s3;
168:    s3 = s1+s2;
169:    cout << "S3:\ t" << s3.GetString() << endl;
170:
171:    String s4;
172:    s4 = "Why does this work?";
173:    cout << "S4:\ t" << s4.GetString() << endl;
174:  return 0;
175:  }
```

РЕЗУЛЬТАТ

```
S1:     initial test
S1:     Hello world
tempTwo:        ; nice to be here!
S1:     Hello world; nice to be here!
S1[4]:    o
S1:     Hello World; nice to be here!
S1[999]:        !
S3:     Hello World; nice to be here! Another string
S4:     Why does this work?
```

АНАЛИЗ В строках 7–31 объявляется простой класс `String`. В строках 11–13 объявляются конструктор по умолчанию, конструктор-копировщик и конструктор для приема существующей строки с концевым нулевым символом (стиль языка C).

В классе `String` перегружаются операторы индексирования (`[]`), суммирования (`+`) и присваивания с суммой (`+=`). Оператор индексирования перегружается дважды. Один раз как константная функция, возвращающая значение типа `char`, а другой — как неконстантная функция, возвращающая указатель на `char`.

Неконстантная версия оператора используется в выражениях вроде строки 161:

```
SomeString[4]='x';
```

В результате открывается прямой доступ к любому символу строки. Поскольку возвращается ссылка на символ, функция получает доступ к символу и может изменить его.

Константная версия оператора используется в тех случаях, когда необходимо получить доступ к константному объекту класса `String`, например при выполнении конструктора-копировщика в строке 63. Обратите внимание, что в этом случае открывается доступ к `rhs[i]`, хотя `rhs` был объявлен как `const String &`. К этому объекту невозможно получить доступ, используя неконстантные функции-члены. Поэтому оператор индексирования необходимо перегрузить как константный.

Если возвращаемый объект окажется слишком большим, возможно, вам потребуется установить возврат не значения, а константной ссылки на объект. Но поскольку в нашем случае один символ занимает всего один байт, в этом нет необходимости.

Конструктор, заданный по умолчанию, выполняется в строках 33—39. Он создает строку нулевой длины. Общепринято, что в классе `String` длина строки измеряется без учета концевого нулевого символа. Таким образом, строка, созданная по умолчанию, содержит только концевой нулевой символ.

Конструктор-копировщик выполняется в строках 63—70. Он задает длину новой строки равной длине существующей строки плюс одна ячейка для концевого нулевого символа. Затем конструктор-копировщик копирует существующую строку в новую и добавляет в конце нулевой символ окончания строки.

В строках 53—60 выполняется конструктор, принимающий строку с концевым нулевым символом. Этот конструктор подобен конструктору-копировщику. Длина существующей строки определяется с помощью стандартной функции `strlen()` из библиотеки `String`.

В строке 28 объявляется еще один конструктор, `String(unsigned short)`, как закрытая функция-член. Он был добавлен для того, чтобы не допустить создания в классе `String` строк произвольной длины каким-нибудь другим пользовательским классом. Этот конструктор позволяет создавать строки только внутри класса `String` в соответствии со сделанными установками, как, например, в строке 131 с помощью `operator+=`. Более подробно этот вопрос рассматривается ниже, при объявлении `operator+=`.

Конструктор `String(unsigned short)` заполняет все элементы своего массива символов значениями `NULL`. Поэтому в цикле `for` выполняется проверка `i<=len`, а не `i<len`.

Деструктор, выполняемый в строках 73—77, удаляет строку текста, поддерживаемую классом `String`. Обратите внимание, что за оператором `delete` следуют квадратные скобки. Если опустить их, то из памяти компьютера будут удалены не все объекты класса, а только первый из них.

Оператор присваивания прежде всего определяет, не соответствуют ли друг другу операнды слева и справа. Если операнды отличаются друг от друга, то текущая строка удаляется, а новая копируется в эту область памяти. Чтобы упростить присвоение, возвращается ссылка на строку, как в следующем примере:

```
String1 = String2 = String3;
```

Оператор индексирования перегружается дважды. В обоих случаях проверяются границы массива. Если пользователь попытается возвратить значение из ячейки памяти, находящейся за пределами массива, будет возвращен последний символ массива (`len-1`).

В строках 117—128 оператор суммирования (+) выполняется как оператор конкатенации. Поэтому допускается создание новой строки из двух строк, как в следующем выражении:

```
String3 = String1 + String2;
```

Оператор (+) вычисляет длину новой строки и сохраняет ее во временной строке `temp`. Эта процедура вовлекает закрытый конструктор, который принимает целочисленный параметр и создает строку, заполненную значениями `NULL`. Нулевые значения затем замещаются символами двух строк. Первой копируется строка левого операнда (`*this`), после чего — строка правого операнда (`rhs`).

Первый цикл `for` последовательно добавляет в новую строку символы левой строки. Второй цикл `for` выполняет ту же операцию с правой строкой. Обратите внимание, что счетчик `i` продолжает отсчет символов новой строки после того, как счетчик `j` начинает отсчет символов строки `rhs`.

Оператор суммирования возвращает временную строку `temp` как значение, которое присваивается строке слева от оператора присваивания (`string1`). Оператор `+=` манипулирует с уже существующими строками, как в случае `string1 += string2`. В этом примере оператор `+=` действует так же, как оператор суммирования, но значение временной строки `temp` присваивается не новой, а текущей строке (`*this = temp`), как в строке 142.

Функция `main()` (строки 145—175) выполняется для проверки результатов работы данного класса. В строке 147 создается объект `String` с помощью конструктора, задающего строки в стиле языка C с концевым нулевым символом. Строка 148 выводит содержимое этого объекта с помощью функции доступа `GetString()`. В строке 150 создается еще одна строка текста в стиле языка C. В строке 151 тестируется перегруженный оператор присваивания, а строка 152 выводит результат.

В строке 154 создается третья строка с концевым нулевым символом — `tempTwo`. В строке 155 с помощью функции `strcpy()` происходит заполнение буфера строкой символов `nice to be here!`. В строке 156 с помощью перегруженного оператора `+=` осуществляется конкатенация строки `tempTwo` к существующей строке `s1`. Результат выводится на экран в строке 158.

В строке 160 возвращается и выводится на экран пятый символ строки — `s1`. Затем в строке 161 этот символ замещается другим с помощью неконстантного оператора индексирования (`[]`). Результат выводится строкой 162, чтобы показать, что символ строки действительно изменился.

В строке 164 делается попытка получить доступ к символу за пределами массива. Возвращается и выводится на печать последний символ строки, как и было предусмотрено при перегрузке оператора индексирования.

В строках 166 и 167 создаются два дополнительных объекта `String`, и в строке 168 используется перегруженный оператор суммирования. Результат выводится строкой 169.

В строке 171 создается еще один объект класса `String` — `s4`. В строке 172 используется оператор присваивания, а строка 173 выводит результат. Оператор присваивания перегружен таким образом, чтобы использовать константную ссылку класса `String`, объявленную в строке 21, но в данном случае в функцию передается строка с концевым нулевым символом. Разве это допустимо?

Хотя компилятор, ожидая получить объект `String`, вместо этого получает массив символов, он автоматически проверяет возможность преобразования полученного значения в ожидаемую строку. В строке 12 объявляется конструктор, который создает объект `String` из массива символов. Компилятор создает временный объект `String` из полученного массива символов и передает его в функцию оператора присваивания. Такой процесс называется *неявным преобразованием*. Если бы в программе не был объявлен соответствующий конструктор, преобразующий массивы символов, то для этой строки компилятор показал бы сообщение об ошибке.

Связанные списки и другие структуры

Массивы являются отличными контейнерами для данных самых разных типов. Единственный их недостаток состоит в том, что при создании массива необходимо задать его фиксированный размер. Если на всякий случай создать слишком большой массив, то попусту будет потрачено много памяти компьютера. Если сэкономить память, то возможности программы по оперированию данными окажутся ограниченными.

Один из способов решения этой проблемы состоит в использовании связанных списков. *Связанный список* представляет собой структуру данных, состоящую из взаимосвязанных блоков, каждый из которых может поддерживать структурную единицу

данных. Идея состоит в том, чтобы создать класс, поддерживающий объекты данных определенного типа, такого как CAT или Rectangle, которые, помимо данных, содержали бы также указатели, связанные с другими объектами этого класса. Таким образом, получается класс, содержащий взаимосвязанные объекты, образующие произвольную структуру-список.

Такие объекты называют *узлами*. Первый узел в списке образует голову, а последний — хвост.

Существует три основных типа связанных списков. Ниже они перечислены в порядке усложнения.

- Однонаправленные списки.
- Двунаправленные списки.
- Деревья.

В однонаправленных связанных списках каждый узел указывает на следующий узел только в одном направлении. Движение по узлам в обратном направлении невозможно. Чтобы найти нужный узел, следует начать с первого узла и двигаться от узла к узлу, подобно кладоискателю, действующему согласно указаниям карты поиска сокровищ: "...от большого камня иди к старому дубу, сделай три шага на восток и начинай копать..." Двунаправленные списки позволяют осуществлять движение в обоих направлениях по цепи. Деревья представляют собой более сложные структуры, в которых один узел может содержать ссылки на два или три следующих узла. Все три типа связанных списков схематично показаны на рис. 12.5.

Общие представления о связанных списках

В данном разделе обсуждаются основные моменты создания сложных структур и, что еще более важно, возможности использования в больших проектах наследования, полиморфизма и инкапсуляции.

Делегирование ответственности

Основная идея объектно-ориентированного программирования состоит в том, что каждый объект специализируется в выполнении определенных задач и передает другим объектам ответственность за выполнение тех задач, которые не соответствуют их основному предназначению.

Примером реализации этой идеи в технике может быть автомобиль. Назначение двигателя — вырабатывать свободную энергию. Распределение энергии уже не входит в круг задач двигателя. За это ответственна трансмиссия. И в конце концов, движение автомобиля за счет отталкивания от дороги осуществляется с помощью колес, а двигатель и трансмиссия принимают в этом деле существенное, но косвенное участие.

Хорошо сконструированная машина состоит из множества деталей с четким распределением функций и структурным взаимодействием между ними, обеспечивающим решение сложных задач. Так же должна выглядеть хорошо написанная программа: каждый класс вплетает свою нить, а в результате получается шикарный персидский ковер.

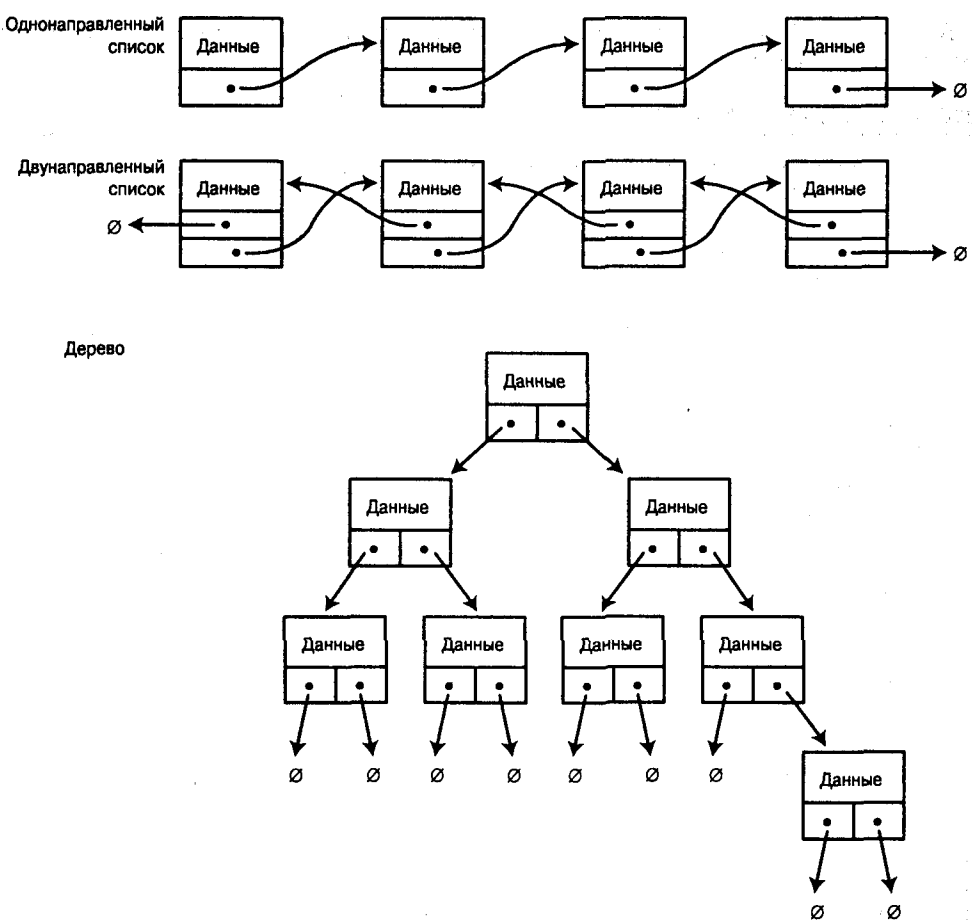


Рис. 12.5. Связанные списки

Компоненты связанных списков

Связанный список состоит из узлов. Узлы представляют собой абстрактные классы. В нашем примере для построения связанного списка используются три подтипа данных. Один узел будет представлять голову связанного списка и отвечать за его инициализацию. Попробуйте догадаться сами, за что отвечает хвостовой узел. Между ними могут быть представлены (либо могут отсутствовать) один или несколько промежуточных узлов, которые отвечают за обработку данных, переданных в список.

Обратите внимание, что данные списка и сам список — это не одно и то же. В списке могут быть представлены данные любого типа, но связываются друг с другом не данные, а узлы, которые *содержат* данные.

Выполняемой части программы ничего не известно об узлах, она работает со связанным списком как с единым целым. В то же время функциональная нагрузка на список как таковой весьма ограничена — он просто распределяет ответственность за выполнение задач между узлами.

В листинге 12.13 рассматривается пример программы со связанным списком, а затем детально анализируется ее работа.

Листинг 12.13. Связанный список

```
0:    // **********************************************
1:    // Листинг 12.13.
2:    //
3:    //   ЦЕЛЬ:   Показать использование связанного списка
4:    //   ПРИМЕЧАНИЯ:
5:    //
6:    //   Авторское право:   Copyright (C) 1998 Liberty Associates, Inc.
7:    //          Все права защищены
8:    //
9:    // Показан один из подходов объектно-ориентированного
10:   // программирования по созданию связанных списков.
11:   // Список распределяет задачи между узлами,
12:   // представляющими собой абстрактные типы данных.
13:   // Список состоит из трех узлов: головного,
14:   // хвостового и промежуточного. Данные содержит
15:   // только промежуточный узел.
16:   // Все объекты, используемые в списке, относятся
17:   // к классу Data.
18:   // **********************************************
19:
20:
21:   #include <iostream.h>
22:
23:   enum {   kIsSmaller, kIsLarger, kIsSame} ;
24:
25:   // Связанный список основывается на объектах класса Data
26:   // Любой класс в связанном списке должен поддерживать два метода:
27:   // Show (отображение значения) и Compare (возвращение относительной позиции узла)
28:   class Data
29:   {
30:   public:
31:     Data(int val):myValue(val){ }
32:     ~Data(){ }
33:     int Compare(const Data &);
34:     void Show() {   cout << myValue << endl; }
35:   private:
36:     int myValue;
37:   } ;
38:
39:   // Сравнение используется для определения
40:   // позиции в списке для нового узла.
41:   int Data::Compare(const Data & theOtherData)
42:   {
43:     if (myValue < theOtherData.myValue)
44:        return kIsSmaller;
45:     if (myValue > theOtherData.myValue)
```

```
46:       return kIsLarger;
47:    else
48:       return kIsSame;
49:  }
50:
51:  // Объявления
52:  class Node;
53:  class HeadNode;
54:  class TailNode;
55:  class InternalNode;
56:
57:  // ADT-представление узловых объектов списка.
58:  // В каждом производном классе должны быть замещены функции Insert и Show
59:  class Node
60:  {
61:  public:
62:    Node(){ }
63:    virtual ~Node(){ }
64:    virtual Node * Insert(Data * theData)=0;
65:    virtual void Show() = 0;
66:  private:
67:  } ;
68:
69:  // Этот узел поддерживает реальные объекты.
70:  // В данном случае объект имеет тип Data
71:  // О другом, более общем методе решения этой
72:  // задачи мы узнаем при рассмотрении шаблонов.
73:  class InternalNode: public Node
74:  {
75:  public:
76:    InternalNode(Data * theData, Node * next);
77:    ~InternalNode(){   delete myNext; delete myData; }
78:    virtual Node * Insert(Data * theData);
79:    virtual void Show() {   myData->Show(); myNext->Show(); }   // Делегирование!
80:
81:  private:
82:       Data * myData;   // данные списка
83:       Node * myNext;   // указатель на следующий узел в связанном списке
84:  } ;
85:
86:  // Инициализация, выполняемая каждым конструктором
87:  InternalNode::InternalNode(Data * theData, Node * next):
88:  myData(theData),myNext(next)
89:  {
90:  }
91:
92:  // Сущность списка.
93:  // Когда в список передается новый объект,
94:  // программа определяет позицию в списке
95:  // для нового узла
```

```
96:  Node * InternalNode::Insert(Data * theData)
97:  {
98:
99:    // Этот новенький больше или меньше чем я?
100:    int result = myData->Compare(*theData);
101:
102:
103:    switch(result)
104:    {
105:         // По соглашению, если он такой же как я, то он идет первым
106:    case kIsSame:      // условие выполняется
107:    case kIsLarger:    // новые данные вводятся перед моими
108:       {
109:         InternalNode * dataNode = new InternalNode(theData, this);
110:         return dataNode;
111:       }
112:
113:       // Он больше чем я, поэтому передается в
114:       // следующий узел, и пусть тот делает с этими данными все, что захочет.
115:    case kIsSmaller:
116:         myNext = myNext->Insert(theData);
117:         return this;
118:    }
119:    return this;    // появляется MSC
120:  }
121:
122:
123:  // Хвостовой узел выполняет роль часового
124:
125:  class TailNode : public Node
126:  {
127:  public:
128:    TailNode(){ }
129:    ~TailNode(){ }
130:    virtual Node * Insert(Data * theData);
131:    virtual void Show() {   }
132:
133:  private:
134:
135:  } ;
136:
137:  // Если данные подходят для меня, то они должны быть вставлены передо мной,
138:  // так как я хвост и НИЧЕГО не может быть после меня
139:  Node * TailNode::Insert(Data * theData)
140:  {
141:    InternalNode * dataNode = ew InternalNode(theData, this);
142:    return dataNode;
143:  }
144:
145:  // Головной узел не содержит данных, он только
```

```
146:  // указывает на начало списка
147:  class HeadNode : public Node
148:  {
149:  public:
150:    HeadNode();
151:    ~HeadNode() {   delete myNext; }
152:    virtual Node * Insert(Data * theData);
153:    virtual void Show() {   myNext->Show(); }
154:  private:
155:    Node * myNext;
156:  } ;
157:
158:  // Как только создается головной узел,
159:  // он создает хвост
160:  HeadNode::HeadNode()
161:  {
162:    myNext = new TailNode;
163:  }
164:
165:  // Ничего не может быть перед головой, поэтому
166:  // любые данные передаются в следующий узел
167:  Node * HeadNode::Insert(Data * theData)
168:  {
169:    myNext = myNext->Insert(theData);
170:    return this;
171:  }
172:
173:  // Я только распределяю задачи между узлами
174:  class LinkedList
175:  {
176:  public:
177:    LinkedList();
178:    ~LinkedList() {   delete myHead; }
179:    void Insert(Data * theData);
180:    void ShowAll() {   myHead->Show(); }
181:  private:
182:    HeadNode * myHead;
183:  } ;
184:
185:  // Список появляется с созданием головного узла,
186:  // который сразу создает хвостовой узел.
187:  // Таким образом, пустой список содержит указатель на головной узел,
188:  // указывающий, в свою очередь, на хвостовой узел, между которыми пока ничего
нет.
189:  LinkedList::LinkedList()
190:  {
191:    myHead = new HeadNode;
192:  }
193:
194:  // Делегирование, делегирование, делегирование
```

```
195:  void LinkedList::Insert(Data * pData)
196:  {
197:    myHead->Insert(pData);
198:  }
199:
200:  // выполняемая тестовая программа
201:  int main()
202:  {
203:    Data * pData;
204:    int val;
205:    LinkedList ll;
206:
207:    // Предлагает пользователю ввести значение,
208:    // которое передается в список
209:    for (;;)
210:    {
211:       cout << "What value? (0 to stop): ";
212:       cin >> val;
213:       if (!val)
214:         break;
215:       pData = new Data(val);
216:       ll.Insert(pData);
217:    }
218:
219:    // теперь пройдемся по списку и посмотрим значения
220:    ll.ShowAll();
221:    return 0;   // ll выходит за установленные рамки и поэтому удалено!
222:  }
```

РЕЗУЛЬТАТ

```
What value? (0 to stop): 5
What value? (0 to stop): 8
What value? (0 to stop): 3
What value? (0 to stop): 9
What value? (0 to stop): 2
What value? (0 to stop): 10
What value? (0 to stop): 0
2
3
5
8
9
10
```

АНАЛИЗ Первое, на что следует обратить внимание, — это константное перечисление, в котором представлены константы `kIsSmaller`, `kIsLarger` и `kIsSame`. Любой объект, представленный в списке, должен поддерживать метод `Compare()`. Константы, показанные выше, возвращаются в результате выполнения этого метода.

В строках 28–37 объявляется класс Data, а в строках 39–49 выполняется метод Compare(). Объекты класса Data содержат данные и могут использоваться для сравнения с другими объектами класса Data. Они также поддерживают метод Show(), отображающий значение объекта класса Data.

Чтобы лучше разобраться в работе связанного списка, проанализируем шаг за шагом выполнение программы, показанной выше. В строке 201 объявляется выполняемый блок программы, в строке 203 — указатель на класс Data, а в строке 205 определяется связанный список.

Для создания связанного списка в строке 189 вызывается конструктор. Единственное, что он делает, — выделяет области памяти для объекта HeadNode и присваивает адрес объекта указателю, поддерживаемому связанным списком и объявленному в строке 182.

При создании объекта HeadNode вызывается еще один конструктор, объявленный в строках 160–163, который, в свою очередь, создает объект TailNode и присваивает его адрес указателю myNext, содержащемуся в объекте HeadNode. При создании объекта TailNode вызывается конструктор TailNode, объявленный в строке 128. Тело конструктора содержится в той же строке программы, и он не создает никаких новых объектов.

Таким образом, создание связанного списка вызывает последовательность взаимосвязанных процессов, в результате которых для него выделяется область стековой памяти, создаются головной и хвостовой узлы и устанавливаются взаимосвязи между ними, как показано на рис. 12.6.

В строке 209 начинается бесконечный цикл. Появляется предложение пользователю ввести значение, которое будет добавлено в связанный список. Ввод новых значений можно продолжать до бесконечности. Ввод значения 0 завершает цикл. Введенное значение проверяется в строке 213.

Если введенное значение отличается от 0, то в строке 215 создается новый объект типа Data, а в строке 216 он вводится в связанный список. Предположим, что пользователь ввел число 15, после чего в строке 195 будет вызван метод Insert.

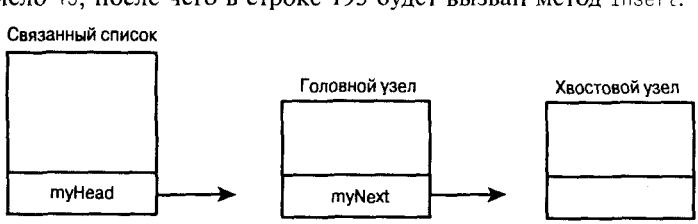


Рис. 12.6. Связанный список сразу после создания

Связанный лист немедленно передаст ответственность за ввод объекта головному узлу, вызвав в строке 167 метод Insert. Головной узел немедленно делегирует ответственность любому другому узлу (вызывает в строке 139 метод Insert), адрес которого хранится в указателе myNext. Сначала в этом указателе представлен адрес хвостового узла (вспомните, что при создании головного узла автоматически создается хвостовой узел и ссылка на него добавляется в головной узел).

Хвостовому узлу TailNode известно, что любой объект, переданный обращением TailNode::Insert, нужно добавить в список непосредственно перед собой. Так, в строке 141 создается объект InternalNode, который добавляется в список перед хвостовым узлом и принимает введенные данные и указатель на хвостовой узел. Эта процедура выполняется с помощью объявленного в строке 87 конструктора объекта InternalNode.

Конструктор объекта InternalNode просто инициализирует указатель класса Data адресом переданного объекта этого класса, а также присваивает указателю myNext этого

объекта адрес того узла, из которого он был передан. В случае создания первого промежуточного узла этому указателю будет присвоен адрес хвостового узла, поскольку, как вы помните, именно хвостовой узел передал свой указатель this.

Теперь, после того как был создан узел InternalNode, адрес этого узла присваивается указателю dataNode в строке 141, и именно этот адрес возвращается теперь методом TailNode::Insert(). Так мы возвращаемся к методу HeadNode::Insert(), где адрес узла InternalNode присваивается указателю myNext узла HeadNode (строка 169). И наконец, адрес узла HeadNode возвращается в связанный список — туда, где в строке 197 он был сброшен (ничего страшного при этом не произошло, так как связанному списку уже был известен адрес головного узла).

Зачем было беспокоиться о возвращении адреса, если он не используется? Метод Insert был объявлен в базовом классе Node. Для выполнения метода необходимо задать значение возврата. Если изменить значение возврата функции HeadNode::Insert(), то компилятор покажет сообщение об ошибке. Это все равно что возвратить узел HeadNode и позволить связанному списку выбросить его адрес.

Так что же все-таки произошло? Данные были введены в список. Список передал эти данные головному узлу. Головной узел передал их дальше по тому адресу, что хранится в его указателе. В первом цикле в этом указателе хранился адрес хвостового узла. Хвостовой узел немедленно создает новый промежуточный узел, указателю которого присваивается адрес хвостового узла. Затем хвостовой узел возвращает адрес промежуточного узла в головной узел, где он присваивается указателю myNext головного узла. Итак, свершилось то, что требовалось: данные расположились в списке правильным образом (рис. 12.7).

После ввода первого промежуточного узла программа переходит к строке 211 и выводит предложение пользователю ввести новое значение. Предположим, что в этот раз было введено значение 3. В результате в строке 215 создается новый объект класса Data и вводится в список в строке 216.

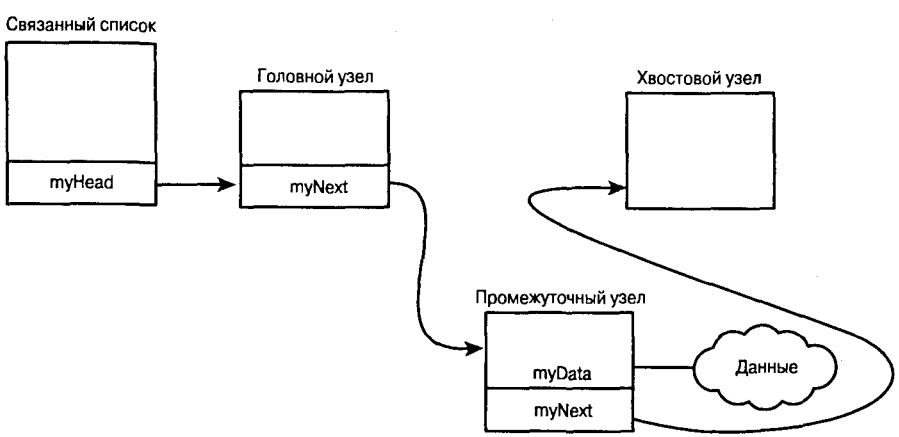


Рис. 12.7. Вид связанного списка после того, как был добавлен первый промежуточный узел

И вновь в строке 197 список передает новое значение в головной узел HeadNode. Метод HeadNode::Insert(), в свою очередь, передает эти данные по адресу, хранящемуся в указателе myNext. Как вы помните, теперь он указывает на узел, содержащий объект типа Data со значением 15. В результате в строке 96 вызывается метод InternalNode::Insert(). В строке

100 указатель myData узла InternalNode сообщает объекту этого узла (значение которого сейчас равно 15) о необходимости вызвать метод Compare(), принимающий в качестве аргумента новый объект Data со значением 3. Метод Compare() объявлен в строке 41.

Происходит сравнение значений двух объектов, и, поскольку значение myValue соответствует 15, а theOtherData.myValue равно 3, метод возвращает константу kIsLarger. В соответствии со значением возвращенной константы программа переходит к выполнению строки 109.

Создается новый узел InternalNode для нового объекта данных. Новый узел будет указывать на текущий объект узла InternalNode, и адрес нового узла InternalNode возвращается методом InternalNode::Insert() в головной узел HeadNode. Таким образом, новый узел, значение которого меньше значения текущего объекта, добавляется в связанный список, после чего связанный список выглядит так, как показано на рис. 12.8.

На третьем цикле пользователь ввел значение 8. Оно больше чем 3, но меньше чем 15, поэтому программа должна ввести новый объект данных между двумя существующими промежуточными узлами. Последует та же серия операций, что и в предыдущем цикле, за тем исключением, что при вызове метода Compare() для объекта типа Data, значение которого равно 3, будет возвращена константа kIsSmaller, а не kIsLarger, как в предыдущем случае (поскольку значение текущего объекта 3 меньше значения нового объекта 8).

В результате метод InternalNode::Insert() переведет выполнение программы на строку 116. Вместо создания и ввода в список нового узла, новые данные будут переданы в метод Insert того объекта, на который ссылается указатель myNext текущего объекта. В данном случае будет вызван метод InsertNode для промежуточного узла, значение объекта которого равняется 15.

Вновь будет проведено сравнение данных, которое в этот раз завершится созданием нового промежуточного узла. Этот новый узел будет ссылаться на тот промежуточный узел, значение которого 15, а адрес нового узла будет присвоен указателю узла со значением 3, как показано в строке 116.

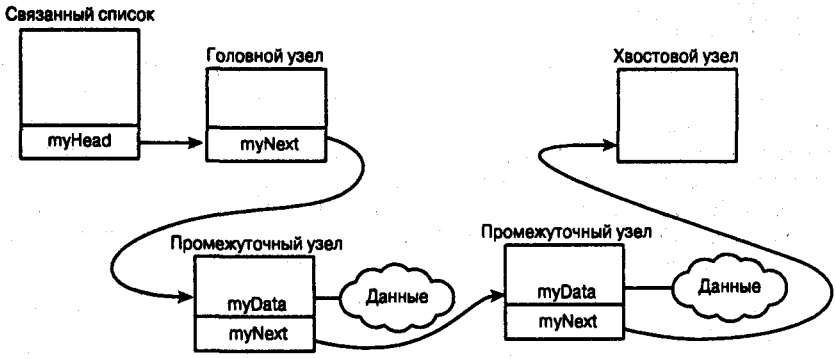


Рис. 12.8. Вид связанного списка после того, как был добавлен второй промежуточный узел

В результате новый узел вновь будет вставлен в правильную позицию.

Если вы переписали эту программу и запустили на своем компьютере, то с помощью средства отладки можно посмотреть, как будет происходить ввод других данных программой. Каждый раз будет проводиться сравнение данных и новые узлы будут добавляться в список строго в порядке возрастания значений.

Что мы узнали в этой главе

В программах, рассмотренных выше, не осталось ничего от привычных нам процедурных программ. При процедурном программировании контрольный метод сравнивает данные и вызывает функцию. При объектно-ориентированном программировании каждый отдельный объект служит для выполнения ограниченного набора четко определенных задач. Так, связанный список отвечает за поддержание головного узла. Головной узел немедленно передает данные по адресу своего указателя, не анализируя ни передаваемые данные, ни адресуемый объект.

Хвостовой узел создает новые узлы и добавляет их в список, если они содержат данные. Хвостовому узлу известно, что если новые объекты содержат какие-то данные, то они должны располагаться в списке до него.

Промежуточные узлы выполняют более сложные функции. Они обращаются к своим текущим объектам и сравнивают их значения со значениями новых объектов. В зависимости от результата сравнения, они либо вставляют объекты перед собой, либо передают их другому узлу.

Обратите внимание, что промежуточные узлы сами по себе не имеют никакого представления о данных объектов и о том, как их сравнивать. Сравнение выполняется методами, вызываемыми объектами. Все, за что отвечает промежуточный узел, — это обращение к своему объекту с требованием вызвать метод сравнения текущего значения с новым переданным значением. В зависимости от того, какую константу возвратит метод сравнения, узел либо добавляет объекты перед собой, либо передает их другому узлу, не беспокоясь о том, что будет с этим объектом дальше.

Кто же стоит над всем этим? В хорошо сконструированной объектно-ориентированной программе нет необходимости создавать какой-либо всеохватывающий объект контроля. Каждый объект выполняет свою маленькую партию, и результаты работы всех объектов сливаются в стройный хор.

Классы массивов

По сравнению с использованием встроенных массивов написание собственного класса массивов дает ряд преимуществ. Так, можно разработать систему контроля за вводом данных в массив для предупреждения ошибок или создать класс массива, динамически изменяющий размер. При создании массив может содержать только один элемент, постепенно прирастая по мере выполнения программы.

Можно разработать механизм сортировки или какого-либо другого упорядочения элементов массива либо использовать множество других эффективных вариантов массивов, наиболее популярны среди которых следующие:

- отсортированные коллекции: каждый член массива автоматически занимает свое определенное место;
- наборы: ни один из членов не повторяется в массиве дважды;
- словари: связанные пары элементов массива, где один член выполняет роль ключа для возвращения второго члена;
- разреженные массивы: допускается использование произвольных значений индексов, но в массив будут добавляться только реально существующие элементы. Так, можно ввести и использовать элемент с индексом `SprseArray[5]` или `SprseArray[200]`, но это не значит, что в массиве реально существуют все элементы с меньшими индексами;
- мультимножества: неупорядоченные коллекции, члены которых добавляются и возвращаются в произвольном порядке.

Перегрузив оператор индексирования ([]), можно превратить связанный список в отсортированную коллекцию. Если добавить функцию отслеживания одинаковых членов, то отсортированная коллекция превратится в набор. Если все объекты списка связать попарно, то связанный список превратится в словарь или в разреженный массив.

Резюме

Сегодня вы узнали, как создавать массивы в C++. Массив представляет собой коллекцию объектов одинакового типа с фиксированным числом элементов.

Массивы никак не контролируют свой размер. Поэтому вполне возможно в программе заносить данные за пределы массива, что часто является причиной ошибок. Отсчет индексов массива начинается с 0. Часто допускаемой ошибкой является указание индекса `n` для массива с размером `n`.

Массивы могут быть одномерными или многомерными. Независимо от размерности, все массивы базовых типов (например, `int`) или массивы объектов классов с конструкторами, заданными по умолчанию, могут быть инициализированы при объявлении.

Объекты массивов или все массивы целиком можно сохранять как в стековой области памяти, так и в области динамического обмена. Если удаляется объект из области динамической памяти, не забудьте установить квадратные скобки после ключевого слова `delete[]`.

Имена массивов представляют собой константные указатели на первый элемент массива. Чтобы получить доступ к другим элементам, имена массивов можно использовать в математических операциях, как при работе с обычными указателями.

Если размер коллекции объектов не известен во время компиляции программы, то для поддержания таких коллекций можно использовать связанные списки. Взяв связанный список за основу, можно разработать много других видов массивов и структур, автоматически выполняющих сложные операции.

Строки представляют собой массивы символов. В C++ существуют дополнительные средства манипулирования текстовыми строками, включая возможность ввода в массив строки, взятой в двойные кавычки.

Вопросы и ответы

Что произойдет, если в массив из 24-х членов вписать значение для 25-го элемента?

Значение будет добавлено в ячейку памяти, не принадлежащую массиву, что может вызвать серьезную ошибку в работе программы.

Что представляют собой элементы неинициализированного массива?

Ячейки памяти, отведенные массиву но не инициализированные, могут содержать любую информацию, ранее сохраненную в этих ячейках. Результат обращения в программе к элементу массива, который не был инициализирован, не предсказуем.

Можно ли создавать комбинации массивов?

Да. Массив может содержать указатель на другой, более крупный массив. В случае работы со строками можно использовать некоторые стандартные функции, такие как `strcat`, чтобы создавать комбинации массивов символов.

Чем связанные списки лучше массивов?

Массивы всегда имеют фиксированный размер, тогда как размер связанного списка может изменяться динамически во время выполнения программы.

Всегда ли нужно в классе строк использовать указатель `char *` для сохранения содержимого строки?

Нет. Можно использовать любую область памяти, которая больше подходит для решения конкретных задач.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Как обратиться к первому и последнему элементам массива `SomeArray[25]`?
2. Как объявить многомерный массив?
3. Выполните инициализацию элементов многомерного массива, созданного при ответе на вопрос 2.
4. Сколько элементов содержит массив `SomeArray[10][5][20]`?
5. Каково максимальное число элементов, которые можно добавить в связанный список?
6. Можно ли в связанном списке использовать индексы?
7. Каким является последний символ в строке "Сергей — хороший парень"?

Упражнения

1. Объявите двухмерный массив, который представляет поле для игры в крестики и нолики.
2. Запишите программный код, инициализирующий значением `0` все элементы созданного перед этим массива.
3. Объявите класс узла `Node`, поддерживающего целые числа.
4. **Жучки**: что неправильно в следующей программе?

```
unsigned short SomeArray[5][4];
for (int i = 0; i<4; i++)
   for (int j = 0; j<5; j++)
        SomeArray[i][j] = i+j;
```

5. **Жучки**: что неправильно в следующей программе?

```
unsigned short SomeArray[5][4];
for (int i = 0; i<=5; i++)
   for (int j = 0; j<=4; j++)
        SomeArray[i][j] = 0;
```

Полиморфизм

На прошлом занятии вы узнали, как создавать виртуальные функции в производных классах. На этом занятии речь пойдет об основном составляющем ядре полиморфизма — возможности во время выполнения программы связывать специфические объекты производных классов с указателями базового класса. Сегодня вы узнаете:

- Что такое множественное наследование и как его использовать
- Что представляет собой виртуальное наследование
- Что такое абстрактные типы данных
- Что такое чистые виртуальные функции

Проблемы с одиночным наследованием

Давайте продолжим работу над программой о животных и предположим, что в ней теперь используется два класса, произведенных от какого-то общего класса. Один — `Bird`, посвященный птицам, а другой — `Mammals`, посвященный млекопитающим. Класс `Bird` содержит функцию-член `Fly()`, задающую возможность полета. Класс `Mammals` разбит на ряд подклассов, включая класс лошадей — `Horse`. Класс содержит две функции-члена — `Whinny()` и `Gallop()`, объявляющих ржание и бег галопом соответственно.

Но внезапно у вас возникает желание создать новый весьма интересный мифический объект — крылатого Пегаса (`Pegasus`), который был бы чем-то вроде гибрида между `Horse` и `Bird`. Сразу предупредим, что, используя только одиночное наследование, вам сложно будет справиться с этой задачей.

Если объявить объект `Pegasus` как член класса `Bird`, то для него станут недоступными функции `Whinny()` и `Gallop()`. Если `Pegasus` объявить как объект класса `Horse`, то ему станет недоступной функция `Fly()`.

Первое решение может состоять в том, чтобы скопировать метод `Fly()` в класс `Horse`, после чего в этом классе создать объект `Pegasus`. При этом оба класса (`Bird` и `Horse`) будут содержать один и тот же метод `Fly()`, и при изменении метода в одном классе нужно будет не забыть внести соответствующие изменения в другом классе. Хорошо, если таких классов будет только два. Если вам придется вносить изменения в программу через некоторое время после ее создания, будет сложно вспомнить, в каких еще классах представлен этот метод.

Когда вы захотите создать списки объектов классов Bird и Horse, перед вами возникнет еще одна проблема. Хотелось бы, чтобы объект Pegasus был представлен в обоих списках, но в данном случае это невозможно.

Для решения возникшей проблемы можно использовать несколько подходов. Например, можно переименовать слишком "лошадиный" метод Gallop() в более обтекаемый Move(), после чего заместить этот метод в объекте Pegasus таким образом, чтобы он выполнял функцию метода Fly(). В других объектах класса Horse метод Move() будет выполняться так же, как раньше выполнялся метод Gallop(). Для объекта Pegasus можно даже определить, что короткие дистанции он должен преодолевать методом Gallop(), а длинные — методом Fly():

```
Pegasus::Move(long distance)
{
if (distance > veryFar)
fly(distance);
else
gallop(distance);
}
```

Но и этот подход имеет ряд ограничений, поскольку объект уже не сможет летать на короткие дистанции и бегать на длинные. Может быть, все же просто перенести метод Fly() в класс Horse, как показано в листинге 13.1? Проблема состоит в том, что лошади, в большинстве своем, летать не умеют, поэтому во всех объектах этого класса, за исключением объекта Pegasus, данный метод не должен ничего выполнять.

Листинг 13.1. Умеют ли лошади летать...

```
1:  // Листинг 13.1. Умеют ли лошади летать...
2:  // Фильтрация метода Fly() в классе Horse
3:
4:  #include <iostream.h>
5:
6:  class Horse
7:  {
8:  public:
9:   void Gallop(){ cout << "Galloping...\ n"; }
10:   virtual void Fly() { cout << "Horses can't fly.\ n" ; }
11: private:
12:   int itsAge;
13: } ;
14:
15: class Pegasus : public Horse
16: {
17: public:
18:   virtual void Fly() { cout << "I can fly! I can fly! I can
    ③fly!\ n"; }
19: } ;
20:
21: const int NumberHorses = 5;
22: int main()
23: {
24:   Horse* Ranch[NumberHorses];
```

```
25:    Horse* pHorse;
26:    int choice,i;
27:    for (i=0; i<NumberHorses; i++)
28:    {
29:     cout << "(1)Horse (2)Pegasus: ";
30:     cin >> choice;
31:     if (choice == 2)
32:       pHorse = new Pegasus;
33:     else
34:       pHorse = new Horse;
35:     Ranch[i] = pHorse;
36:    }
37:    cout << "\ n";
38:    for (i=0; i<NumberHorses; i++)
39:    {
40:     Ranch[i]->Fly();
41:     delete Ranch[i];
42:    }
43:   return 0;
44:  }
```

РЕЗУЛЬТАТ

```
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1

Horses can't fly.
I can fly! I can fly! I can fly!
Horses can't fly.
I can fly! I can fly! I can fly!
Horses can't fly.
```

АНАЛИЗ Безусловно, эта программа будет работать ценой добавления в класс `Horse` редко используемого метода `Fly()`. Это произошло в строке 10. Для объектов данного класса этот метод констатирует факт, что лошади летать не умеют. И только для объекта `Pegasus` метод замещается в строке 18 таким образом, что при вызове его объект заявляет, что умеет летать.

В строке 24 используется массив указателей на объекты класса `Horse`, с помощью которого метод `Fly()` вызывается для разных объектов класса. В зависимости от того, для какого из объектов в данный момент вызывается метод, программа выводит на экран разные сообщения.

ПРИМЕЧАНИЕ

Показанный выше пример программы был значительно сокращен, чтобы выделить именно те моменты, которые сейчас рассматриваются. Так, для простоты программы из нее были удалены конструктор и виртуальные деструкторы.

Перенос метода вверх по иерархии классов

Очень часто для решения подобных проблем объявление метода переносят вверх по иерархическому списку классов, чтобы сделать его доступным большему числу производных классов. Но при этом есть угроза, что базовый класс уподобится кладовке, захламленной старыми вещами. Такой подход делает программу громоздкой и нарушает саму идею иерархии классов в C++, когда производные классы дополняют своими функциями небольшой набор общих функций базового класса.

Противоречие состоит в том, что при переносе функции из производных классов вверх по иерархии в базовый класс трудно сохранить уникальность интерфейсов производных классов. Так, можно предположить, что у наших двух классов `Bird` и `Horse` есть базовый класс `Animal`, в котором собраны функции, общие для всех производных классов, например функция питания — `Eat()`. Перенеся метод `Fly()` в базовый класс, придется позаботиться о том, чтобы этот метод вызывался только в некоторых производных классах.

Приведение указателя к типу производного класса

Продолжая держаться за одиночное наследование, эту проблему можно решить таким образом, что метод `Fly()` будет вызываться только в случае, если указатель в данный момент связан с объектом `Pegasus`. Для этого необходимо иметь возможность обратиться к указателю и определить, на какой объект он указывает в текущий момент. Такой подход известен как RTTI (Runtime Type Identification — определение типа при выполнении). Но возможность выполнения RTTI была добавлена только в последние версии компиляторов C++.

Если ваш компилятор не поддерживает RTTI, можете реализовать его собственными силами, добавив в программу метод, который возвращает перечисление типов каждого класса. Возвращенное значение можно анализировать во время выполнения программы и допускать вызов метода `Fly()` только в том случае, если возвращается значение `Pegasus`.

> Не злоупотребляйте использованием RTTI в своих программах, так как этот подход рассматривается как аварийный и свидетельствует о том, что структура программы изначально была плохо продумана. Профессиональный программист предпочтет использование виртуальных функций, шаблонов или множественного наследования, речь о котором пойдет ниже в этой главе.

Чтобы вызвать метод `Fly()`, необходимо во время выполнения изменить тип указателя, определив, что он связан не с объектом `Horse`, а с объектом производного класса `Pegasus`. Этот способ называют *приведением вниз*, поскольку объект базового класса `Horse` приводится к объекту производного класса `Pegasus`.

Этот подход, хоть и с неохотой, теперь уже официально признан в C++, и для его реализации добавлен новый оператор — `dynamic_cast`.

Если в программе создан указатель на объекты базового класса `Horse` и ему присвоен адрес объекта производного класса `Pegasus`, то такой указатель можно использовать полиморфно. Чтобы обратиться к методу производного класса, нужно динамически подменить указатель базового класса указателем производного класса с помощью оператора `dynamic_cast`.

Во время выполнения программы происходит тестирование указателя базового класса. Если устанавливается, что текущий объект, на который ссылается указатель базового класса, в действительности является объектом производного класса, то с этим объектом связывается указатель производного класса. В противном случае указатель производного класса становится нулевым. Пример использования этого подхода показан в листинге 13.2.

Листинг 13.2. Приведение вниз

```
1:  // Листинг 13 2  Использование оператора dynamic_cast
2:  // Использование rtti
3
4:  #include <iostream h>
5:  enum TYPE { HORSE, PEGASUS } ,
6:
7:  class Horse
8:  {
9:  public:
10:     virtual void Gallop(){   cout << "Galloping...\n"; }
11:
12:  private:
13:     int itsAge;
14:  } ,
15.
16:  class Pegasus : public Horse
17:  {
18:  public:
19:
20:     virtual void Fly() {   cout << "I can fly! I can fly! I can fly!\ n"; }
21:  } ;
22:
23.  const int NumberHorse = 5,
24:  int main()
25:  {
26:     Horse* Ranch[NumberHorse];
27:     Horse* pHorse;
28:     int choice,i,
29:     for (i=0; i<NumberHorse, i++)
30:     {
31:    cout << "(1)Horse (2)Pegasus: ";
32:    cin >> choice;
33.    if (choice == 2)
34:      pHorse = new Pegasus;
35:    else
36:      pHorse = new Horse;
37:    Ranch[i] = pHorse,
38:   }
39:   cout << "\ n";
40:   for (i=0; i<NumberHorses; i++)
41:   {
42.    Pegasus *pPeg = dynamic_cast< Pegasus *> (Ranch[i]);
```

```
42:     if (pPeg)
43:       pPeg->Fly();
44:     else
45:       cout << "Just a horse\ n";
46:
47:     delete Ranch[i];
48:    }
49:   return 0;
50: }
```

РЕЗУЛЬТАТ

```
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1
(1)Horse (2)Pegasus: 2
(1)Horse (2)Pegasus: 1

Just a horse
I can fly! I can fly! I can fly!
Just a horse
I can fly! I can fly! I can fly!
Just a horse
```

Вопросы и ответы

Во время компиляции появляется сообщение об ошибке C4541: 'dynamic_cast' used on polymorphic type 'class Horse' with/GR-; unpredictable behavior may result. Как поступить?

Это сообщение MFC действительно может смутить начинающего программиста. Чтобы устранить ошибку, выполните ряд действий.

1. Выберите в окне проекта команду Project⇨Settings.
2. Перейдите к вкладке C/C++.
3. Выберите в раскрывающемся списке Category опцию C++ Language.
4. Установите Enable Runtime Type Information (RTTI).
5. Повторно скомпилируйте весь проект.

АНАЛИЗ Этот пример программы также будет вполне работоспособным. Метод `Fly()` не связан напрямую с классом `Horse` и не будет вызываться для обычных объектов этого класса. Он выполняется только для объектов класса `Pegasus`, но для этого программе приходится каждый раз анализировать, с каким объектом связан указатель, и приводить текущий указатель к типу производного класса.

Все же следует признать, что программы с приведением типа объектов выглядят несколько неуклюже и в них легко запутаться. Кроме того, данный подход идет в разрез с основной идеей полиморфизма виртуальных функций, поскольку выполняемость метода теперь зависит от приведения типа объекта во время выполнения программы.

Добавление объекта в два списка

Другая проблема состоит в том, что при объявлении Pegasus как объекта типа Horse становится невозможным добавить его в список объектов класса Bird. Приходилось то переносить функцию Fly() вверх по иерархии классов, то выполнять приведение указателя, но так и не удалось в полной мере достичь необходимого функционирования программы.

Таким образом, придерживаясь только одиночного наследования, оказалось невозможным элегантно решить эту проблему. Можно перенести все три функции — Fly(), Whinny() и Gallop() — в базовый класс Animal, общий для двух производных классов Bird и Horse. В результате вместо двух списков объектов для классов Bird и Horse получится один общий список объектов класса Animal. Недостаток метода состоит в том, что базовый класс принимает на себя слишком много функций.

В качестве альтернативы можно оставить методы там, где они есть, и заняться приведением типов объектов классов Horse, Bird и Pegasus, но результат в конечном итоге будет еще хуже!

Рекомендуется	Не рекомендуется
Переносите вверх по иерархии классов функции общего использования. **Избегайте** использовать коды, основанные на определении типов объектов во время выполнения программы. Вместо этого используйте виртуальные методы, шаблоны и множественное наследование.	**Не переносите** вверх по иерархии классов интерфейсы производных классов.

Множественное наследование

Существует возможность производить новые классы более чем от одного базового класса. Такой процесс называется *множественным наследованием*. Чтобы произвести подобный класс, базовые классы в объявлении разделяются запятыми. В листинге 13.3 класс Pegasus объявлен таким образом, что наследует свойства двух базовых классов — Bird и Horse. Затем программа добавляет объект Pegasus в списки объектов обоих классов.

Листинг 13.3. Множественное наследование

```
1:   // Листинг 13.3. Множественное наследование.
2:   // Множественное наследование
3:
4:   #include <iostream.h>
5:
6:   class Horse
7:   {
8:   public:
9:    Horse() { cout << "Horse constructor... "; }
10:   virtual ~Horse() { cout << "Horse destructor... "; }
11:   virtual void Whinny() const { cout << "Whinny!... "; }
```

```
12:  private:
13:    int itsAge;
14:  } ;
15:
16:  class Bird
17:  {
18:  public:
19:    Bird() { cout << "Bird constructor... "; }
20:    virtual ~Bird() { cout << "Bird destructor... "; }
21:    virtual void Chirp() const { cout << "Chirp... "; }
22:    virtual void Fly() const
23:    {
24:     cout << "I can fly! I can fly! I can fly! ";
25:    }
26:  private:
27:    int itsWeight;
28:  } ;
29:
30:  class Pegasus : public Horse, public Bird
31:  {
32:  public:
33:    void Chirp() const { Whinny(); }
34:    Pegasus() { cout << "Pegasus constructor... "; }
35:    ~Pegasus() { cout << "Pegasus destructor... "; }
36:  } ;
37:
38:  const int MagicNumber = 2;
39:  int main()
40:  {
41:    Horse* Ranch[MagicNumber];
42:    Bird* Aviary[MagicNumber];
43:    Horse * pHorse;
44:    Bird * pBird;
45:    int choice,i;
46:    for (i=0; i<MagicNumber; i++)
47:    {
48:     cout << "\ n(1)Horse (2)Pegasus: ";
49:     cin >> choice;
50:     if (choice == 2)
51:       pHorse = new Pegasus;
52:     else
53:       pHorse = new Horse;
54:     Ranch[i] = pHorse;
55:    }
56:    for (i=0; i<MagicNumber; i++)
57:    {
58:     cout << "\ n(1)Bird (2)Pegasus: ";
59:     cin >> choice;
60:     if (choice == 2)
61:       pBird = new Pegasus;
```

```
62:     else
63:       pBird = new Bird;
64:     Aviary[i] = pBird;
65:    }
66:
67:    cout << "\ n";
68:    for (i=0; i<MagicNumber; i++)
69:    {
70:     cout << "\ nRanch[" << i << "]: " ;
71:     Ranch[i]->Whinny();
72:     delete Ranch[i];
73:    }
74:
75:    for (i=0; i<MagicNumber; i++)
76:    {
77:     cout << "\ nAviary[" << i << "]: " ;
78:     Aviary[i]->Chirp();
79:     Aviary[i]->Fly();
80:     delete Aviary[i];
81:    }
82:   return 0;
83:  }
```

РЕЗУЛЬТАТ

```
(1)Horse (2)Pegasus: 1
Horse constructor...
(1)Horse (2)Pegasus: 2
Horse constructor... Bird constructor... Pegasus constructor...
(1)Bird (2)Pegasus: 1
Bird constructor...
(1)Bird (2)Pegasus: 2
Horse constructor... Bird constructor... Pegasus constructor...

Ranch[0]: Whinny!... Horse destructor...
Ranch[1]: Whinny!... Pegasus destructor... Bird destructor...
Horse destructor...
Aviary[0]: Chirp... I can fly! I can fly! I can fly! Bird destructor...
Aviary[1]: Whinny!... I can fly! I can fly! I can fly!
Pegasus destructor... Bird destructor... Horse destructor...
```

АНАЛИЗ В строках 6—14 объявляется класс `Horse`. Конструктор и деструктор выводят на экран сообщения о своей работе, а метод `Whinny()` печатает `Whinny!` (И-го-го).

Класс `Bird` объявляется в строках 16—28. В дополнение к своим конструктору и деструктору этот класс содержит два метода: `Chirp()` и `Fly()`, каждый из которых выводит на экран соответствующие сообщения. В реальных программах эти методы могут воспроизводить определенный звуковой файл или управлять анимационными эффектами на экране.

Наконец, в строках 30–36 объявляется класс Pegasus. Он производится сразу от двух базовых классов — Bird и Horse. В классе замещается метод Chirp() таким образом, что вызывается метод Whinny(), который унаследован этим классом от класса Horse.

Создается два списка: Ranch (конюшня), который в строке 41 связывается с классом Horse, и Aviary (птичник), который в строке 42 связывается с классом Bird. В строках 46–55 в список Ranch добавляются два объекта — Horse и Pegasus. В строках 56–65 в список Aviary добавляются объекты Bird и Pegasus.

Вызовы виртуальных методов с помощью указателей классов Bird и Horse одинаково выполняются для объекта Pegasus. Например, в строке 78 метод Chirp() вызывается последовательно для всех объектов, указатели на которые представлены в массиве Aviary. Поскольку этот метод объявлен в классе Bird как виртуальный, он правильно выполняется для всех объектов списка.

По выводимым на экран строкам можно заключить, что при создании объекта Pegasus вызываются конструкторы всех трех классов — Bird, Horse и Pegasus, каждый из которых создает свою часть объекта. При удалении объекта также удаляются его части, относящиеся к классам Bird и Horse, для чего деструкторы в этих классах объявлены как виртуальные.

Объявление множественного наследования

Чтобы указать, что создаваемый объект наследует свойства более чем одного базового класса, после имени создаваемого класса ставится двоеточие, вслед за которым через запятую перечислены имена всех базовых классов.

Пример 1:

```
class Pegasus : public Horse, public Bird
```

Пример 2:

```
class Schnoodle : public Schnauzer, public Poodle
```

Из каких частей состоят объекты, полученные в результате множественного наследования

Когда в памяти компьютера создается объект Pegasus, конструкторы обоих классов принимают участие в его построении, как показано на рис. 13.1.

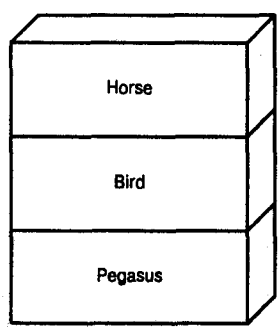


Рис. 13.1. Объект, полученный в результате множественного наследования

В случае использования множественного наследования возникает ряд непростых и весьма интересных вопросов. Например, что произойдет, если оба базовых класса будут иметь одно и то же имя либо содержать виртуальные функции или данные с одинаковыми именами? Как инициализируются конструкторы разных базовых классов? Что произойдет, если два базовых класса будут произведены от одного и того же родительского класса? Все эти вопросы будут рассмотрены в следующем разделе, после чего можно переходить к практическому использованию множественного наследования.

Конструкторы классов, полученных в результате множественного наследования

Если класс `Pegasus` производится от двух базовых классов — `Bird` и `Horse`, а в каждом из них объявлены конструкторы со списками параметров, то класс `Pegasus` инициализирует эти конструкторы. Как это происходит, показано в листинге 13.4.

Листинг 13.4. Создание объектов при множественном наследовании

```
1:  // Листинг 13.4.
2:  // Создание объектов при множественном наследовании
3:  #include <iostream.h>
4:  typedef int HANDS;
5:  enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown };
6:
7:  class Horse
8:  {
9:  public:
10:     Horse(COLOR color, HANDS height);
11:   virtual ~Horse() { cout << "Horse destructor...\ n"; }
12:   virtual void Whinny()const { cout << "Whinny!... "; }
13:   virtual HANDS GetHeight() const { return itsHeight; }
14:   virtual COLOR GetColor() const { return itsColor; }
15: private:
16:   HANDS itsHeight;
17:   COLOR itsColor;
18: } ;
19:
20: Horse::Horse(COLOR color, HANDS height):
21:   itsColor(color),itsHeight(height)
22: {
23:   cout << "Horse constructor...\ n";
24: }
25:
26: class Bird
27: {
28: public:
29:   Bird(COLOR color, bool migrates);
30:   virtual ~Bird() { cout << "Bird destructor...\ n"; }
31:   virtual void Chirp()const { cout << "Chirp... "; }
32:   virtual void Fly()const
33:   {
```

```
34:     cout << "I can fly! I can fly! I can fly! ";
35:   }
36:   virtual COLOR GetColor()const { return itsColor; }
37:   virtual bool GetMigration() const { return itsMigration; }
38:
39: private:
40:   COLOR itsColor;
41:   bool itsMigration;
42: } ;
43:
44: Bird::Bird(COLOR color, bool migrates):
45:   itsColor(color), itsMigration(migrates)
46: {
47:   cout << "Bird constructor...\ n";
48: }
49:
50: class Pegasus : public Horse, public Bird
51: {
52: public:
53:   void Chirp()const { Whinny(); }
54:   Pegasus(COLOR, HANDS, bool,long);
55:   ~Pegasus() { cout << "Pegasus destructor...\ n";}
56:   virtual long GetNumberBelievers() const
57:   {
58:     return itsNumberBelievers;
59:   }
60:
61: private:
62:   long itsNumberBelievers;
63: } ;
64:
65: Pegasus::Pegasus(
66:   COLOR aColor,
67:   HANDS height,
68:   bool migrates,
69:   long NumBelieve):
70: Horse(aColor, height),
71: Bird(aColor, migrates),
72: itsNumberBelievers(NumBelieve)
73: {
74:   cout << "Pegasus constructor...\ n";
75: }
76:
77: int main()
78: {
79:   Pegasus *pPeg = new Pegasus(Red, 5, true, 10);
80:   pPeg->Fly();
81:   pPeg->Whinny();
82:   cout << "\ nYour Pegasus is " << pPeg->GetHeight();
83:   cout << " hands tall and ";
```

```
84:    if (pPeg->GetMigration())
85:     cout << "it does migrate.";
86:    else
87:     cout << "it does not migrate.";
88:    cout << "\ nA total of " << pPeg->GetNumberBelievers();
89:    cout << " people believe it exists.\ n";
90:    delete pPeg;
91:   return 0;
92:  }
```

РЕЗУЛЬТАТ

```
Horse constructor...
Bird constructor...
Pegasus constructor...
I can fly! I can fly! I can fly! Whinny!...
Your Pegasus is 5 hands tall and it does migrate.
A total of 10 people believe it exists.

Pegasus destructor...
Bird destructor...
Horse destructor...
```

Класс `Horse` объявляется в строках 7–18. Конструктор этого класса принимает два параметра: один из них — это перечисление, объявленное в строке 5, а второй — новый тип, объявленный с помощью `typedef` в строке 4. Этот конструктор выполняется в строках 20–24. При этом инициализируется одна переменная-член и на экран выводится сообщение о работе конструктора класса `Horse`.

В строках 26–42 объявляется класс `Bird`, конструктор которого выполняется в строках 45–49. Конструктор этого класса также принимает два параметра. Обратите внимание на интересный факт: конструкторы обоих классов принимают перечисления цветов, с помощью которых в программе можно установить цвет лошади или цвет перьев у птицы. В результате, когда вы попытаетесь установить цвет Пегаса, может возникнуть проблема в работе программы, которая обсуждается несколько ниже.

Класс `Pegasus` объявляется в строках 50–63, а его конструктор — в строках 65–75. Инициализация объекта `Pegasus` выполняется тремя строками программы. Сначала конструктор класса `Horse` определяет цвет и рост. Затем конструктор класса `Bird` инициализируется цветом перьев и логической переменной. Наконец, происходит инициализация переменной-члена `itsNumberBelievers`, относящейся к классу `Pegasus`. После всех этих операций вызывается конструктор класса `Pegasus`.

В функции `main()` создается указатель на класс `Pegasus`, который используется для получения доступа к функциям-членам базовых объектов.

Двусмысленность ситуации

В листинге 13.4 оба класса — `Horse` и `Bird` — имеют метод `GetColor()`. В программе может потребоваться возвратить цвет объекта `Pegasus`, но возникает вопрос: какой из двух унаследованных методов при этом будет использоваться? Ведь методы, объявленные в обоих базовых классах, имеют одинаковые имена и сигнатуры. В результате при компилировании программы возникнет неопределенность, которую необходимо разрешить до компиляции.

Если просто записать:

```
COLOR currentColor = pPeg->GetColor();
```

компилятор покажет сообщение об ошибке `Member is ambiguous: 'Horse::GetColor' and 'Bird::GetColor'` (Член не определен).

Эту неопределенность можно разрешить, явно обратившись к методу того класса, который вам необходим:

```
COLOR currentColor = pPeg->Horse::GetColor();
```

В любом случае при возникновении подобной ситуации, когда требуется сделать выбор между одноименными методами или переменными-членами разных классов, следует явно указывать имя необходимого базового класса перед именем функции-члена или переменной.

Если в классе `Pegasus` эта функция будет замещена, то проблема решится сама собой, так как в этом случае вызывается функция-член класса `Pegasus`:

```
virtual COLOR GetColor()const { return Horse::GetColor(); }
```

Таким образом, проблему неопределенности можно обойти благодаря инкапсуляции явного указания базового класса в объявлении замещенной функции. Если возникнет необходимость использовать метод другого класса, то обращение к нему с помощью приведенного ниже выражения не будет ошибкой.

```
COLOR currentColor = pPeg->Bird::GetColor();
```

Наследование от общего базового класса

Что произойдет, если оба базовых класса, от которых производится другой класс, сами были произведены от одного общего базового класса, как, например, классы `Bird` и `Horse` от класса `Animal`. Эта ситуация показана на рис. 13.2.

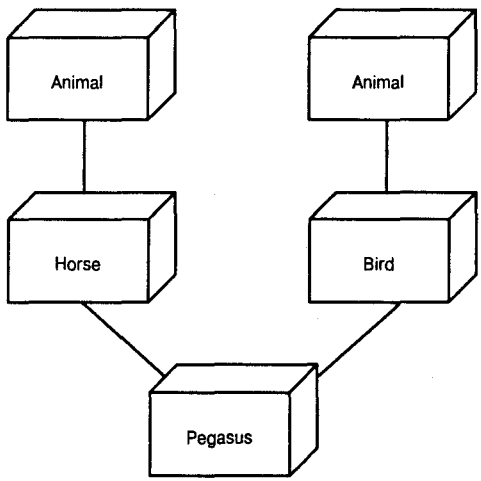


Рис. 13.2. Общий базовый класс

Как показано на рис. 13.2, два класса, являющихся базовыми для класса `Pegasus`, сами производятся от одного общего класса `Animal`. Компилятор при этом рассматривает классы `Bird` и `Horse` как производные от двух одноименных базовых классов, что

может привести к очередной неопределенности. Например, если в классе Animal объявлены переменная-член itsAge и функция-член GetAge(), а в программе делается вызов pGet->GetAge(), то будет ли при этом вызываться функция GetAge(), унаследованная классом Bird от класса Animal или классом Horse от базового класса? Это противоречие разрешается в листинге 13.5.

Листинг 13.5. Общий базовый класс

```
1:  // Листинг 13.5.
2:  // Общий базовый класс
3:  #include <iostream.h>
4:
5:  typedef int HANDS;
6:  enum COLOR {   Red, Green, Blue, Yellow, White, Black, Brown }   ;
7:
8:  class Animal     // общий базовый класс для классов horse и bird
9:  {
10:  public:
11:     Animal(int);
12:   virtual ~Animal() { cout << "Animal destructor...\ n"; }
13:   virtual int GetAge() const { return itsAge; }
14:   virtual void SetAge(int age) { itsAge = age; }
15: private:
16:   int itsAge;
17: } ;
18:
19: Animal::Animal(int age):
20: itsAge(age)
21: {
22:   cout << "Animal constructor...\ n";
23: }
24:
25: class Horse : public Animal
26: {
27: public:
28:   Horse(COLOR color, HANDS height, int age);
29:   virtual ~Horse() { cout << "Horse destructor...\ n"; }
30:   virtual void Whinny()const { cout << "Whinny!... "; }
31:   virtual HANDS GetHeight() const { return itsHeight; }
32:   virtual COLOR GetColor() const { return itsColor; }
33: protected:
34:   HANDS itsHeight;
35:   COLOR itsColor;
36: } ;
37:
38: Horse::Horse(COLOR color, HANDS height, int age):
39:   Animal(age),
40:   itsColor(color),itsHeight(height)
41: {
42:   cout << "Horse constructor...\ n";
```

```
43: }
44:
45: class Bird : public Animal
46: {
47: public:
48:   Bird(COLOR color, bool migrates, int age);
49:   virtual ~Bird() { cout << "Bird destructor...\n"; }
50:   virtual void Chirp()const { cout << "Chirp... "; }
51:   virtual void Fly()const
52:     { cout << "I can fly! I can fly! I can fly! "; }
53:   virtual COLOR GetColor()const { return itsColor; }
54:   virtual bool GetMigration() const { return itsMigration; }
55: protected:
56:   COLOR itsColor;
57:   bool itsMigration;
58: };
59:
60: Bird::Bird(COLOR color, bool migrates, int age):
61:   Animal(age),
62:   itsColor(color), itsMigration(migrates)
63: {
64:   cout << "Bird constructor...\n";
65: }
66:
67: class Pegasus : public Horse, public Bird
68: {
69: public:
70:   void Chirp()const { Whinny(); }
71:   Pegasus(COLOR, HANDS, bool, long, int);
72:   virtual ~Pegasus() { cout << "Pegasus destructor...\n";}
73:   virtual long GetNumberBelievers() const
74:    { return itsNumberBelievers; }
75:   virtual COLOR GetColor()const { return Horse::itsColor; }
76:   virtual int GetAge() const { return Horse::GetAge(); }
77: private:
78:   long itsNumberBelievers;
79: };
80:
81: Pegasus::Pegasus(
82:   COLOR aColor,
83:   HANDS height,
84:   bool migrates,
85:   long NumBelieve,
86:   int age):
87: Horse(aColor, height,age),
88: Bird(aColor, migrates,age),
89: itsNumberBelievers(NumBelieve)
90: {
91:   cout << "Pegasus constructor...\n";
92: }
```

```
93:
94:  int main()
95:  {
96:    Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
97:    int age = pPeg->GetAge();
98:    cout << "This pegasus is " << age << " years old.\ n";
99:   delete pPeg;
100:   return 0;
101:  }
```

РЕЗУЛЬТАТ

```
Animal constructor...
Horse constructor...
Animal constructor...
Bird constructor...
Pegasus constructor...
This pegasus is 2 years old.
Pegasus destructor...
Bird destructor...
Animal destructor...
Horse destructor...
Animal destructor...
```

АНАЛИЗ В листинге содержится ряд интересных решений. Так, в строках 8–17 объявляется новый класс `Animal` с переменной-членом `itsAge` и двумя методами — `GetAge()` и `SetAge()`.

В строке 25 класс `Horse` производится от класса `Animal`. Конструктор класса `Horse` теперь имеет третий параметр `age`, который передается в базовый класс `Animal`. Обратите внимание, что в классе `Horse` метод `GetAge()` не замещается, а просто наследуется.

В строке 46 класс `Bird` производится от класса `Animal`. Конструктор этого класса также содержит параметр `age`, с помощью которого инициализируется базовый класс `Animal`. Метод `GetAge()` также наследуется этим классом без замещения.

Класс `Pegasus` производится от двух базовых классов `Horse` и `Bird`, поэтому с исходным базовым классом `Animal` он связан двумя линиями наследования. Если для объекта класса `Animal` будет вызван метод `GetAge()`, то для преодоления неопределенности нужно точно указать, к какому базовому классу следует обращаться за этим методом, либо метод `GetAge()` следует заместить в классе `Pegasus`.

В нашем примере программы метод `GetAge()` замещается для класса `Pegasus` таким образом, что в нем явно указывается обращение к аналогичному методу конкретного базового класса.

Замещение функции с добавлением обращения к методу базового класса позволяет решить две проблемы. Во-первых, преодолевается неопределенность обращения к базовым классам; во-вторых, функцию можно заместить таким образом, что в производном классе при обращении к этой функции будут выполняться дополнительные операции, которых не было в базовом классе. Причем по желанию программиста эти дополнительные операции могут выполняться до вызова функции базового класса или после вызова с использованием значения, возвращенного функцией базового класса.

Конструктор класса `Pegasus` принимает пять параметров: цвет крылатого коня, его рост (в футах); логическую переменную, которая определяет, мигрирует сейчас это животное или мирно пасется на пастбище; число людей, верящих в существование Пегаса,

и возраст животного. В строке 87 конструктор инициализирует переменные, определенные в классе Horse (цвет, рост и возраст). В следующей строке инициализируется часть, относящаяся к классу Bird: цвет, миграции и возраст. Наконец, в строке 89 инициализируется переменная itsNumberBelievers, относящаяся непосредственно к классу Pegasus.

Вызов конструктора класса Horse в строке 87 выполняет операторы, записанные в строке 38. С помощью параметра age конструктор класса Horse инициализирует переменную itsAge, унаследованную классом Horse от класса Animal. Затем инициализируются две переменные-члена класса Horse — itsColor и itsHeight.

Вызов конструктора класса Bird в строке 88 выполняет операторы, записанные в строке 60. И в данном случае параметр age используется для инициализации переменной-члена, унаследованной классом Bird от класса Animal.

Обратите внимание, что значение параметра цвета объекта Pegasus используется для инициализации соответствующих переменных-членов обоих классов, Bird и Horse. Параметр age также инициализирует переменную itsAge обоих этих классов, унаследованную ими от базового класса Animal.

Виртуальное наследование

В листинге 13.5 решалась проблема неопределенности, а именно: от какого базового класса унаследована функция getAge() в объекте класса Pegasus. Но в действительности этот метод производится от одного общего базового класса Animal.

В C++ существует возможность указать, что мы имеем дело не с двумя одноименными классами, как показано в рис. 13.2, а с одним общим базовым классом (рис. 13.3).

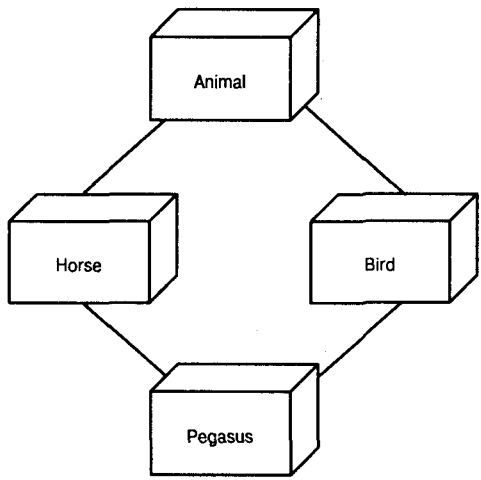


Рис. 13.3. Виртуальное наследование

Для этого класс Animal нужно объявить как виртуальный базовый класс для двух производных классов, Horse и Bird. Класс Animal при этом не подвергается никаким изменениям. В классах Horse и Bird изменения состоят в том, что в их объявлении указывается виртуальность наследования от базового класса Animal. Класс Pegasus изменяется существенно.

Обычно конструктор класса инициализирует только собственные переменные и переменные-члены базового класса. Из этого правила делается исключение, если используется виртуальное наследование. Переменные основного базового класса ини-

циализируются конструкторами не следующих производных от него классов, а тех, которые являются последними в иерархии классов. Поэтому класс Animal инициализируется не конструкторами классов Horse и Bird, а конструктором класса Pegasus. Конструкторы классов Horse и Bird также содержат команды инициализации базового класса Animal, но при создании объекта Pegasus эта инициализация перекрывается конструктором данного класса.

Листинг 13.6 представляет собой программный код из листинга 13.5, переписанный таким образом, чтобы можно было воспользоваться преимуществами виртуального наследования.

Листинг. 13.6. Пример использования виртуального наследования

```
1:  // Листинг 13.6.
2:  // Виртуальное наследование
3:  #include <iostream.h>
4:
5:  typedef int HANDS;
6:  enum COLOR {   Red, Green, Blue, Yellow, White, Black, Brown }   ;
7:
8:  class Animal      // общий базовый класс для двух производных классов horse и bird
9:  {
10:  public:
11:     Animal(int);
12:   virtual ~Animal() { cout << "Animal destructor...\ n"; }
13:   virtual int GetAge() const { return itsAge; }
14:   virtual void SetAge(int age) { itsAge = age; }
15: private:
16:   int itsAge;
17: } ;
18:
19: Animal::Animal(int age):
20: itsAge(age)
21: {
22:   cout << "Animal constructor...\ n";
23: }
24:
25: class Horse : virtual public Animal
26: {
27: public:
28:   Horse(COLOR color, HANDS height, int age);
29:   virtual ~Horse() { cout << "Horse destructor...\ n"; }
30:   virtual void Whinny()const { cout << "Whinny!... "; }
31:   virtual HANDS GetHeight() const { return itsHeight; }
32:   virtual COLOR GetColor() const { return itsColor; }
33: protected:
34:   HANDS itsHeight;
35:   COLOR itsColor;
36: } ;
37:
38: Horse::Horse(COLOR color, HANDS height, int age):
39:   Animal(age),
```

```
40:    itsColor(color),itsHeight(height)
41:  {
42:    cout << "Horse constructor...\ n";
43:  }
44:
45:  class Bird : virtual public Animal
46:  {
47:  public:
48:    Bird(COLOR color, bool migrates, int age);
49:    virtual ~Bird() { cout << "Bird destructor...\ n"; }
50:    virtual void Chirp()const { cout << "Chirp... "; }
51:    virtual void Fly()const
52:     { cout << "I can fly! I can fly! I can fly! "; }
53:    virtual COLOR GetColor()const { return itsColor; }
54:    virtual bool GetMigration() const { return itsMigration; }
55:  protected:
56:    COLOR itsColor;
57:    bool itsMigration;
58:  } ;
59:
60:  Bird::Bird(COLOR color, bool migrates, int age):
61:    Animal(age),
62:    itsColor(color), itsMigration(migrates)
63:  {
64:    cout << "Bird constructor...\ n";
65:  }
66:
67:  class Pegasus : public Horse, public Bird
68:  {
69:  public:
70:    void Chirp()const { Whinny(); }
71:    Pegasus(COLOR, HANDS, bool, long, int);
72:    virtual ~Pegasus() { cout << "Pegasus destructor...\ n";}
73:    virtual long GetNumberBelievers() const
74:     { return itsNumberBelievers; }
75:    virtual COLOR GetColor()const { return Horse::itsColor; }
76:  private:
77:    long itsNumberBelievers;
78:  } ;
79:
80:  Pegasus::Pegasus(
81:    COLOR aColor,
82:    HANDS height,
83:    bool migrates,
84:    long NumBelieve,
85:    int age):
86:  Horse(aColor, height,age),
87:  Bird(aColor, migrates,age),
88:  Animal(age*2),
89:  itsNumberBelievers(NumBelieve)
```

```
90:  {
91:    cout << "Pegasus constructor...\ n";
92:  }
93:
94:  int main()
95:  {
96:    Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
97:    int age = pPeg->GetAge();
98:    cout << "This pegasus is " << age << " years old.\ n";
99:   delete pPeg;
100:   return 0;
101:  }
```

РЕЗУЛЬТАТ

```
Animal constructor...
Horse constructor...
Bird constructor...
Pegasus constructor...
This pegasus is 4 years old.
Pegasus destructor...
Bird destructor...
Horse destructor...
Animal destructor...
```

АНАЛИЗ

В строке 25 класс `Horse` виртуально наследуется от класса `Animal`, а в строке 45 так же наследуется класс `Bird`. Обратите внимание, что конструкторы обоих классов по-прежнему инициализируют класс `Animal`. Но как только создается объект `Pegasus`, конструктор этого класса заново инициализирует класс `Animal`, отменяя прежние инициализации. Убедиться в этом вы можете по результату, выводимому программой на экран. При первой инициализации переменной `itsAge` присваивается значение `2`, но конструктор класса `Pegasus` удваивает это значение. В результате строка 98 программы выводит на экран значение `4`.

Проблемы с неопределенностью наследования метода в классе `Pegasus` больше не возникает, поскольку теперь метод `GetAge()` наследуется непосредственно из класса `Animal`. В то же время при обращении к методу `GetColor()` по-прежнему необходимо явно указывать базовый класс, так как этот метод объявлен в обоих классах, `Horse` и `Bird`.

Проблемы с множественным наследованием

Хотя множественное наследование дает ряд преимуществ по сравнение с одиночным, многие программисты с неохотой используют его. Основная проблема состоит в том, что многие компиляторы C++ все еще не поддерживают множественное наследование; это осложняет отладку программы, тем более что все возможности, реализуемые этим методом, можно получить и без него.

Действительно, если вы решите использовать в своей программе множественное наследование, следует учесть, что с отладкой программы могут возникнуть проблемы и чрезмерное усложнение программы, связанное с использованием этого подхода, не всегда оправдывается полученным эффектом.

Указание виртуального наследования при объявлении класса

Чтобы быть уверенным, что производные классы будут рассматривать исходный базовый класс как единый источник, виртуальность наследования следует указать во всех промежуточных классах.

Пример 1:

```
class Horse : virtual public Animal
class Bird : virtual public Animal
class Pegasus : public Horse, public Bird
```

Пример 2:

```
class Schnauzer : virtual public Dog
class Poodle : virtual public Dog
class Schnoodle : public Schnauzer, public Poodle
```

Рекомендуется	Не рекомендуется
Используйте множественное наследование в тех случаях, когда в классе необходимо применять данные и методы, объявленные в разных классах. **Используйте** виртуальное наследование, чтобы как можно элегантнее обойти проблемы с неопределенностью источника наследования метода или данных. **Инициализируйте** исходный базовый класс конструктором класса, наиболее удаленного от базового по иерархии классов.	**Не используйте** множественное наследование в тех случаях, когда можно обойтись одиночным наследованием.

Классы-мандаты

Промежуточным решением между одиночным и множественным наследованием классов может быть использование *классов-мандатов*. Так, класс `Horse` можно произвести от двух базовых классов — `Animal` и `Displayable`, причем последний добавляет только некоторые методы отображения объектов на экране.

Классом-мандатом называется класс, открывающий доступ к ряду методов, но не содержащий никаких данных (или, по крайней мере, содержащий минимальный набор данных).

Методы класса-мандата передаются в производные классы с помощью обычного наследования. Единственное отличие классов-мандатов от других классов состоит в том, что они практически не содержат никаких данных. Различие довольно субъективное и отражает только общую тенденцию программирования, сводящуюся к тому, что добавление функциональности классам не должно сопровождаться усложнением программы. Использование классов-мандатов также снижает вероятность возникновения неопределенностей при использовании в производном классе данных, унаследованных из других базовых классов.

Например, предположим, что класс Horse производится от двух классов — Animal и Displayable, причем последний добавляет только новые методы, но не содержит данных. В таком случае все наследуемые данные класса Horse происходят только от одного базового класса Animal, а методы наследуются от обоих классов.

Классы-мандаты (capability class) иногда еще называют *миксинами* (mixin). Этот термин произошел от названия десерта, представляющего собой смесь пирожного с мороженым, политую сверху шоколадной глазурью. Этот десерт продавался в супермаркетах Sommerville в штате Массачусетс. Видимо, это блюдо когда-то попробовал один из программистов, занимающийся разработкой средств объектно-ориентированного программирования для языка SCOOPS, где этот термин впервые появился.

Абстрактные типы данных

В объектном программировании довольно часто создаются иерархии логически связанных классов. Например, представим класс Shape, от которого произведены классы Rectangle и Circle. Затем от класса Rectangle производится класс Square, как частный вид прямоугольника.

В каждом из производных классов замещаются методы Draw(), GetArea() и др. Основной костяк программы с классом Shape и производными от него Rectangle и Circle показан в листинге 13.7.

Листинг 13.7. Классы семейства Shape

```
1:  //Листинг 13.7. Классы семейства Shape
2:
3:  #include <iostream.h>
4:
5:
6:  class Shape
7:  {
8:  public:
9:     Shape(){ }
10:     virtual ~Shape(){ }
11:     virtual long GetArea() {   return -1; }
12:     virtual long GetPerim() {   return -1; }
13:     virtual void Draw() { }
14:  private:
15:  } ;
16:
17:  class Circle : public Shape
18:  {
19:  public:
20:       Circle(int radius):itsRadius(radius){ }
21:       ~Circle(){ }
22:       long GetArea() {   return 3 * itsRadius * itsRadius; }
23:       long GetPerim() {   return 6 * itsRadius; }
24:       void Draw();
25:  private:
26:     int itsRadius;
```

```
27:     int itsCircumference;
28:  } ;
29:
30:  void Circle::Draw()
31:  {
32:     cout << "Circle drawing routine here!\ n";
33:  }
34:
35:
36:  class Rectangle : public Shape
37:  {
38:  public:
39:        Rectangle(int len, int width):
40:        itsLength(len), itsWidth(width){ }
41:        virtual ~Rectangle(){ }
42:        virtual long GetArea() {    return itsLength * itsWidth; }
43:        virtual long GetPerim() { return 2*itsLength + 2*itsWidth; }
44:        virtual int GetLength() {    return itsLength; }
45:        virtual int GetWidth() {    return itsWidth; }
46:        virtual void Draw();
47:  private:
48:     int itsWidth;
49:     int itsLength;
50:  } ;
51:
52:  void Rectangle::Draw()
53:  {
54:     for (int i = 0; i<itsLength; i++)
55:     {
56:        for (int j = 0; j<itsWidth; j++)
57:        cout << "x ";
58:
59:        cout << "\ n";
60:     }
61:  }
62:
63:  class Square : public Rectangle
64:  {
65:  public:
66:        Square(int len);
67:        Square(int len, int width);
68:        ~Square(){ }
69:        long GetPerim() { return 4 * GetLength();}
70:  } ;
71:
72:  Square::Square(int len):
73:     Rectangle(len,len)
74:  { }
75:
76:  Square::Square(int len, int width):
```

```
77:     Rectangle(len,width)
78:
79:   {
80:     if (GetLength() != GetWidth())
81:       cout << "Error, not a square... a Rectangle??\ n";
82:   }
83:
84:   int main()
85:   {
86:     int choice;
87:     bool fQuit = false;
88:     Shape * sp;
89:
90:     while ( ! fQuit )
91:     {
92:       cout << "(1)Circle (2)Rectangle (3)Square (0)Quit:";
93:       cin >> choice;
94:
95:       switch (choice)
96:       {
97:       case 0: fQuit = true;
98:       break;
99:       case 1: sp = new Circle(5);
100:       break;
101:       case 2: sp = new Rectangle(4,6);
102:       break;
103:       case 3: sp = new Square(5);
104:       break;
105:       default: cout << "Please enter a number between 0 and 3" << endl;
106:       continue;
107:       break;
108:     }
109:     if(! fQuit)
110:     sp->Draw();
111:       delete sp;
112:     cout << "\ n";
113:    }
114:  return 0;
115:  }
```

РЕЗУЛЬТАТ

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit:3
x x x x x
```

```
x x x x x
x x x x x
x x x x x
x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit:0
```

Анализ В строках 6—15 объявляется класс `Shape`. Методы `GetArea()` и `GetPerim()` возвращают `-1` как сообщение об ошибке, а метод `Draw()` не выполняет никаких действий. Давайте подумаем, можно ли в принципе нарисовать форму? Можно нарисовать окружность, прямоугольник или квадрат, но форма — это абстракция, которую невозможно изобразить.

Класс `Circle` производится от класса `Shape`, и в нем замещаются три виртуальных метода. Обратите внимание, что в данном случае нет необходимости использовать ключевое слово `virtual`, поскольку виртуальность функций наследуется в производном классе. Тем не менее для напоминания о винртуальности используемых функций не лишним будет явно указать это.

Класс `Square` производится от класса `Rectangle` и наследует от него все методы, причем метод `GetPerim()` замещается в новом классе.

Все методы должны функционировать нормально в производных классах, но не в базовом классе `Shape`, поскольку невозможно создать экземпляр формы как таковой. Программа должна быть защищена от попытки пользователя создать объект этого класса. Класс `Shape` существует только для того, чтобы поддерживать интерфейс, общий для всех производных классов, поэтому об этом типе данных говорят как об абстрактном, или ADT (Abstract Data Type).

Абстрактный класс данных представляет общую концепцию, такую как форма, а не отдельные объекты, такие как окружность или квадрат. В C++ ADT по отношению к другим классам всегда выступает как базовый, для которого невозможно создать функциональный объект абстрактного класса.

Чистые виртуальные функции

C++ поддерживает создание абстрактных типов данных с чистыми виртуальными функциями. *Чистыми виртуальными функциями* называются такие, которые инициализируются нулевым значением, например:

```
virtual void Draw() = 0;
```

Класс, содержащий чистые виртуальные функции, является ADT. Невозможно создать объект для класса, который является ADT. Попытка создания объекта для такого класса вызовет сообщение об ошибке во время компиляции. Помещение в класс чистой виртуальной функции будет означать следующее:

- невозможность создания объекта этого класса;
- необходимость замещения чистой виртуальной функции в производном классе.

Любой класс, произведенный от ADT, унаследует от него чистую виртуальную функцию, которую необходимо будет заместить, чтобы получить возможность создавать объекты этого класса. Так, если класс `Rectangle` наследуется от класса `Shape`, который содержит три чистые виртуальные функции, то в классе `Rectangle` должны быть замещены все эти три функции, иначе он тоже будет ADT. В листинге 13.8 изменено объявление клас-

са Shape таким образом, чтобы он стал абстрактным типом данных. Остальная часть листинга 13.7 не изменилась, поэтому не приводится. Просто замените объявление класса в строках 7–16 листинга 13.7 листингом 13.8 и запустите программу.

Листинг 13.8. Абстрактные типы данных

```
1:   класс Shape
2:   {
3:   public:
4:      Shape(){ }
5:      ~Shape(){ }
6:      virtual long GetArea() = 0; // ошибка
7:      virtual long GetPerim()= 0;
8:      virtual void Draw() = 0;
9:   private:
10: } ;
```

РЕЗУЛЬТАТ

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x

(1)Circle (2)Rectangle (3)Square (0)Quit: 0
```

АНАЛИЗ Как видите, выполнение программы не изменилось. Просто теперь в программе невозможно создать объект класса Shape.

Абстрактные типы данных

Чтобы объявить класс как абстрактный тип данных, достаточно добавить в него одну или несколько чистых виртуальных функций. Для этого после объявления функции необходимо добавить = 0, например:

```
class Shape
{
virtual void Draw() = 0;   // чистая виртуальная функция
};
```

Выполнение чистых виртуальных функций

Обычно чистые виртуальные функции объявляются в абстрактном базовом классе и не выполняются. Поскольку невозможно создать объект абстрактного базового класса, как правило, нет необходимости и в выполнении чистой виртуальной функции. Класс ADT существует только как объявление интерфейса объектов, создаваемых в производных классах.

Тем не менее все же иногда возникает необходимость выполнения чистой виртуальной функции. Она может быть вызвана из объекта, произведенного от ADT, например чтобы обеспечить общую функциональность для всех замещенных функций. В листинге 13.9 представлен видоизмененный листинг 13.7, в котором класс `Shape` объявлен как ADT и в программе выполняется чистая виртуальная функция `Draw()`. Функция замещается в классе `Circle`, что необходимо для создания объекта этого класса, но в объявлении замещенной функции делается вызов чистой виртуальной функции из базового класса. Это средство используется для достижения дополнительной функциональности методов класса.

В данном примере дополнительная функциональность состоит в выведении на экран простого сообщения. В реальной программе чистая виртуальная функция может содержать достаточно сложный программный код, например создание окна, в котором рисуются все фигуры, выбираемые пользователем.

Листинг 13.9. Выполнение чистых виртуальных функций

```
1:   // Выполнение чистых виртуальных функций
2:
3:   #include <iostream.h>
4:
5:   class Shape
6:   {
7:   public:
8:      Shape(){ }
9: virtual ~Shape(){ }
10:      virtual long GetArea() = 0;
11:      virtual long GetPerim()= 0;
12:      virtual void Draw() = 0;
13:   private:
14:   } ;
15:
16:   void Shape::Draw()
17:   {
18:    cout << "Abstract drawing mechanism!\ n";
19:  }
20:
21:  class Circle : public Shape
22:  {
23:  public:
24:     Circle(int radius):itsRadius(radius){ }
25: virtual ~Circle(){ }
26:     long GetArea() { return 3 * itsRadius * itsRadius; }
27:     long GetPerim() { return 9 * itsRadius; }
```

```
28:     void Draw();
29:  private:
30:    int itsRadius;
31:    int itsCircumference;
32:  } ;
33:
34:  void Circle::Draw()
35:  {
36:    cout << "Circle drawing routine here!\ n";
37:    Shape::Draw();
38:  }
39:
40:
41:  class Rectangle : public Shape
42:  {
43:  public:
44:     Rectangle(int len, int width):
45:       itsLength(len), itsWidth(width){ }
46: virtual ~Rectangle(){ }
47:     long GetArea() { return itsLength * itsWidth; }
48:     long GetPerim() { return 2*itsLength + 2*itsWidth; }
49:     virtual int GetLength() { return itsLength; }
50:     virtual int GetWidth() { return itsWidth; }
51:     void Draw();
52:  private:
53:    int itsWidth;
54:    int itsLength;
55:  } ;
56:
57:  void Rectangle::Draw()
58:  {
59:    for (int i = 0; i<itsLength; i++)
60:    {
61:     for (int j = 0; j<itsWidth; j++)
62:       cout << "x ";
63:
64:     cout << "\ n";
65:    }
66:    Shape::Draw();
67:  }
68:
69:
70:  class Square : public Rectangle
71:  {
72:  public:
73:     Square(int len);
74:     Square(int len, int width);
75: virtual ~Square(){ }
76:     long GetPerim() { return 4 * GetLength();}
77:  } ;
```

```
78:
79:  Square::Square(int len):
80:    Rectangle(len,len)
81:  { }
82:
83:  Square::Square(int len, int width):
84:    Rectangle(len,width)
85:
86:  {
87:    if (GetLength() != GetWidth())
88:     cout << "Error, not a square... a Rectangle??\ n";
89:  }
90:
91:  int main()
92:  {
93:    int choice;
94:    bool fQuit = false;
95:    Shape * sp;
96:
97:    while (1)
98:   {
99:     cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
100:     cin >> choice;
101:
102:     switch (choice)
103:     {
104:      case 1: sp = new Circle(5);
105:      break;
106:      case 2: sp = new Rectangle(4,6);
107:      break;
108:      case 3: sp = new Square (5);
109:      break;
110:      default: fQuit = true;
111:      break;
112:     }
113:     if (fQuit)
114:      break;
115:
116:     sp->Draw();
117:     delete sp;
118:     cout << "\ n";
119:   }
120:   return 0;
121:  }
```

РЕЗУЛЬТАТ

```
(1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
```

РЕЗУЛЬТАТ

```
x x x x x x
Abstract drawing mechanism!

(1)Circle (2)Rectangle (3)Square (0)Quit: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
Abstract drawing mechanism!

(1)Circle (2)Rectangle (3)Square (0)Quit: 0
```

АНАЛИЗ

В строках 5–14 объявляется класс абстрактного типа данных `Shape` с тремя чистыми виртуальными функциями. Впрочем, для того чтобы класс стал ADT, достаточно было объявить в нем хотя бы один из методов как чистую виртуальную функцию.

Далее в программе все три функции базового класса замещаются в производных классах `Circle` и `Rectangle`, но одна из них — функция `Draw()` — выполняется как чистая виртуальная функция, поскольку в объявлении замещенного варианта функции в производных классах есть вызов исходной функции из базового класса. В результате выполнение этой функции в обоих производных классах приводит к выведению на экран одного и того же сообщения.

Сложная иерархия абстракций

Иногда бывает необходимо произвести один класс ADT от другого класса ADT, например для того, чтобы в производном классе ADT преобразовать в обычные методы часть функций, объявленных в базовом классе как чистые виртуальные, оставив при этом другие функции чистыми.

Так, в классе `Animal` можно объявить методы `Eat()`, `Sleep()`, `Move()` и `Reproduce()` как чистые виртуальные функции. Затем от класса `Animal` производятся классы `Mammal` и `Fish`.

Исходя из соображения, что все млекопитающие размножаются практически одинаково, имеет смысл в классе `Mammal` преобразовать метод `Reproduce()` в обычный, оставив при этом методы `Eat()`, `Sleep()` и `Move()` чистыми виртуальными функциями.

Затем от класса `Mammal` производится класс `Dog`, в котором необходимо заместить все три оставшиеся чистые виртуальные функции, чтобы получить возможность создавать объекты класса `Dog`.

Таким образом, наследование одного класса ADT от другого класса ADT позволяет объявлять общие методы для всех следующих производных классов, чтобы не замещать потом эти функции по отдельности в каждом производном классе.

В листинге 13.10 показан базовый костяк программы, в котором используется объявленный выше подход.

Листинг 13.10. Наследование класса ADT от другого класса ADT

```
1:   // Листинг 13.10.
2:   // Deriving ADTs from other ADTs
3:   #include <iostream.h>
4:
5:   enum COLOR {   Red, Green, Blue, Yellow, White, Black, Brown }   ;
```

```
6:
7:  class Animal      // Общий базовый класс для классов Mammal и Fish
8:  {
9:  public:
10:     Animal(int);
11:   virtual ~Animal() { cout << "Animal destructor...\ n"; }
12:   virtual int GetAge() const { return itsAge; }
13:   virtual void SetAge(int age) { itsAge = age; }
14:   virtual void Sleep() const = 0;
15:   virtual void Eat() const = 0;
16:   virtual void Reproduce() const = 0;
17:   virtual void Move() const = 0;
18:   virtual void Speak() const = 0;
19: private:
20:   int itsAge;
21: } ;
22:
23: Animal::Animal(int age):
24: itsAge(age)
25: {
26:   cout << "Animal constructor...\ n";
27: }
28:
29: class Mammal : public Animal
30: {
31: public:
32:   Mammal(int age):Animal(age)
33:    { cout << "Mammal constructor...\ n";}
34:   virtual ~Mammal() { cout << "Mammal destructor...\ n";}
35:   virtual void Reproduce() const
36:     { cout << "Mammal reproduction depicted...\ n"; }
37: } ;
38:
39: class Fish : public Animal
40: {
41: public:
42:   Fish(int age):Animal(age)
43:    { cout << "Fish constructor...\ n";}
44:   virtual ~Fish() { cout << "Fish destructor...\ n"; }
45:   virtual void Sleep() const { cout << "fish snoring...\ n"; }
46:   virtual void Eat() const { cout << "fish feeding...\ n"; }
47:   virtual void Reproduce() const
48:    { cout << "fish laying eggs...\ n"; }
49:   virtual void Move() const
50:     { cout << "fish swimming...\ n";  }
51:   virtual void Speak() const { }
52: } ;
53:
54: class Horse : public Mammal
55: {
```

```
56:  public:
57:    Horse(int age, COLOR color ):
58:    Mammal(age), itsColor(color)
59:       { cout << "Horse constructor...\ n"; }
60:    virtual ~Horse() { cout << "Horse destructor...\ n"; }
61:    virtual void Speak()const { cout << "Whinny!... \ n"; }
62:    virtual COLOR GetItsColor() const { return itsColor; }
63:    virtual void Sleep() const
64:      { cout << "Horse snoring...\ n"; }
65:    virtual void Eat() const { cout << "Horse feeding...\ n"; }
66:    virtual void Move() const { cout << "Horse running...\ n";}
67:
68:  protected:
69:    COLOR itsColor;
70:  } ;
71:
72:  class Dog : public Mammal
73:  {
74:  public:
75:    Dog(int age, COLOR color ):
76:     Mammal(age), itsColor(color)
77:       { cout << "Dog constructor...\ n"; }
78:    virtual ~Dog() { cout << "Dog destructor...\ n"; }
79:    virtual void Speak()const { cout << "Woof!... \ n"; }
80:    virtual void Sleep() const { cout << "Dog snoring...\ n"; }
81:    virtual void Eat() const { cout << "Dog eating...\ n"; }
82:    virtual void Move() const { cout << "Dog running...\ n"; }
83:    virtual void Reproduce() const
84:      { cout << "Dogs reproducing...\ n"; }
85:
86:  protected:
87:    COLOR itsColor;
88:  } ;
89:
90:  int main()
91:  {
92:    Animal *pAnimal=0;
93:    int choice;
94:    bool fQuit = false;
95:
96:    while (1)
97:    {
98:     cout << "(1)Dog (2)Horse (3)Fish (0)Quit: ";
99:     cin >> choice;
100:
101:     switch (choice)
102:     {
103:       case 1: pAnimal = new Dog(5,Brown);
104:       break;
105:       case 2: pAnimal = new Horse(4,Black);
```

```
106:       break;
107:      case 3: pAnimal = new Fish (5);
108:      break;
109:      default: fQuit = true;
110:      break;
111:     }
112:     if (fQuit)
113:      break;
114:
115:     pAnimal->Speak();
116:     pAnimal->Eat();
117:     pAnimal->Reproduce();
118:     pAnimal->Move();
119:     pAnimal->Sleep();
120:     delete pAnimal;
121:     cout << "\ n";
122:   }
123:   return 0;
124: }
```

РЕЗУЛЬТАТ

```
(1)Dog (2)Horse (3)Bird (0)Quit: 1
Animal constructor...
Mammal constructor...
Dog constructor...
Woof!...
Dog eating...
Dog reproducing....
Dog running...
Dog snoring...
Dog destructor...
Mammal destructor...
Animal destructor...

(1)Dog (2)Horse (3)Bird (0)Quit: 0
```

АНАЛИЗ В строках 7–21 объявляется абстрактный тип данных `Animal`. Единственный метод этого класса, не являющийся чистой виртуальной функцией, это общий для объектов всех производных классов метод `itsAge`. Остальные пять методов — `Sleep()`, `Eat()`, `Reproduce()`, `Move()` и `Speak()` — объявлены как чистые виртуальные функции.

Класс `Mammal` производится от `Animal` в строках 29–37 и не содержит никаких данных. В нем замещается функция `Reproduce()`, чтобы задать способ размножения, общий для всех млекопитающих. Класс `Fish` производится непосредственно от класса `Animal`, поэтому функция `Reproduce()` в нем замещается иначе, чем в классе `Mammal` (и это соответствует реальности).

Во всех других классах, производимых от класса `Mammal`, теперь нет необходимости замещать общий для всех метод `Reproduce()`, хотя при желании это можно сделать для определенного класса, как, например, в нашей программе это было сделано в строке 83 для класса `Dog`. Все остальные чистые виртуальные функции были замещены в классах `Fish`, `Horse` и `Dog`, поэтому для каждого из них можно создавать соответствующие объекты.

В теле программы используется указатель класса Animal, с помощью которого делаются ссылки на все объекты производных классов. В зависимости от того, с каким объектом связан указатель в текущий момент, вызываются соответствующие виртуальные функции.

При попытке создать объекты для классов абстрактных типов данных Animal или Mammal компилятор покажет сообщение об ошибке.

Когда следует использовать абстрактные типы данных

В одних примерах программ, рассмотренных нами ранее, класс Animal являлся абстрактным типом данных, в других — нет. В каких же случаях нужно объявлять класс как абстрактный тип данных?

Нет никаких правил, которые требовали бы объявления класса как абстрактного. Программист принимает решение о создании абстрактного типа данных, основываясь на том, какую роль играет этот класс в программе. Так, если вы хотите смоделировать виртуальную ферму или зоопарк, то имеет смысл класс Animal объявить как абстрактный и для создания объектов производить от него другие классы, такие как Dog.

Если же вы хотите смоделировать виртуальную псарню, то теперь класс Dog будет абстрактным, от которого можно производить подклассы, представляющие разные породы собак. Количество уровней абстрактных классов следует выбирать в зависимости от того, насколько детально вы хотите смоделировать реальный объект или явление.

Рекомендуется	Не рекомендуется
Используйте абстрактные типы данных для создания общего интерфейса для всех производных классов. **Обязательно замещайте** в производных классах все чистые виртуальные функции. **Объявляйте** все функции, которые нужно замещать в производных классах, как чистые виртуальные функции.	**Не пытайтесь** создать объект абстрактного класса.

Логика использования абстрактных классов

В последнее время в программировании на C++ активно используется концепция создания абстрактных логических конструкций. С помощью таких конструкций можно находить решения для многих общих задач и создавать при этом программы, которые легко читаются и документируются. Рассмотрим пример создания логической конструкции с использованием наследования классов.

Представим, что нужно создать класс Timer, который умеет отсчитывать секунды. Такой класс может иметь целочисленную переменную-член itsSeconds, а также метод, осуществляющий приращение переменной itsSeconds.

Теперь предположим, что программа должна отслеживать и сообщать о каждом изменении переменной itsSeconds. Первое решение, которое приходит на ум, — это добавить в класс Timer метод уведомления об изменении переменной-члена. Но логически это не совсем верно, так как программа уведомления может быть достаточно сложной и по сути своей не является логической частью программы отсчета времени.

Гораздо логичнее рассматривать программу отслеживания и информирования об изменении переменной как абстрактный класс, который в равной степени может использоваться как с программой отсчета времени, так и с любой другой программой с периодически изменяющимися переменными.

Таким образом, лучшим решением будет создание абстрактного класса обозревателя `Observer` с чистой виртуальной функцией `Update()`.

Теперь создадим второй абстрактный класс — `Subject`. Он содержит массив объектов класса `Observer`. Кроме того, в нем объявлены два дополнительных метода: `Register()`, который регистрирует объекты класса `Observer`, и `Notify()`, который отслеживает изменения указанной переменной.

Эта конструкция классов может затем использоваться во многих программах. Те классы, которые будут отслеживать изменения и сообщать о них, наследуются от класса `Observer`. Класс `Timer` в нашем примере наследуется от класса `Subject`. При изменении контролируемой переменной (в нашем примере — `itsSeconds`) вызывается метод `Notify()`, унаследованный от класса `Subject`.

Наконец, можно создать новый класс `ObserverTimer`, унаследованный сразу от двух базовых классов — `Observer` и `Timer`, который будет сочетать в себе возможности отсчитывать время и сообщать об этом.

Пара слов о множественном наследовании, абстрактных типах данных и языке Java

Многие программисты знают, что в основу языка Java положен C++. Также известно, что создатели языка Java удалили из него возможность множественного наследования потому, что, по их мнению, это средство слишком усложняет программный код и идет в разрез с концепцией упрощения программных кодов, положенной в основу Java. С точки зрения создателей Java, 90% всех возможностей, предоставляемых множественным наследованием, можно получить с помощью интерфейса.

Интерфейс в терминологии Java представляет собой нечто подобное абстрактному типу данных, в том смысле, что в нем также определяются функции, которые могут быть реализованы только в производных классах. Но новые классы не производятся непосредственно от интерфейса. Классы производят от других классов и в них передаются функции интерфейса, что напоминает множественное наследование. Так, союз абстрактных классов и множественного наследования породил на свет аналог классов-мандатов, в результате чего удалось избежать чрезмерного усложнения программных кодов, как в случае с множественным наследованием. Кроме того, поскольку интерфейсы не содержат ни выполняемых функций, ни переменных-членов, отпадает необходимость в виртуальном наследовании.

Насколько удобны или целесообразны эти изменения, зависит от привычек конкретного программиста. Во всяком случае, если вы хорошо разберетесь в множественном наследовании и абстрактных типах данных языка C++, то это послужит хорошей базой при изучении и освоении последних достижений и тенденций программирования, реализованных в языке Java (если у вас возникнет интерес к нему).

Использование логических конструкций в языках C++ и Java подробно рассматривается в следующей статье: Robert Martin, C++ and Java: A Critical Comparison // C++ Report. — January, 1997.

Резюме

Сегодня вы познакомились с методами преодоления некоторых ограничений одиночного наследования. Вы узнали об опасности передачи вверх по иерархии классов интерфейса производных функций и об ограничениях приведения типа данных объектов базового класса к производным классам во время выполнения программы. Кроме того, вы узнали, когда и как используется множественное наследование классов, какие проблемы при этом могут возникнуть и как их преодолеть.

На этом занятии также было представлено объявление абстрактных типов данных и способы создания абстрактного класса с помощью чистых виртуальных функций. Особое внимание уделялось логике использования абстрактных данных для моделирования реальных ситуаций.

Вопросы и ответы

Что означает передача функциональности вверх по иерархии классов?

Речь идет о переносе описаний общих функций-членов в базовые классы более высокого уровня. Если одна и та же функция используется в производных классах, имеет смысл описать эту функцию в общем для них базовом классе.

Во всех ли случаях передача функциональности вверх целесообразна в программе?

Если передаются вверх по иерархии только функции общего использования, то это целесообразно, но смысл теряется, если в базовые классы передается специфичный интерфейс производных классов. Другими словами, если метод не может быть использован во всех производных классах, то нет смысла описывать его в базовом классе. В противном случае вам во время выполнения программы придется отслеживать тип текущего объекта, прежде чем вызвать функцию.

В чем проблема с контролем типа объекта при выполнении программы?

В больших программах для выполнения контроля за типом объекта придется использовать достаточно массивный и сложный программный блок. Идея использования виртуальных функций состоит в том, что тип объекта определяется программой автоматически с помощью виртуальной таблицы, вместо того чтобы использовать для этого специальные программные блоки.

Что плохого в приведении типа объектов?

Приведение типов объектов к определенному типу данных, используемому конкретной функцией, довольно часто и эффективно используется в программах на C++. Но если программист применяет приведение типов для того, чтобы обойти заложенный в C++ строгий контроль за соответствием типов данных, например в случае приведения типа указателя к установленному во время выполнения программы типу объекта, то это говорит о серьезных недостатках в структуре программы, противоречащих идеологии C++.

Почему бы не сделать все функции виртуальными?

Для поддержания работы виртуальных функций создается виртуальная таблица, что увеличивает потребление памяти программой и время выполнения программы. Если в программе используется небольшой класс, от которого не производятся подклассы, то в использовании виртуальных функций нет никакого смысла.

В каких случаях используются виртуальные деструкторы?

Виртуальные деструкторы следует описывать в том случае, если в программе планируется использование указателя базового класса для получения доступа к объектам подклассов. Существует одно простое правило: если в программе описываются виртуальные функции, то обязательно должны использоваться виртуальные деструкторы.

Для чего возиться с созданием абстрактных типов данных? Не проще ли создать обычный базовый класс, для которого просто не создавать объектов в программе?

При написании программы всегда следует использовать такие подходы, которые гарантировали бы обнаружение ошибок в программе не во время ее выполнения, а во время компиляции. Если класс явно будет описан как абстрактный, то любая попытка создать объект этого класса приведет к показу компилятором сообщения об ошибке.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Что такое приведение типа объекта вниз?
2. Что такое `v-ptr`?
3. Предположим, для создания прямоугольника с закругленными углами используется класс `RoundRect`, произведенный от двух базовых классов — `Rectangle` и `Circle`, которые, в свою очередь, производятся от общего класса `Shape`. Как много объектов класса `Shape` создается при создании одного объекта класса `RoundRect`?
4. Если классы `Horse` и `Bird` виртуально наследуются от класса `Animal` как открытые, будут ли конструкторы этих классов инициализировать конструктор класса `Animal`? Если класс `Pegasus` наследуется сразу от двух классов, `Horse` и `Bird`, как в нем будет инициализироваться конструктор класса `Animal`?
5. Объявите класс `Vehicle` (Машина) как абстрактный тип данных.
6. Если в программе объявлен класс ADT с тремя чистыми виртуальными функциями, сколько из них нужно заместить в производных классах, чтобы получить возможность создания объектов этих классов?

Упражнения

1. Объявите класс `JetPlane` (Реактивный самолет), наследуя его от двух базовых классов — `Rocket` (Ракета) и `Airplane` (Самолет).
2. Произведите от класса `JetPlane`, объявленного в первом упражнении, новый класс `747`.

3. Напишите программу, производящую классы `Car` (Легковой автомобиль) и `Bus` (Автобус) от класса `Vehicle` (Машина). Объявите класс `Vehicle` как абстрактный тип данных с двумя чистыми виртуальными функциями. Классы `Car` и `Bus` не должны быть абстрактными.
4. Измените программу из предыдущего упражнения таким образом, чтобы класс `Car` тоже стал ADT, и произведите от него три новых класса: `SportsCar` (Спортивный автомобиль), `Wagon` (Фургон) и `Coupe` (Двухместный автомобиль-купе). В классе `Car` должна замещаться одна из виртуальных функций, объявленных в классе `Vehicle`, с вызовом функции базового класса.

Специальные классы и функции

Язык программирования C++ предлагает несколько способов ограничения области видимости и использования переменных и указателей. В предыдущих главах вы научились создавать глобальные переменные, используемые во всей программе, и локальные переменные, используемые в отдельных функциях. Вы узнали, что собой представляют указатели на переменные и переменные-члены класса. Сегодня вы узнаете:

- Что такое статические переменные-члены и функции-члены
- Как используются статические переменные-члены и функции-члены
- Как создавать и применять указатели на функции и на функции-члены
- Как работать с массивами указателей на функции

Статические переменные-члены

До настоящего момента вы считали, что всякие данные объекта уникальны для того объекта, в котором используются, и не могут совместно применяться несколькими объектами класса. Другими словами, если было создано пять объектов `Cat`, то каждый из них характеризуется своим временем жизни, размерами и т.п. При этом время жизни одного не влияет на время жизни остальных.

Однако иногда возникает необходимость контроля за накоплением данных программой. Может потребоваться информация о том, сколько всего было создано объектов определенного класса и сколько их существует в данный момент. Статические переменные-члены совместно используются всеми объектами класса. Они являются чем вроде "золотой серединки" между глобальными данными, доступными всем частям программы, и данными членов, доступными, как правило, только одному объекту.

Можно полагать, что статические члены принадлежат классу, а не объекту. Если данные обычных членов доступны одному объекту, то статические члены могут использоваться всем классом. В листинге 14.1 объявляется объект `Cat` со статическим членом `HowManyCats`. Эта переменная учитывает количество созданных объектов `Cat`, что реализуется приращением статической переменной `HowManyCats` при вызове конструктора или отрицательным приращением при вызове деструктора.

Листинг 14.1. Статические переменные-члены

```
1:  //Листинг 14.1. Статические переменные-члены
2:
3:  #include <iostream.h>
4:
5:  class Cat
6:  {
7:  public:
8:   Cat(int age):itsAge(age){ HowManyCats++; }
9:   virtual ~Cat() { HowManyCats--; }
10:   virtual int GetAge() { return itsAge; }
11:   virtual void SetAge(int age) { itsAge = age; }
12:   static int HowManyCats;
13:
14: private:
15:   int itsAge;
16:
17: } ;
18:
19: int Cat::HowManyCats = 0;
20:
21: int main()
22: {
23:   const int MaxCats = 5; int i;
24:   Cat *CatHouse[MaxCats];
25:   for (i = 0; i<MaxCats; i++)
26:    CatHouse[i] = new Cat(i);
27:
28:   for (i = 0; i<MaxCats; i++)
29:   {
30:    cout << "There are ";
31:    cout << Cat::HowManyCats;
32:    cout << " cats left!\ n";
33:    cout << "Deleting the one which is ";
34:    cout << CatHouse[i]->GetAge();
35:    cout << " years old\ n";
36:    delete CatHouse[i];
37:    CatHouse[i] = 0;
38:   }
39:  return 0;
40: }
```

РЕЗУЛЬТАТ

```
There are 5 cats left!
Deleting the one which is 0 years old
There are 4 cats left!
Deleting the one which is 1 years old
There are 3 cats left!
Deleting the one which is 2 years old
```

```
There are 2 cats left!
Deleting the one which is 3 years old
There are 1 cats left!
Deleting the one which is 4 years old
```

АНАЛИЗ Обычный класс Cat объявляется в строках 5–17. С помощью ключевого слова static в строке 12 объявляется статическая переменная-член HowManyCats типа int.

Объявление статической переменной HowManyCats само по себе не определяет никакого целочисленного значения, т.е. в памяти компьютера не резервируется область для данной переменной при ее объявлении, поскольку, по сути, она не является переменной-членом конкретного объекта Cat. Определение и инициализация переменной HowManyCats происходит в строке 19.

Не забывайте отдельно определять статическую переменную-член класса (весьма распространенная ошибка среди начинающих программистов). В противном случае редактор связей во время компиляции программы выдаст следующее сообщение об ошибке:

```
undefined symbol Cat::HowManyCats
```

Обратите внимание, что для обычной переменной-члена itsAge не требуется отдельное определение, поскольку обычные переменные-члены определяются автоматически каждый раз при создании объекта Cat, как, например, в строке 26.

Конструктор объекта Cat, объявленный в строке 8, увеличивает значение статической переменной-члена на единицу. Деструктор, объявленный в строке 9, уменьшает это значение на 1. Таким образом, в любой момент времени переменная HowManyCats отражает текущее количество созданных объектов класса Cat.

В строках программы 21–40 создается пять объектов Cat, указатели на которые заносятся в массив. Это сопровождается пятью вызовами конструктора класса Cat, в результате чего пять раз происходит приращение на единицу переменной HowManyCats, начиная с исходного значения 0.

Затем в программе цикл for последовательно удаляет все объекты Cat из массива, предварительно выводя на экран текущее значение переменной HowManyCats. Вывод начинается со значения 5 (ведь было создано пять объектов) и с каждым циклом уменьшается.

Обратите внимание: переменная HowManyCats объявлена как public и может вызываться из функции main(). Однако нет веских причин объявлять эту переменную-член таким образом. Если предполагается обращаться к статической переменной только через объекты класса Cat, предпочтительней сделать ее закрытой вместе с другими переменными-членами и создать открытый метод доступа. С другой стороны, если необходимо получать прямой доступ к данным без использования объекта Cat, то можно либо оставить ее открытой, как показано в листинге 14.2, либо создать статическую функцию-член. Реализация последнего варианта рассматривается далее в этой главе.

Листинг 14.2. Доступ к статическим членам без использования объектов

```
1:   //Листинг 14.2. Статические переменные-члены
2:
3:   #include <iostream.h>
4:
5:   class Cat
6:   {
7:   public:
```

```
8:    Cat(int age):itsAge(age){ HowManyCats++; }
9:    virtual ~Cat() { HowManyCats--; }
10:    virtual int GetAge() { return itsAge; }
11:    virtual void SetAge(int age) {itsAge = age;}
12:    static int HowManyCats;
13:
14:  private:
15:    int itsAge;
16:
17:  } ;
18:
19:  int Cat::HowManyCats = 0;
20:
21:  void TelepathicFunction();
22:
23:  int main()
24:  {
25:    const int MaxCats = 5; int i;
26:    Cat *CatHouse[MaxCats];
27:    for (i = 0; i<MaxCats; i++)
28:    {
29:     CatHouse[i] = new Cat(i);
30:     TelepathicFunction();
31:    }
32:
33:    for ( i = 0; i<MaxCats; i++)
34:    {
35:     delete CatHouse[i];
36:     TelepathicFunction();
37:    }
38:    return 0;
39:  }
40:
41:  void TelepathicFunction()
42:  {
43:    cout << "There are ";
44:    cout << Cat::HowManyCats << " cats alive!\ n";
45:  }
```

РЕЗУЛЬТАТ

```
There are 1 cats alive!
There are 2 cats alive!
There are 3 cats alive!
There are 4 cats alive!
There are 5 cats alive!
There are 4 cats alive!
There are 3 cats alive!
There are 2 cats alive!
There are 1 cats alive!
There are 0 cats alive!
```

Листинг 14.2 аналогичен листингу 14.1, однако включает новую функцию TelepahicFunction(). Она не создает объект Cat и даже не использует его в качестве параметра, однако может получить доступ к переменной-члену HowManyCats. Не лишним будет еще раз напомнить, что эта переменная-член относится не к какому-либо определенному объекту, а ко всему классу в целом. Поэтому если она объявлена как public, то может использоваться любой функцией программы.

Если статическая переменная-член будет объявлена как закрытая, то доступ к ней можно получить с помощью функции-члена. Но для этого необходимо наличие хотя бы одного объекта данного класса. Именно такой подход реализован в листинге 14.3. Затем мы перейдем к изучению статических функций-членов.

Листинг 14.3. Доступ к статическим членам с помощью обычных функций-членов

```
1:  //Листинг 14.3. Закрытые статические переменные-члены
2:
3:  #include <iostream.h>
4:
5:  class Cat
6:  {
7:  public:
8:   Cat(int age):itsAge(age){ HowManyCats++; }
9:   virtual ~Cat() { HowManyCats--; }
10:   virtual int GetAge() { return itsAge; }
11:   virtual void SetAge(int age) { itsAge = age; }
12:   virtual int GetHowMany() { return HowManyCats; }
13:
14:
15: private:
16:   int itsAge;
17:   static int HowManyCats;
18: } ;
19:
20: int Cat::HowManyCats = 0;
21:
22: int main()
23: {
24:   const int MaxCats = 5; int i;
25:   Cat *CatHouse[MaxCats];
26:   for (i = 0; i<MaxCats; i++)
27:    CatHouse[i] = new Cat(i);
28:
29:   for (i = 0; i<MaxCats; i++)
30:   {
31:    cout << "There are ";
32:    cout << CatHouse[i]->GetHowMany();
33:    cout << " cats left!\ n";
34:    cout << "Deleting the one which is ";
35:    cout << CatHouse[i]->GetAge()+2;
36:    cout << " years old\ n";
37:    delete CatHouse[i];
```

```
38:      CatHouse[i] = 0;
39:    }
40:   return 0;
41:  }
```

Результат:

```
There are 5 cats left!
Deleting the one which is 2 years old
There are 4 cats left!
Deleting the one which is 3 years old
There are 3 cats left!
Deleting the one which is 4 years old
There are 2 cats left!
Deleting the one which is 5 years old
There are 1 cats left!
Deleting the one which is 6 years old
```

Анализ: В строке 17 статическая переменная-член `HowManyCats` объявлена как `private`. Поэтому теперь доступ к ней закрыт для функций, не являющихся членами класса, например для функции `TelepathicFunction` из предыдущего листинга.

Хотя переменная `HowManyCats` является статической, она все же находится в области видимости класса. Поэтому любая функция класса, например `GetHoqMany()`, может получить доступ к ней так же, как к любой обычной переменной-члену. Однако для вызова `GetHowMany()` функция должна иметь объект, через который осуществляется вызов.

Рекомендуется	Не рекомендуется
Применяйте статические переменные-члены для совместного использования данных несколькими объектами класса. **Ограничьте** доступ к статическим переменным-членам, объявив их как `private` или `protected`.	**Не используйте** статические переменные-члены для хранения данных одного объекта. Эти переменные предназначены для обмена данными между объектами.

Статические функции-члены

Статические функции-члены подобны статическим переменным-членам: они не принадлежат одному объекту, а находятся в области видимости всего класса. Именно поэтому их можно вызывать даже в тех случаях, когда не было создано ни одного объекта класса, как показано в листинге 14.4.

Листинг 14.4. Статические функции-члены

```
1:   //Листинг 14.4. Статические функции-члены
2:
3:   #include <iostream.h>
4:
5:   class Cat
```

```
6:  {
7:  public:
8:   Cat(int age):itsAge(age){ HowManyCats++; }
9:   virtual ~Cat() { HowManyCats--; }
10:   virtual int GetAge() { return itsAge; }
11:   virtual void SetAge(int age) { itsAge = age; }
12:   static int GetHowMany() { return HowManyCats; }
13: private:
14:   int itsAge;
15:   static int HowManyCats;
16: } ;
17:
18: int Cat::HowManyCats = 0;
19:
20: void TelepathicFunction();
21:
22: int main()
23: {
24:   const int MaxCats = 5;
25:   Cat *CatHouse[MaxCats]; int i;
26:   for (i = 0; i<MaxCats; i++)
27:   {
28:    CatHouse[i] = new Cat(i);
29:    TelepathicFunction();
30:   }
31:
32:   for ( i = 0; i<MaxCats; i++)
33:   {
34:    delete CatHouse[i];
35:    TelepathicFunction();
36:   }
37:   return 0;
38: }
39:
40: void TelepathicFunction()
41: {
42:   cout << "There are " << Cat::GetHowMany() << " cats alive!\ n";
43: }
```

Результат

```
There are 1 cats alive!
There are 2 cats alive!
There are 3 cats alive!
There are 4 cats alive!
There are 5 cats alive!
There are 4 cats alive!
There are 3 cats alive!
There are 2 cats alive!
There are 1 cats alive!
There are 0 cats alive!
```

АНАЛИЗ В строке 15 в объявлении класса `Cat` создается закрытая статическая переменная-член `HowManyCats`. В строке 12 объявляется открытая статическая функция-член `GetHowMany()`.

Так как функция `GetHowMany()` открыта, доступ к ней может получить любая другая функция, а при объявлении ее статической отпадает необходимость в существовании объекта типа `Cat`. Именно поэтому функция `TelepathicFunction()` в строке 42 может получить доступ к `GetHowMany()`, не имея доступа к объекту `Cat`. Конечно же, к функции `GetHowMany()` можно было обратиться из блока `main()` так же, как к обычным методам объектов `Cat`.

ПРИМЕЧАНИЕ

Статические функции-члены не содержат указателя `this`. Поэтому они не могут объявляться со спецификатором `const`. Кроме того, поскольку функции-члены получают доступ к переменным-членам с помощью указателя `this`, статические функции-члены не могут использовать обычные нестатические переменные-члены!

Статические функции-члены

Доступ к статическим функциям-членам можно получить, либо вызывая их из объектов класса как обычные функции-члены, либо вызывая их без объектов, явно указав в этом случае имя класса.

Пример:

```
class Cat
  {
  public:
  static int GetHowMany() { return HowManyCats; }
  private:
  static int HowManyCats;
  };
  int Cat::HowManyCats = 0;
  int main()
  {
  int howMany;
  Cat theCat;          // определение объекта
  howMany = theCat.GetHowMany();   // доступ через объект
  howMany = Cat::GetHowMany();   // доступ без объекта
  }
```

Указатели на функции

Точно так же, как имя массива постоянно указывает на его первый элемент, имя функции является указателем на саму функцию. Можно объявить переменную-указатель функции и в дальнейшем вызывать ее с помощью этого указателя. Такая возможность может оказаться весьма полезной, поскольку позволяет создавать программы, в которых функции вызываются по командам пользователя, вводимым с клавиатуры.

Единственная важная деталь для определения указателя на функцию — знание типа объекта, на который ссылается указатель. Указатель типа `int` обязательно связан с целочисленной переменной. Аналогичным образом указатель на функцию может вызывать только функции с заданными сигнатурой и типом возврата.

В объявлении

```
long (* funcPtr) (int);
```

создается указатель на функцию funcPtr (обратите внимание на символ * перед именем указателя), которая принимает целочисленный параметр и возвращает значение типа long. Круглые скобки вокруг (* funcPtr) обязательны, поскольку скобки вокруг (int) имеют больший приоритет по сравнению с оператором косвенного обращения (*). Если убрать первые скобки, то это выражение будет объявлять функцию funcPtr, принимающую целочисленный параметр и возвращающую указатель на значение типа long. (Вспомните, что все пробелы в C++ игнорируются.)

Рассмотрим два следующих объявления:

```
long * Function (int);
long (* funcPtr) (int);
```

В первой строке Function() — это функция, принимающая целочисленный параметр и возвращающая указатель на переменную типа long. Во втором примере funcPtr — это указатель на функцию, принимающую целочисленный параметр и возвращающую переменную типа long.

Объявление указателя на функцию всегда содержит тип возвращаемой переменной и заключенный в скобки список типов формальных параметров, если таковые имеются. Пример объявления и использования указателя на функцию показан в листинге 14.5.

Листинг 14.5. Указатели на функции

```
1:  // Листинг 14.5. Использование указателей на функции
2:
3:  #include <iostream.h>
4:
5:  void Square (int&,int&);
6:  void Cube (int&, int&);
7:  void Swap (int&, int &);
8:  void GetVals(int&, int&);
9:  void PrintVals(int, int);
10:
11: int main()
12: {
13:   void (* pFunc) (int &, int &);
14:   bool fQuit = false;
15:
16:   int valOne=1, valTwo=2;
17:   int choice;
18:   while (fQuit == false)
19:   {
20:    cout << "(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap";
21:    cin >> choice;
22:    switch (choice)
23:    {
24:      case 1: pFunc = GetVals; break;
25:      case 2: pFunc = Square; break;
26:      case 3: pFunc = Cube; break;
27:      case 4: pFunc = Swap; break;
28:      default : fQuit = true; break;
```

```
29:     }
30:
31:     if (fQuit)
32:       break;
33:
34:     PrintVals(valOne, valTwo);
35:     pFunc(valOne, valTwo);
36:     PrintVals(valOne, valTwo);
37:    }
38:   return 0;
39:  }
40:
41:  void PrintVals(int x, int y)
42:  {
43:    cout << "x: " << x << " y: " << y << endl;
44:  }
45:
46:  void Square (int & rX, int & rY)
47:  {
48:    rX *= rX;
49:    rY *= rY;
50:  }
51:
52:  void Cube (int & rX, int & rY)
53:  {
54:    int tmp;
55:
56:    tmp = rX;
57:    rX *= rX;
58:    rX = rX * tmp;
59:
60:    tmp = rY;
61:    rY *= rY;
62:    rY = rY * tmp;
63:  }
64:
65:  void Swap(int & rX, int & rY)
66:  {
67:    int temp;
68:    temp = rX;
69:    rX = rY;
70:    rY = temp;
71:  }
72:
73:  void GetVals (int & rValOne, int & rValTwo)
74:  {
75:    cout << "New value for ValOne: ";
76:    cin >> rValOne;
77:    cout << "New value for ValTwo: ";
78:    cin >> rValTwo;
79:  }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
New value for ValOne: 2
New value for ValTwo: 3
x: 2 y: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y: 64
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

АНАЛИЗ

В строках 5–8 объявляются четыре функции с одинаковыми типами возврата и сигнатурами. Все эти функции возвращают `void` и принимают ссылки на значения типа `int`.

В строке 13 переменная `pFunc` объявлена как указатель на функцию, принимающую две ссылки на `int` и возвращающую `void`. Этот указатель может ссылаться на каждую из упоминавшихся ранее функций. Пользователю предлагается выбрать функцию, после чего она связывается с указателем `pFunc`. В строках 34–36 выводятся текущие значения двух целочисленных переменных, вызывается текущая функция и выводятся результаты вычислений.

Указатель на функцию

Обращение к функции через указатель записывается так же, как и обычный вызов функции, на которую он указывает. Просто вместо имени функции используется имя указателя на эту функцию.

Чтобы связать указатель на функцию с определенной функцией, нужно просто присвоить ему имя функции без каких-либо скобок. Имя функции, как вы уже знаете, представляет собой константный указатель на саму функцию. Поэтому указатель на функцию используется так же, как и ее имя. При вызове функции через указатель следует задать все параметры, установленные для текущей функции.

Пример:

```
long(*pFuncOne) (int,int);
long SomeFunction (int,int);
pFuncOne = SomeFunction;
pFuncOne (5,7);
```

Зачем нужны указатели на функции

В программе, показанной в листинге 14.5, можно было бы обойтись и без указателей на функции, однако с их помощью значительно упрощается и становится читабельнее код программы: достаточно только выбрать функцию из списка и затем вызвать ее.

В листинге 14.6 используются прототипы и объявления функций листинга 14.5, но отсутствуют указатели на функции. Оцените различия между этими двумя листингами.

Листинг 14.6. Видоизмененный вариант листинга 14.5 без использования указателей на функции

```
1:  // Листинг 14.6. Видоизмененный вариант листинга 14.5 без использования
    // указателей на функции
2:
3:  #include <iostream.h>
4:
5:  void Square (int&,int&);
6:  void Cube (int&, int&);
7:  void Swap (int&, int &);
8:  void GetVals(int&, int&);
9:  void PrintVals(int, int);
10:
11: int main()
12: {
13:   bool fQuit = false;
14:   int valOne=1, valTwo=2;
15:   int choice;
16:   while (fQuit == false)
17:   {
18:    cout << << "(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap";
19:    cin >> choice;
20:    switch (choice)
21:    {
22:      case 1:
23:       PrintVals(valOne, valTwo);
24:       GetVals(valOne, valTwo);
25:       PrintVals(valOne, valTwo);
26:       break;
27:
28:      case 2:
29:       PrintVals(valOne, valTwo);
30:       Square(valOne,valTwo);
31:       PrintVals(valOne, valTwo);
32:       break;
33:
34:      case 3:
35:       PrintVals(valOne, valTwo);
36:       Cube(valOne, valTwo);
37:       PrintVals(valOne, valTwo);
38:       break;
39:
40:      case 4:
41:       PrintVals(valOne, valTwo);
42:       Swap(valOne, valTwo);
43:       PrintVals(valOne, valTwo);
44:       break;
45:
46:      default :
```

```
47:         fQuit = true;
48:         break;
49:       }
50:
51:       if (fQuit)
52:         break;
53:     }
54:    return 0;
55:  }
56:
57:  void PrintVals(int x, int y)
58:  {
59:    cout << "x: " << x << " y: " << y << endl;
60:  }
61:
62:  void Square (int & rX, int & rY)
63:  {
64:    rX *= rX;
65:    rY *= rY;
66:  }
67:
68:  void Cube (int & rX, int & rY)
69:  {
70:    int tmp;
71:
72:    tmp = rX;
73:    rX *= rX;
74:    rX = rX * tmp;
75:
76:    tmp = rY;
77:    rY *= rY;
78:    rY = rY * tmp;
79:  }
80:
81:  void Swap(int & rX, int & rY)
82:  {
83:    int temp;
84:    temp = rX;
85:    rX = rY;
86:    rY = temp;
87:  }
88:
89:  void GetVals (int & rValOne, int & rValTwo)
90:  {
91:    cout << "New value for ValOne: ";
92:    cin >> rValOne;
93:    cout << "New value for ValTwo: ";
94:    cin >> rValTwo;
95:  }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
New value for ValOne: 2
New value for ValTwo: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y: 64
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

АНАЛИЗ

Функции работают так же, как и в листинге 14.5. Информация, выводимая программой на экран, также не изменилась. Но размер программы увеличился с 22 до 46 строк. Причина в том, что вызов функции `PrintVals()`приходится повторять для каждого блока с оператором `case`.

Заманчивым может показаться вариант размещения функции `PrintVals()` вверху и внизу цикла `while`, а не в каждом блоке с оператором `case`. Но тогда функция `PrintVals()` будет вызываться даже в случае выхода из цикла, чего быть не должно.

Упрощенный вариант вызова функции

Имя указателя на функцию вовсе не должно дублировать имя самой функции, хотя вы вполне вправе это сделать. Пусть, например, `pFunc` — указатель на функцию, принимающую целочисленное значение и возвращающую переменную типа `long`, и имя этой функции — `pFunc`. У вас есть возможность вызвать ее любым из двух обращений:

```
pFunc(x);
```

или

```
(*pFunc) (x);
```

Оба выражения приведут к одному и тому же результату. Хотя первое выражение короче, второе придает программе больше гибкости.

Увеличение размера кода за счет повторяющихся вызовов одной и той же функции ухудшает читабельность программы. Этот вариант приведен специально для того, чтобы показа эффективность использования указателей на функции. В реальных условиях преимущества применения указателей на функции еще более очевидны, так как они позволяют исключить дублирование кода и делают программу более четкой. Например, указатели на функции можно собрать в один массив и вызывать из него функции в зависимости от текущей ситуации.

Массивы указателей на функции

Аналогично объявлению массива указателей целых чисел можно объявить массив указателей на функции с определенной сигнатурой, возвращающих значения определенного типа. Листинг 14.7 является еще одним вариантом программы из листинга 14.5, в которой все указатели на функции собраны в массив.

Листинг 14.7. Использование массива указателей на функции

```
1:  // Листинг 14.7. Пример использования массива указателей на функции
2:
3:  #include <iostream.h>
4:
5:  void Square (int&,int&);
6:  void Cube (int&, int&);
7:  void Swap (int&, int &);
8:  void GetVals(int&, int&);
9:  void PrintVals(int, int);
10:
11: int main()
12: {
13:   int valOne=1, valTwo=2;
14:   int choice, i;
15:   const MaxArray = 5;
16:   void (*pFuncArray[MaxArray])(int&, int&);
17:
18:   for (i=0;i<MaxArray;i++)
19:   {
20:    cout << "(1)Change Values (2)Square (3)Cube (4)Swap: ";
21:    cin >> choice;
22:    switch (choice)
23:    {
24:      case 1:pFuncArray[i] = GetVals; break;
25:      case 2:pFuncArray[i] = Square; break;
26:      case 3:pFuncArray[i] = Cube; break;
27:      case 4:pFuncArray[i] = Swap; break;
28:      default:pFuncArray[i] = 0;
29:    }
30:   }
31:
32:   for (i=0;i<MaxArray; i++)
33:   {
34:    if ( pFuncArray[i] == 0 )
35:       continue;
36:    pFuncArray[i](valOne,valTwo);
37:    PrintVals(valOne,valTwo);
38:   }
39:  return 0;
40: }
41:
42: void PrintVals(int x, int y)
43: {
44:   cout << "x: " << x << " y: " << y << endl;
45: }
46:
47: void Square (int & rX, int & rY)
48: {
49:   rX *= rX;
```

```
50:    rY *= rY;
51:  }
52:
53:  void Cube (int & rX, int & rY)
54:  {
55:    int tmp;
56:
57:    tmp = rX;
58:    rX *= rX;
59:    rX = rX * tmp;
60:
61:    tmp = rY;
62:    rY *= rY;
63:    rY = rY * tmp;
64:  }
65:
66:  void Swap(int & rX, int & rY)
67:  {
68:    int temp;
69:    temp = rX;
70:    rX = rY;
71:    rY = temp;
72:  }
73:
74:  void GetVals (int & rValOne, int & rValTwo)
75:  {
76:    cout << "New value for ValOne: ";
77:    cin >> rValOne;
78:    cout << "New value for ValTwo: ";
79:    cin >> rValTwo;
80:  }
```

РЕЗУЛЬТАТ

```
(1)Change Values (2)Square (3)Cube (4)Swap: 1
(1)Change Values (2)Square (3)Cube (4)Swap: 2
(1)Change Values (2)Square (3)Cube (4)Swap: 3
(1)Change Values (2)Square (3)Cube (4)Swap: 4
(1)Change Values (2)Square (3)Cube (4)Swap: 2
New Value for ValOne: 2
New Value for ValTwo: 3
x: 2 y: 3
x: 4 y: 9
x: 64 y: 729
x: 729 y: 64
x: 531441 y:4096
```

АНАЛИЗ Как и в предыдущем листинге, для экономии места не были показаны выполнения объявленных функций, поскольку сами функции остались теми же, что и в листинге 14.5. В строке 16 объявляется массив `pFuncArray`, содержащий пять указателей на функции, которые возвращают `void` и принимают две ссылки на значения типа `int`.

В строках 18–30 пользователю предлагается установить последовательность вызова функций. Каждый член массива связывается с соответствующей функцией. Последовательный вызов функции осуществляется в строках 32–38, причем после каждого вызова на экран сразу выводится результат.

Передача указателей на функции в другие функции

Указатели на функции (или массивы указателей) могут передаваться в другие функции для вызова в них с помощью указателя нужной функции.

Листинг 14.5 можно усовершенствовать, передав указатель на выбранную функцию другой функции (кроме `main()`), которая выведет исходные значения на печать, вызовет функцию и вновь напечатает измененные значения. Именно такой подход применен в листинге 14.8.

Листинг 14.8. Передача указателя на функцию другой функции

```
1:  // Листинг 14.8. Передача указателя на функцию другой функции
2:
3:  #include <iostream.h>
4:
5:  void Square (int&,int&);
6:  void Cube (int&, int&);
7:  void Swap (int&, int &);
8:  void GetVals(int&, int&);
9:  void PrintVals(void (*)(int&, int&),int&, int&);
10:
11: int main()
12: {
13:   int valOne=1, valTwo=2;
14:   int choice;
15:   bool fQuit = false;
16:
17:   void (*pFunc)(int&, int&);
18:
19:   while (fQuit == false)
20:   {
21:    cout << "(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: ";
22:    cin >> choice;
23:    switch (choice)
24:    {
25:      case 1:pFunc = GetVals; break;
26:      case 2:pFunc = Square; break;
27:      case 3:pFunc = Cube; break;
28:      case 4:pFunc = Swap; break;
29:      default:fQuit = true; break;
30:    }
31:    if (fQuit == true)
32:      break;
33:    PrintVals ( pFunc, valOne, valTwo);
34:   }
35:
```

```
36:   return 0;
37:  }
38:
39:  void PrintVals( void (*pFunc)(int&, int&),int& x, int& y)
40:  {
41:    cout << "x: " << x << " y: " << y << endl;
42:    pFunc(x,y);
43:    cout << "x: " << x << " y: " << y << endl;
44:  }
45:
46:  void Square (int & rX, int & rY)
47:  {
48:    rX *= rX;
49:    rY *= rY;
50:  }
51:
52:  void Cube (int & rX, int & rY)
53:  {
54:    int tmp;
55:
56:    tmp = rX;
57:    rX *= rX;
58:    rX = rX * tmp;
59:
60:    tmp = rY;
61:    rY *= rY;
62:    rY = rY * tmp;
63:  }
64:
65:  void Swap(int & rX, int & rY)
66:  {
67:    int temp;
68:    temp = rX;
69:    rX = rY;
70:    rY = temp;
71:  }
72:
73:  void GetVals (int & rValOne, int & rValTwo)
74:  {
75:    cout << "New value for ValOne: ";
76:    cin >> rValOne;
77:    cin >> rValTwo;
78:    cin >> rValTwo;
79:  }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
New value for ValOne: 2
New value for ValTwo: 3
```

РЕЗУЛЬТАТ

```
x: 2 y: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y:64
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

АНАЛИЗ

В строке 17 объявляется указатель на функцию `pFunc`, принимающую две ссылки на `int` и возвращающую `void`. Функция `PrintVals`, для которой задается три параметра, объявляется в строке 9. Первым в списке параметров стоит указатель на функцию, возвращающую `void` и принимающую две ссылки на `int`. Второй и третий параметры функции `PrintVals` представляют собой ссылки на значения типа `int`. После того как пользователь выберет нужную функцию, в строке 33 происходит вызов функции `PrintVals`.

Спросите у знакомого программиста, работающего с C++, что означает следующее выражение:

```
void PrintVals(void (*)(int&, int&), int&, int&);
```

Это вид объявлений, который используется крайне редко и заставляет программистов обращаться к книгам каждый раз, когда нечто подобное встречается в тексте. Но временами данный подход позволяет значительно усовершенствовать код программы, как в нашем примере.

Использование typedef с указателями на функции

Конструкция `void (*)(int&, int&)` весьма громоздка. Для ее упрощения можно воспользоваться ключевым словом `typedef`, объявив новый тип (назовем его `VPF`) указателей на функции, возвращающие `void` и принимающие две ссылки на значения типа `int`. Листинг 14.9 представляет собой переписанную версию листинга 14.8 с использованием этого подхода.

Листинг 14.9. Использование оператора `typedef` для объявления типа указателей на функции

```
1:  // Листинг 14.9. Использование typedef для
2:  // объявления типа указателей на функции
3:  #include <iostream.h>
4:
5:  void Square (int&,int&);
6:  void Cube (int&, int&);
7:  void Swap (int&, int &);
8:  void GetVals(int&, int&);
9:  typedef void (*VPF) (int&, int&) ;
10:  void PrintVals(VPF,int&, int&);
11:
```

```
12: int main()
13: {
14: int valOne=1, valTwo=2;
15: int choice;
16: bool fQuit = false;
17:
18: VPF pFunc;
19:
20: while (fQuit == false)
21: {
22: cout << "(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: ";
23: cin >> choice;
24: switch (choice)
25: {
26: case 1:pFunc = GetVals; break;
27: case 2:pFunc = Square; break;
28: case 3:pFunc = Cube; break;
29: case 4:pFunc = Swap; break;
30: default:fQuit = true; break;
31: }
32: if (fQuit == true)
33: break;
34: PrintVals ( pFunc, valOne, valTwo);
35: }
36: return 0;
37: }
38:
39: void PrintVals( VPF pFunc,int& x, int& y)
40: {
41: cout << "x: " << x << " y: " << y << endl;
42: pFunc(x,y);
43: cout << "x: " << x << " y: " << y << endl;
44: }
45:
46: void Square (int & rX, int & rY)
47: {
48:    rX *= rX;
49:    rY *= rY;
50: }
51:
52: void Cube (int & rX, int & rY)
53: {
54:    int tmp;
55:
56:    tmp = rX;
57:    rX *= rX;
58:    rX = rX * tmp;
59:
60:    tmp = rY;
61:    rY *= rY;
```

```
62:    rY = rY * tmp;
63:  }
64:
65:  void Swap(int & rX, int & rY)
66:  {
67:    int temp;
68:    temp = rX;
69:    rX = rY;
70:    rY = temp;
71:  }
72:
73:  void GetVals (int & rValOne, int & rValTwo)
74:  {
75:    cout << "New value for ValOne: ";
76:    cin >> rValOne;
77:    cout << "New value for ValTwo: ";
78:    cin >> rValTwo;
79:  }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 1
x: 1 y: 2
New value for ValOne: 2
New value for ValTwo: 3
x: 2 y: 3
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 3
x: 2 y: 3
x: 8 y: 27
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 2
x: 8 y: 27
x: 64 y: 729
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 4
x: 64 y: 729
x: 729 y: 64
(0)Quit (1)Change Values (2)Square (3)Cube (4)Swap: 0
```

АНАЛИЗ

В строке 9 с помощью оператора `typedef` объявляется новый тип `VPF` как указатели на функции, возвращающие `void` и принимающие две ссылки на `int`.

В строке 10 объявляется функция `PrintVals()`, которая принимает три параметра: `VPF` и две ссылки на `integer`. В строке 18 указатель `Pfunc` объявляется как принадлежащий типу `VPF`.

После объявления типа `VPF` дальнейшее использование указателя `pFunc` и функции `PrintVals()` становится проще и понятнее. Информация, выводимая программой на экран, не изменилась.

Указатели на функции-члены

До настоящего времени все создаваемые указатели на функции использовались для общих функций, не принадлежащих к какому-нибудь одному классу. Однако разрешается создавать указатели и на функции, являющиеся членами классов (методы).

Для создания такого указателя используется тот же синтаксис, что и для указателя на обычную функцию, но с добавлением имени класса и оператора области видимости (::). Таким образом, объявление указателя pFunc на функции-члены класса Shape, принимающие два целочисленных параметра и возвращающие void, выглядит следующим образом:

```
void (Shape::*pFunc) (int,int);
```

Указатели на функции-члены используются так же, как и рассмотренные ранее указатели простых функции. Единственное отличие состоит в том, что для вызова функции необходимо наличие объекта соответствующего класса, для которого вызываются функции. В листинге 14.10 показано использование указателя на метод класса.

Листинг 14.10. Указатели на функции-члены

```
1:  //Листинг 14.10. Указатели на виртуальные функции-члены
2:
3:  #include <iostream.h>
4:
5:  class Mammal
6:  {
7:  public:
8:   Mammal():itsAge(1) {  }
9:   virtual ~Mammal() { }
10:    virtual void Speak() const = 0;
11:    virtual void Move() const = 0;
12: protected:
13:    int itsAge;
14: } ;
15:
16: class Dog : public Mammal
17: {
18: public:
19:    void Speak()const { cout << "Woof!\ n"; }
20:    void Move() const { cout << "Walking to heel...\ n"; }
21: } ;
22:
23:
24: class Cat : public Mammal
25: {
26: public:
27:    void Speak()const { cout << "Meow!\ n"; }
28:    void Move() const { cout << "slinking...\ n"; }
29: } ;
30:
31:
32: class Horse : public Mammal
33: {
34: public:
35:    void Speak()const { cout << "Whinny!\ n"; }
36:    void Move() const { cout << "Galloping...\ n"; }
37: } ;
38:
39:
```

```
40: int main()
41: {
42:   void (Mammal::*pFunc)() const =0;
43:   Mammal* ptr =0;
44:   int Animal;
45:   int Method;
46:   bool fQuit = false;
47:
48:   while (fQuit == false)
49:   {
50:    cout << "(0)Quit (1)dog (2)cat (3)horse: ";
51:    cin >> Animal;
52:    switch (Animal)
53:    {
54:      case 1: ptr = new Dog; break;
55:      case 2: ptr = new Cat; break;
56:      case 3: ptr = new Horse; break;
57:      default: fQuit = true; break;
58:    }
59:    if (fQuit)
60:      break;
61:
62:    cout << "(1)Speak (2)Move: ";
63:    cin >> Method;
64:    switch (Method)
65:    {
66:      case 1: pFunc = Mammal::Speak; break;
67:      default: pFunc = Mammal::Move; break;
68:    }
69:
70:    (ptr->*pFunc)();
71:    delete ptr;
72:   }
73:  return 0;
74: }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)dog (2)cat (3)horse: 1
(1)Speak (2)Move: 1
Woof!
(0)Quit (1)dog (2)cat (3)horse: 2
(1)Speak (2)Move: 1
Meow!
(0)Quit (1)dog (2)cat (3)horse: 3
(1)Speak (2)Move: 2
Galloping
(0)Quit (1)dog (2)cat (3)horse: 0
```

АНАЛИЗ В строках 4—14 объявляется тип абстрактных данных `Mammal` с двумя чистыми виртуальными методами `Speak()` и `Move()`. От класса `Mammal` производятся подклассы `Dog`, `Cat` и `Horse`, в каждом из которых замещаются соответствующим образом функции `Speak()` и `Move()`.

В процессе выполнения тела функции main() пользователю предлагается выбрать животное, после чего в области динамического обмена создается новый подкласс выбранного животного, адрес которого присваивается в строках 54–56 указателю ptr.

Затем пользователь выбирает метод, который связывается с указателем pFunc. В строке 70 выбранный метод вызывается для созданного объекта посредством предоставления доступа к объекту с помощью указателя ptr и к функции с помощью указателя pFunc.

Наконец, строкой 71 для указателя ptr вызывается функция delete, которая очищает область памяти, занятую созданным ранее объектом. Заметьте, что нет смысла вызывать delete для pFunc, поскольку последний является указателем на код, а не на объект в области памяти. Хотя даже при попытке сделать это вы получите сообщение об ошибке компиляции.

Массивы указателей на функции-члены

Аналогично указателям на обычные функции, указатели на функции-члены могут храниться в массиве. Для инициализации такого массива можно использовать адреса различных функций-членов. В таком случае, чтобы вызвать для объекта тот или иной метод, достаточно просто указать массив и индекс смещения. Именно такой подход применяется в листинге 14.11.

Листинг 14.11. Массив указателей на функции-члены

```
1:  //Листинг 14.11. Массивы указателей на функции-члены
2:
3:  #include <iostream.h>
4:
5:  class Dog
6:  {
7:  public:
8:   void Speak()const { cout << "Woof!\ n"; }
9:   void Move() const { cout << "Walking to heel...\ n"; }
10:   void Eat() const { cout << "Gobbling food...\ n"; }
11:   void Growl() const { cout << "Grrrrr\ n"; }
12:   void Whimper() const { cout << "Whining noises...\ n"; }
13:   void RollOver() const { cout << "Rolling over...\ n"; }
14:   void PlayDead() const { cout << "Is this the end of Little Caesar?\ n"; }
15: } ;
16:
17: typedef void (Dog::*PDF)()const ;
18: int main()
19: {
20:   const int MaxFuncs = 7;
21:   PDF DogFunctions[MaxFuncs] =
22:    {  Dog::Speak,
23:       Dog::Move,
24:       Dog::Eat,
25:       Dog::Growl,
26:       Dog::Whimper,
27:       Dog::RollOver,
28:       Dog::PlayDead } ;
```

```
29:
30:    Dog* pDog =0;
31:    int Method;
32:    bool fQuit = false;
33:
34:    while (!fQuit)
35:    {
36:     cout <<  "(0)Quit (1)Speak (2)Move (3)Eat (4)Growl";
37:     cout << " (5)Whimper (6)Roll Over (7)Play Dead: ";
38:     cin >> Method;
39:     if (Method == 0)
40:     {
41:       fQuit = true;
42:     }
43:     else
44:     {
45:       pDog = new Dog;
46:       (pDog->*DogFunctions[Method-1])();
47:       delete pDog;
48:     }
49:    }
50:   return 0;
51:  }
```

РЕЗУЛЬТАТ

```
 (0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over (7)Play
Dead: 1
Woof!
 (0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over (7)Play
Dead: 4
Grrr
 (0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over (7)Play
Dead: 7
Is this the end of Little Caesar?
 (0)Quit (1)Speak (2)Move (3)Eat (4)Growl (5)Whimper (6)Roll Over (7)Play
Dead: 0
```

АНАЛИЗ

В строках 5–15 создается класс `Dog`, содержащий семь функций-членов, характеризующихся одинаковыми сигнатурой и типом возврата. В строке 17 с помощью `typedef` объявляется тип `PDF` константных указателей на функции-члены `Dog`, которые не принимают и не возвращают никаких значений.

В строках 21–28 объявляется массив `DogFunctions`, предназначенный для хранения указателей на семь функций-членов.

В строках 36 и 37 пользователю предлагается выбрать метод. Выбор любого элемента, кроме `Quit`, приводит к созданию объекта класса `Dog`, после чего из массива вызывается соответствующий метод (строка 46). Ниже представлена еще одна строка, которая может немного смутить ваших знакомых программистов, работающих с C++:

```
(pDog->*-DogFunctions[Method-1])();
```

Это выражение, безусловно, немного экзотично, но с его помощью можно создать таблицу функций-членов, что сделает код программы проще и читабельнее.

Рекомендуется	Не рекомендуется
Используйте указатели на функции-члены для вызова методов в объектах класса. **Используйте** `typedef`, чтобы упростить объявление указателя на функцию-член.	**Не злоупотребляйте** созданием указателей на функции-члены, если без них можно обойтись.

Резюме

Сегодня вы познакомились с созданием статических переменных-членов класса, которые, в отличие от обычных переменных-членов, принадлежат всему классу, а не отдельному объекту. Если статическая переменная-член объявлена как `public`, то обратиться к ней можно просто по имени, даже не используя объектов класса, которому принадлежит эта переменная.

Статические переменные-члены можно использовать в качестве счетчиков объектов класса. Поскольку они не являются частями объектов, при их объявлении не происходит автоматическое резервирование памяти, как при объявлении обычных переменных-членов. Поэтому в программе за пределами объявления класса обязательно должна быть строка, в которой происходит определение и инициализация статической переменной-члена.

Статические функции-члены также принадлежат всему классу, подобно статическим переменным-членам. Вызвать статическую функцию-член класса можно даже в том случае, если не было создано ни одного объекта этого класса. Сами же эти функции могут использоваться для открытия доступа к статическим переменным-членам. Поскольку статические переменные-члены не имеют указателя `this`, они не могут использовать обычные переменные-члены.

Из-за отсутствия указателя `this` статические функции-члены не могут объявляться как `const`. Дело в том, что при объявлении функции-члена со спецификатором `const` устанавливается, что указатель `this` этой функции является константным.

Кроме того, вы узнали, как объявлять и использовать указатели на обычные функции и на функции-члены, а также познакомились с созданием массивов этих указателей и с передачей указателей на функции в другие функции.

Как указатели на функции, так и указатели на функции-члены могут использоваться для создания таблиц функций, что облегчает управление их вызовом в процессе выполнения программы. Это придает программе гибкость и делает программный код более читабельным.

Вопросы и ответы

Зачем использовать статические данные, если есть глобальные?

Область видимости статических данных ограничивается классом. Обращаться к статической переменной-члену следует из объектов класса, либо из внешнего кода программы, явно указав имя класса (в случае, если статическая переменная-член описана как `public`), либо с помощью открытой статической функции-члена этого класса. Статические переменные-члены относятся к типу данных того класса, которому они

принадлежат. Ограничение доступа к членам класса, вызванное строгим контролем за типом данных в C++, делает использование статических переменных-членов более безопасным по сравнению с глобальными данными.

Зачем использовать статические функции-члены, если можно воспользоваться глобальными функциями?

Статические функции-члены принадлежат классу и могут вызываться только с помощью объектов класса или с явным указанием имени класса, например: `ClassName::FunctionName()`.

Насколько часто в программах используются указатели на функции и указатели на функции-члены?

Такие указатели используются достаточно редко. Это дело вкуса программиста. Даже в сложных и мощных программах без них можно вполне обойтись.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Могут ли статические переменные-члены быть закрытыми?
2. Объявите статическую переменную-член.
3. Объявите статическую функцию.
4. Объявите указатель на функцию, принимающую параметр типа `int` и возвращающую значение типа `long`.
5. Измените указатель, созданный в задании 4, на указатель на функцию-член класса `Car`.
6. Объявите массив из десяти указателей, созданных в задании 5.

Упражнения

1. Напишите короткую программу, объявляющую класс с одной обычной переменной-членом и одной статической переменной-членом. Создайте конструктор, выполняющий инициализацию переменной-члена и приращение статической переменной-члена. Затем опишите деструктор, который уменьшает на единицу значение статической переменной.
2. Используя программный блок из упражнения 1, напишите короткую выполняемую программу, которая создает три объекта, а затем выводит значения их переменных-членов и статической переменной-члена класса. Затем последовательно удаляйте объекты и выводите на экран значение статической переменной-члена.

3. Измените программу упражнения 2 таким образом, чтобы доступ к статической переменной-члену осуществлялся с помощью статической функции-члена. Сделайте статическую переменную-член закрытой.
4. Создайте в программе упражнения 3 указатель на функцию-член для доступа к значению нестатической переменной-члена и воспользуйтесь им для вывода этих значений на печать.
5. Добавьте две дополнительные переменные-члена к классу из предыдущих заданий. Добавьте методы доступа, возвращающие значения всех этих переменных. Все функции-члены должны возвращать значения одинакового типа и иметь одинаковую сигнатуру. Для доступа к этим методам используйте указатель на функцию-член.

Подведение итогов

В этой главе вашему вниманию предлагается достаточно мощная программа, в которой используется большинство средств и подходов программирования, освоенных вами в течение двух недель.

В этой программе используются связанные списки, виртуальные функции, чистые виртуальные функции, замещения функций, полиморфизм, открытое наследование, перегрузка функций, вечные циклы, указатели, ссылки и многие другие знакомые вам средства. Обратите внимание, что представленный здесь связанный список отличается от рассмотренных ранее. Язык C++ предоставляет множество способов достижения одной и той же цели.

Цель данной программы состоит в создании функционального связанного списка. В узлах созданного списка можно хранить записи о деталях и агрегатах, что позволяет использовать его в реальных прикладных программах баз данных складов. Хотя здесь представлена не окончательная форма программы, она достаточно хорошо демонстрирует возможности создания совершенной структуры накопления и обработки данных. Листинг программы содержит 311 строк. Попробуйте самостоятельно проанализировать код, прежде чем прочтете анализ, приведенный после листинга.

Итоги второй недели

```
1:    // ********************************************
2:    //
3:    // Название:       Неделя 2. Подведение итогов
4:    //
5:    // Файл:       Week2
6:    //
7:    // Описание:   Демонстрация создания и использования связанного списка
8:    //
9:    // Классы:      PART - содержит идентификационный
10:    // номер детали и обеспечивает возможность
11:    // добавлять другие данные
12:    // PartNode - функционирует как узел в PartsList
13:    //
14:    // PartsList - реализует механизм связывания
15:    // узлов в список
16:    //
17:    // ************************************************
```

```
18:
19:    #include <iostream.h>
20:
21:
22:
23:    // *************** Part ************
24:
25:    // Абстрактный базовый класс, общий для всех деталей
26:    class Part
27:    {
28:    public:
29:       Part():itsPartNumber(1) { }
30:       Part(int PartNumber):itsPartNumber(PartNumber){ }
31:       virtual ~Part(){ } ;
32:       int GetPartNumber() const {  return itsPartNumber; }
33:       virtual void Display() const =0;  // должна быть замещена как private
34:    private:
35:       int itsPartNumber;
36:    } ;
37:
38:    // выполнение чистой виртуальной функции в
39:    // стандартном виде для всех производных классов
40:    void Part::Display() const
41:    {
42:        cout << "\ nНомер детали: " << itsPartNumber << endl;
43:    }
44:
45:    // ************* Автомобильные детали **********
46:
47:    class CarPart : public Part
48:    {
49:    public:
50:       CarPart():itsModelYear(94){ }
51:       CarPart(int year, int partNumber);
52:       virtual void Display() const
53:      {
54:         Part::Display(); cout << "Год создания: ";
55:         cout << itsModelYear << endl;
56:      }
57:    private:
58:       int itsModelYear;
59:    } ;
60:
61:    CarPart::CarPart(int year, int partNumber):
62:       itsModelYear(year),
63:       Part(partNumber)
64:    { }
65:
66:
67:    // ************* Авиационные детали **********
```

```
68:
69:     class AirPlanePart : public Part
70:     {
71:     public:
72:        AirPlanePart():itsEngineNumber(1){ } ;
73:        AirPlanePart(int EngineNumber, int PartNumber);
74:        virtual void Display() const
75:       {
76:          Part::Display(); cout << "Номер двигателя: ";
77:          cout << itsEngineNumber << endl;
78:       }
79:     private:
80:        int itsEngineNumber;
81:     } ;
82:
83:     AirPlanePart::AirPlanePart(int EngineNumber, int PartNumber):
84:        itsEngineNumber(EngineNumber),
85:        Part(PartNumber)
86:     { }
87:
88:     // ************** Узлы списка деталей **********
89:     class PartNode
90:     {
91:     public:
92:        PartNode (Part*);
93:        ~PartNode();
94:        void SetNext(PartNode * node) {  itsNext = node; }
95:        PartNode * GetNext() const;
96:        Part * GetPart() const;
97:     private:
98:        Part *itsPart;
99:        PartNode * itsNext;
100:    } ;
101:
102:    // Выполнение PartNode...
103:
104:    PartNode::PartNode(Part* pPart):
105:    itsPart(pPart),
106:    itsNext(0)
107:    { }
108:
109:    PartNode::~PartNode()
110:    {
111:       delete itsPart;
112:       itsPart = 0;
113:       delete itsNext;
114:       itsNext = 0;
115:    }
116:
117:    //Возвращается NULL, если нет следующего узла PartNode
```

```
118:    PartNode * PartNode::GetNext() const
119:    {
120:          return itsNext;
121:    }
122:
123:    Part * PartNode::GetPart() const
124:    {
125:       if (itsPart)
126:          return itsPart;
127:       else
128:          return NULL; // ошибка
129:    }
130:
131:    // *************** Список деталей ***********
132:    class PartsList
133:    {
134:    public:
135:       PartsList();
136:       ~PartsList();
137:    // Необходимо, чтобы конструктор-копировщик и оператор соответствовали друг другу!
138:       Part*      Find(int & position, int PartNumber)  const;
139:       int      GetCount() const {  return itsCount; }
140:       Part*      GetFirst() const;
141:       static     PartsList& GetGlobalPartsList()
142:     {
143:        return  GlobalPartsList;
144:     }
145:       void       Insert(Part *);
146:       void       Iterate(void (Part::*f)()const) const;
147:       Part*      operator[](int) const;
148:    private:
149:       PartNode * pHead;
150:       int itsCount;
151:       static PartsList GlobalPartsList;
152:    } ;
153:
154:    PartsList PartsList::GlobalPartsList;
155:
156:    // Выполнение списка...
157:
158:    PartsList::PartsList():
159:       pHead(0),
160:       itsCount(0)
161:       { }
162:
163:    PartsList::~PartsList()
164:    {
165:       delete pHead;
166:    }
167:
```

```
168:    Part*   PartsList::GetFirst() const
169:    {
170:       if (pHead)
171:          return pHead->GetPart();
172:       else
173:          return NULL;  // ловушка ошибок
174:    }
175:
176:    Part *  PartsList::operator[](int offSet) const
177:    {
178:       PartNode* pNode = pHead;
179:
180:       if (!pHead)
181:          return NULL; // ловушка ошибок
182:
183:       if (offSet > itsCount)
184:          return NULL; // ошибка
185:
186:       for (int i=0;i<offSet; i++)
187:          pNode = pNode->GetNext();
188:
189:      return   pNode->GetPart();
190:    }
191:
192:    Part*   PartsList::Find(int & position, int PartNumber)  const
193:    {
194:       PartNode * pNode = 0;
195:       for (pNode = pHead, position = 0;
196:             pNode!=NULL;
197:             pNode = pNode->GetNext(), position++)
198:       {
199:          if (pNode->GetPart()->GetPartNumber() == PartNumber)
200:             break;
201:       }
202:       if (pNode == NULL)
203:          return NULL;
204:       else
205:          return pNode->GetPart();
206:    }
207:
208:    void PartsList::Iterate(void (Part::*func)()const) const
209:    {
210:       if (!pHead)
211:          return;
212:       PartNode* pNode = pHead;
213:       do
214:          (pNode->GetPart()->*func)();
215:       while (pNode = pNode->GetNext());
216:    }
217:
```

```
218:    void PartsList::Insert(Part* pPart)
219:    {
220:       PartNode * pNode = new PartNode(pPart);
221:       PartNode * pCurrent = pHead;
222:       PartNode * pNext = 0;
223:
224:       int New =  pPart->GetPartNumber();
225:       int Next = 0;
226:       itsCount++;
227:
228:       if (!pHead)
229:       {
230:          pHead = pNode;
231:          return;
232:       }
233:
234:       // Если это значение меньше головного узла,
235:       // то текущий узел становится головным
236:       if (pHead->GetPart()->GetPartNumber() > New)
237:       {
238:          pNode->SetNext(pHead);
239:          pHead = pNode;
240:          return;
241:       }
242:
243:       for (;;)
244:       {
245:          // Если нет следующего, вставляется текущий
246:          if (!pCurrent->GetNext())
247:          {
248:             pCurrent->SetNext(pNode);
249:             return;
250:          }
251:
252:          // Если текущий больше предыдущего, но меньше следующего, то вставляем
253:           // здесь. Иначе присваиваем значение указателя Next
254:          pNext = pCurrent->GetNext();
255:          Next = pNext->GetPart()->GetPartNumber();
256:          if (Next > New)
257:          {
258:             pCurrent->SetNext(pNode);
259:             pNode->SetNext(pNext);
260:             return;
261:          }
262:          pCurrent = pNext;
263:       }
264:    }
265:
266:    int main()
267:    {
```

```
268:        PartsList&pl = PartsList::GetGlobalPartsList();
269:        Part * pPart = 0;
270:        int PartNumber;
271:        int value;
272:        int choice;
273:
274:        while (1)
275:        {
276:           cout << "(0)Quit (1)Car (2)Plane: ";
277:           cin >> choice;
278:
279:           if (!choice)
280:              break;
281:
282:           cout << "New PartNumber?: ";
283:           cin >>  PartNumber;
284:
285:           if (choice == 1)
286:           {
287:              cout << "Model Year?: ";
288:              cin >> value;
289:              pPart = new CarPart(value,PartNumber);
290:           }
291:           else
292:           {
293:              cout << "Engine Number?: ";
294:              cin >> value;
295:              pPart = new AirPlanePart(value,PartNumber);
296:           }
297:
298:           pl.Insert(pPart);
299:        }
300:        void (Part::*pFunc)()const = Part::Display;
301:        pl.Iterate(pFunc);
302:      return 0;
303:    }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2837
Model Year? 90
(0)Quit (1)Car (2)Plane: 2
New PartNumber?: 378
Engine Number?: 4938
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4499
Model Year? 94
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 3000
Model Year? 93
```

```
РЕЗУЛЬТАТ
(0)Quit (1)Car (2)Plane: 0

Part Number: 378
Engine No.: 4938

Part Number: 2837
Model Year: 90

Part Number: 3000
Model Year: 93

Part Number: 4499
Model Year: 94
```

Представленная программа создает связанный список объектов класса Part. Связанный список — это динамическая структура данных вроде массива, за исключением того, что в список можно добавлять произвольное число объектов указанного типа и удалять любой из введенных объектов.

Данный связанный список разработан для хранения объектов класса Part, где Part — абстрактный тип данных, служащий базовым классом для любого объекта с заданной переменной-членом itsPartNumber. В программе от класса Part производятся два подкласса — CarPart и AirPlanePart.

Класс Part описывается в строках 26–36, где задаются одна переменная-член и несколько методов доступа. Предполагается, что затем в объекты класса будет добавлена другая ценная информация и возможность контроля за числом созданных объектов в базе данных. Класс Part описывается как абстрактный тип данных, на что указывает чистая виртуальная функция Display().

Обратите внимание, что в строках 40–43 определяется выполнение чистой виртуальной функции Display(). Предполагается, что метод Display() будет замещаться в каждом производном классе, но в определении замещенного варианта допускается просто вызывать стандартный метод из базового класса.

Два простых производных класса, CarPart и AirPlanePart, описываются в строках 47–59 и 69–87 соответственно. В каждом из них замещается метод Display() простым обращением к методу Display() базового класса.

Класс PartNode выступает в качестве интерфейса между классами Part и PartList. Он содержит указатель на Part и указатель на следующий узел в списке. Методы этого класса только возвращают и устанавливают следующий узел в списке и возвращают соответствующий объект Part.

За "интеллектуальность" связанного списка полностью отвечает класс PartsList, описываемый в строках 132–152. Этот класс содержит указатель на головной узел списка (pHead) и, с его помощью продвигаясь по списку, получает доступ ко всем другим методам. Продвижение по списку означает запрашивание текущего узла об адресе следующего вплоть до обнаружения узла, указатель которого на следующий узел равен NULL.

Безусловно, в этом примере представлен упрощенный вид связанного списка. В реально используемой программе список должен обеспечивать еще больший доступ к первому и последнему узлам списка или создавать специальный объект итерации, с помощью которого клиенты смогут легко продвигаться по списку.

В то же время класс PartsList предлагает ряд интересных методов, упорядоченных по алфавиту. Зачастую такой подход весьма эффективен, поскольку упрощает поиск нужных функций.

Функция Find() принимает в качестве параметров PartNumber и значение int. Если найден раздел с указанным значением PartNumber, функция возвращает указатель на Part и порядковый номер этого раздела в списке. Если же раздел с номером PartNumber не обнаружен, функция возвращает значение NULL.

Функция GetCount() проходит по всем узлам списка и возвращает количество объектов в списке. В PartsList это значение записывается в переменную-член itsCount, хотя его можно легко вычислить, последовательно продвигаясь по списку.

Функция GetFirst() возвращает указатель на первый объект Part в списке или значение NULL, если список пустой.

Функция GetGlobalPartsList() возвращает ссылку на статическую переменную-член GlobalPartsList. Описание статической переменной GlobalPartsList является типичным решением для классов типа PartsList, хотя, безусловно, могут использоваться и другие имена. В законченном виде реализация этой идеи состоит в автоматическом изменении конструктора класса Part таким образом, чтобы каждому новому объекту класса присваивался номер с учетом текущего значения статической переменной GlobalPartsList.

Функция Insert принимает значение указателя на объект Part, создает для него PartNode и добавляет объект Part в список в порядке возрастания номеров PartNumber.

Функция Iterate принимает указатель на константную функцию-член класса Part без параметров, которая возвращает void. Эта функция вызывается для каждого объекта Part в списке. В описании класса Part таким характеристикам соответствует единственная функция Display(), замещенная во всех производных классах. Таким образом, будет вызываться вариант метода Display(), соответствующий типу объекта Part.

Функция Operator[] позволяет получить прямой доступ к объекту Part по заданному смещению. Этот метод обеспечивает простейший способ определения границ списка: если список нулевой, или заданное смещение больше числа объектов в списке, возвращается значение NULL, сигнализирующее об ошибке.

В реальной программе имело бы смысл все эти комментарии с описанием назначений функций привести в описании класса.

Тело функции main() представлено в строках 266–303. В строке 268 описывается ссылка на PartsList и инициализируется значением GlobalPartsList. Обратите внимание, что GlobalPartsList инициализируется в строке 154. Эта строка необходима, поскольку описание статической переменной-члена не сопровождается ее автоматическим определением. Поэтому определение статической переменной-члена должно выполняться за пределами описания класса.

В строках 274–299 пользователю предлагается указать, вводится ли деталь для машины или для самолета. В зависимости от выбора, запрашиваются дополнительные сведения и создается новый объект, который добавляется в список в строке 298.

Выполнение метода Insert() класса PartList показано в строках 218–264. При вводе идентификационного номера первой детали — 2837 — создается объект CarPart, который передается в LinkedList::Insert()с введенными номером детали и годом создания 90.

В строке 220 создается новый объект PartNode, принимающий значения новой детали. Переменная New инициализируется номером детали. Переменная-член itsCount класса PartsList увеличивается на единицу в строке 226.

В строке 228 проверяется равенство указателя pHead значению NULL. В данном случае возвращается значение TRUE, поскольку это первый узел списка и указатель pHead в нем нулевой. В результате в строке 230 указателю pHead присваивается адрес нового узла и функция возвращается.

Пользователю предлагается ввести следующую деталь. В нашем примере вводится деталь от самолета с идентификационным номером 37 и номером двигателя 4938. Сно-

ва вызывается функция PartsList::Insert() и pNode инициализируется новым узлом. Статическая переменная-член itsCount становится равной 2 и вновь проверяется pHead. Поскольку теперь pHead не равен нулю, то значение указателя больше не изменяется.

В строке 236 номер детали, указанный в головном узле, на который ссылается pHead (в нашем случае это 2837), сравнивается с номером новой детали — 378. Поскольку последний номер меньше, условное выражение в строке 236 возвращает TRUE и головным узлом в списке становится новый объект.

Строкой 238 указателю pNode присваивается адрес того узла, на который ссылался указатель pHead. Обратите внимание, что в следующий узел списка передается не новый объект, а тот, который был введен ранее. В строке 239 указателю pHead присваивается адрес нового узла.

На третьем цикле пользователь вводит деталь для автомобиля под номером 4499 с годом выпуска 94. Происходит очередное приращение счетчика и сравнивается номер текущего объекта с объектом головного узла. В этот раз новый введенный идентификационный номер детали оказывается больше номера объекта, определяемого в pHead, поэтому запускается цикл for в строке 243.

Значение идентификационного номера головного узла равно 378. Второй узел содержит объект со значением 2837. Текущее значение — 4499. Исходно указатель pCurrent связывается с головным узлом. Поэтому при обращении к переменной next объекта, на который указывает pCurrent, возвращается адрес второго узла. Следовательно, условное выражение в строке 246 возвратит False.

Указатель pCurrent устанавливается на следующий узел, и цикл повторяется. Теперь проверка в строке 246 приводит к положительному результату. Если следующего элемента нет, то новый узел вставляется в конец списка.

На четвертом цикле вводится номер детали 3000. Дальнейшее выполнение программы напоминает предыдущий этап, однако в этом случае текущий узел имеет номер 2837, а значение следующего узла равно 4499. Проверка в строке 256 возвращает TRUE, и новый узел вставляется между двумя существующими.

Когда пользователь вводит 0, условное выражение в строке 279 возвращает TRUE и цикл while(1) прерывается. В строке 300 функция-член Display() присваивается указателю на функции-члены pFunc. В профессиональной программе присвоение должно проходить динамически, основываясь на выборе пользователем.

Указатель функции-члена передается методу Iterate класса PartsList. В строке 208 метод Iterate() проверяет, не является ли список пустым. Затем в строках 213–215 последовательно с помощью указателя функции-члена вызываются из списка все объекты Part. В итоге для объекта Part вызывается соответствующий вариант метода Display(), в результате чего для разных объектов выводится разная информация.

Основные вопросы

Итак, две недели изучения C++ уже позади. Сейчас вы наверняка свободно ориентируетесь в некоторых достаточно сложных аспектах объектно-ориентированного программирования, включая инкапсуляцию и полиморфизм.

Что дальше

Последняя неделя начинается с изучения дополнительных возможностей наследования. Затем на занятии 16 вы изучите потоки, а на занятии 17 познакомитесь с одним замечательным дополнением стандартов C++ — пространствами имен. Занятие 18 посвящено анализу основ объектно-ориентированного программирования. В этот день внимание будет сконцентрировано не столько на синтаксисе языка, сколько на изучении концепций объектно-ориентированного программирования. На занятии 19 вы познакомитесь с использованием шаблонов, а на занятии 20 узнаете о методах отслеживания исключительных ситуаций и ошибок. Наконец, на последнем занятии будут раскрыты некоторые хитрости и секреты программирования на C++, что сделает вас настоящим гуру в этой области.

Дополнительные возможности наследования

До настоящего момента вы использовали одиночное и множественное наследование для создания относительно простых связей между классами. Сегодня вы узнаете:

- Что такое вложение и как его использовать
- Что такое делегирование и как его использовать
- Как выполнить один класс внутри другого
- Как использовать закрытое наследование

Вложение

Анализируя примеры, приведенные на предыдущих занятиях, вы, вероятно, заметили, что в классах допускается использование в переменных-членах объектов других классов. В этом случае программисты на C++ говорят, что внешний класс содержит внутренний. Так, класс `Employee` в качестве переменных-членов может содержать строковые объекты (с именем сотрудника) и объекты с целочисленными значениями (зарплатой и т.д.).

В листинге 15.1 представлен незавершенный, но весьма полезный класс `String`. Запуск такой программы не приведет к выводу каких-либо результатов, но она потребуется при написании других программ этого занятия.

Листинг 15.1. Класс `String`

```
1:   #include <iostream.h>
2:   #include <string.h>
3:
4:   class String
5:   {
6:    public:
7:      // конструкторы
8:      String();
9:      String(const char *const);
```

```
10:      String(const String &);
11:     ~String();
12:
13:     // перегруженные операторы
14:     char & operator[](int offset);
15:     char operator[](int offset) const;
16:     String operator+(const String&);
17:     void operator+=(const String&);
18:     String & operator= (const String &);
19:
20:     // Общие методы доступа
21:     int GetLen()const { return itsLen; }
22:     const char * GetString() const { return itsString; }
23:     // статический целочисленный счетчик ConstructorCount;
24:
25:   private:
26:     String (int);     // закрытый конструктор
27:     char * itsString;
28:     unsigned short itsLen;
29:
30: } ;
31:
32: // конструктор класса String по умолчанию создает строку длиной 0 байт
33: String::String()
34: {
35:   itsString = new char[1];
36:   itsString[0] = '\ 0';
37:   itsLen=0;
38:   // cout << "\ tDefault string constructor\ n";
39:   // ConstructorCount++;
40: }
41:
42: // закрытый конструктор, используемый только
43: // методами класса для создания новой строки
44: // указанного размера, заполненной нулями
45: String::String(int len)
46: {
47:   itsString = new char[len+1];
48:   for (int i = 0; i<=len; i++)
49:     itsString[i] = '\ 0';
50:   itsLen=len;
51:   // cout << "\ tString(int) constructor\ n";
52:   // ConstructorCount++;
53: }
54:
55: // Преобразует массив символов в строку
56: String::String(const char * const cString)
57: {
58:   itsLen = strlen(cString);
59:   itsString = new char[itsLen+1];
```

```
60:    for (int i = 0; i<itsLen; i++)
61:     itsString[i] = cString[i];
62:    itsString[itsLen]='\ 0';
63:    // cout << "\ tString(char*) constructor\ n";
64:    // ConstructorCount++;
65:  }
66:
67:  // конструктор-копировщик
68:  String::String (const String & rhs)
69:  {
70:    itsLen=rhs.GetLen();
71:    itsString = new char[itsLen+1];
72:    for (int i = 0; i<itsLen;i++)
73:     itsString[i] = rhs[i];
74:    itsString[itsLen] = '\ 0';
75:    // cout << "\ tString(String&) constructor\ n";
76:    // ConstructorCount++;
77:  }
78:
79:  // деструктор освобождает занятую память
80:  String::~String ()
81:  {
82:    delete [] itsString;
83:    itsLen = 0;
84:    // cout << "\ tString destructor\ n";
85:  }
86:
87:  // этот оператор освобождает память, а затем
88:  // копирует строку и размер
89:  String& String::operator=(const String & rhs)
90:  {
91:    if (this == &rhs)
92:     return *this;
93:    delete [] itsString;
94:    itsLen=rhs.GetLen();
95:    itsString = new char[itsLen+1];
96:    for (int i = 0; i<itsLen;i++)
97:     itsString[i] = rhs[i];
98:    itsString[itsLen] = '\ 0';
99:    return *this;
100:   // cout << "\ tString operator=\ n";
101:  }
102:
103:  // неконстантный оператор индексирования,
104:  // возвращает ссылку на символ, который можно
105:  // изменить
106:  char & String::operator[](int offset)
107:  {
108:   if (offset > itsLen)
109:     return itsString[itsLen-1];
```

```
110:   else
111:     return itsString[offset];
112:  }
113:
114:  // константный оператор индексирования,
115:  // используется для константных объектов (см. конструктор-копировщик!)
116:  char String::operator[](int offset) const
117:  {
118:   if (offset > itsLen)
119:     return itsString[itsLen-1];
120:   else
121:     return itsString[offset];
122:  }
123:
124:  // создает новую строку, добавляя текущую
125:  // строку к rhs
126:  String String::operator+(const String& rhs)
127:  {
128:   int totalLen = itsLen + rhs.GetLen();
129:   String temp(totalLen);
130:   int i, j;
131:   for (i = 0; i<itsLen; i++)
132:     temp[i] = itsString[i];
133:   for (j = 0; j<rhs.GetLen(); j++, i++)
134:     temp[i] = rhs[j];
135:   temp[totalLen]='\ 0';
136:   return temp;
137:  }
138:
139:  // изменяет текущую строку, ничего не возвращая
140:  void String::operator+=(const String& rhs)
141:  {
142:   unsigned short rhsLen = rhs.GetLen();
143:   unsigned short totalLen = itsLen + rhsLen;
144:   String temp(totalLen);
145:   int i, j;
146:   for (i = 0; i<itsLen; i++)
147:     temp[i] = itsString[i];
148:   for (j = 0; j<rhs.GetLen(); j++, i++)
149:     temp[i] = rhs[i-itsLen];
150:   temp[totalLen]='\ 0';
151:   *this = temp;
152:  }
153:
154:  // int String::ConstructorCount = 0;
```

РЕЗУЛЬТАТ Нет

Представленный в листинге 15.1 класс String напоминает другой класс, использованный в листинге 12.12. Однако есть одно важное отличие между этими двумя классами: конструкторы и некоторые функции листинга 12.12 включали операторы вывода на печать, благодаря которым на экране отображались сообщения об их использовании. В листинге 15.1 эти операторы временно заблокированы, но они будут использоваться в следующих примерах.

Статическая переменная-член ConstructorCount объявляется и инициализируется соответственно в строках 23 и 154. Значение этой переменной увеличивается на единицу при вызове любого конструктора класса String. Эти функции также заблокированы и будут использоваться в следующих листингах.

В листинге 15.2 объявляется класс Employee, содержащий три объекта класса String.

Листинг 15.2. Класс Employee

```
1:  #include "String.hpp"
2:
3:  class Employee
4:  {
5:
6:  public:
7:   Employee();
8:   Employee(char *, char *, char *, long);
9:   ~Employee();
10:   Employee(const Employee&);
11:   Employee & operator= (const Employee &);
12:
13:   const String & GetFirstName() const
14:    { return itsFirstName; }
15:   const String & GetLastName() const { return itsLastName; }
16:   const String & GetAddress() const { return itsAddress; }
17:   long GetSalary() const { return itsSalary; }
18:
19:   void SetFirstName(const String & fName)
20:     { itsFirstName = fName; }
21:   void SetLastName(const String & lName)
22:    { itsLastName = lName; }
23:   void SetAddress(const String & address)
24:     { itsAddress = address; }
25:   void SetSalary(long salary) { itsSalary = salary; }
26: private:
27:   String  itsFirstName;
28:   String  itsLastName;
29:   String  itsAddress;
30:   long    itsSalary;
31: } ;
32:
33: Employee::Employee():
34:   itsFirstName(""),
35:   itsLastName(""),
36:   itsAddress(""),
```

```
37:    itsSalary(0)
38:  { }
39:
40:  Employee::Employee(char * firstName, char * lastName,
41:    char * address, long salary):
42:    itsFirstName(firstName),
43:    itsLastName(lastName),
44:    itsAddress(address),
45:    itsSalary(salary)
46:  { }
47:
48:  Employee::Employee(const Employee & rhs):
49:    itsFirstName(rhs.GetFirstName()),
50:    itsLastName(rhs.GetLastName()),
51:    itsAddress(rhs.GetAddress()),
52:    itsSalary(rhs.GetSalary())
53:  { }
54:
55:  Employee::~Employee() { }
56:
57:  Employee & Employee::operator= (const Employee & rhs)
58:  {
59:    if (this == &rhs)
60:     return *this;
61:
62:    itsFirstName = rhs.GetFirstName();
63:    itsLastName = rhs.GetLastName();
64:    itsAddress = rhs.GetAddress();
65:    itsSalary = rhs.GetSalary();
66:
67:    return *this;
68:  }
69:
70:  int main()
71:  {
72: Employee Edie("Jane","Doe","1461 Shore Parkway", 20000);
73:    Edie.SetSalary(50000);
74:    String LastName("Levine");
75:    Edie.SetLastName(LastName);
76:    Edie.SetFirstName("Edythe");
77:
78:    cout << "Имя: ";
79:    cout << Edie.GetFirstName().GetString();
80:    cout << " " << Edie.GetLastName().GetString();
81:    cout << ".\ nАдрес: ";
82:    cout << Edie.GetAddress().GetString();
83:    cout << ".\ nЗарплата: " ;
84:    cout << Edie.GetSalary();
85:    return 0;
86:  }
```

Сохраните листинг 15.1 в файле с именем `String.hpp`. Затем всякий раз, когда понадобится класс `String`, вы сможете вставить листинг 15.1, просто добавив строку `#include "String.hpp"`. Это первая строка в листинге 15.2.

```
Name: Edythe Levine.
Address: 1461 Shore Parkway.
Salary: 50000
```

В листинге 15.2 объявляется класс `Employee`, переменными-членами которого выступают три объекта класса `String` — `itsFirstName`, `itsLastName` и `itsAddress`.

В строке 72 создается объект `Employee`, который инициализируется четырьмя значениями. В строке 73 вызывается метод доступа `SetSalary` класса `Employee`, который принимает константное значение `50000`. В реальной программе это значение определялось бы либо динамически в процессе выполнения программы, либо устанавливалось бы константой.

В строке 74 создается и инициализируется строковой константой объект класса `String`, который в строке 75 используется в качестве аргумента функции `SetLastName()`.

В строке 76 вызывается метод `SetFirstName` класса `Employee` с еще одной строковой константой в качестве параметра. Однако если вы обратитесь к объявлению класса `Employee`, то увидите, что в нем нет функции `SetFirstName()`, принимающей строку символов как аргумент. Для функции `SetFirstName()` в качестве параметра задана константная ссылка на объект `String`. Тем не менее компилятор не покажет сообщения об ошибке, поскольку в строке 9 листинга 15.1 объявлен конструктор, создающий объект `String` из строковой константы.

Доступ к членам вложенного класса

В классе `Employee` не объявлены специальные методы доступа к переменным-членам класса `String`. Если объект `Edie` класса `Employee` попытается напрямую обратиться к переменной-члену `itsLen`, содержащейся в его собственной переменной-члене `itsFirstName`, это приведет к возникновению ошибки компиляции. Однако в таком обращении нет необходимости. Методы доступа класса `Employee` просто создают интерфейс для класса `String`, и классу `Employee` нет нужды беспокоиться о деталях выполнения класса `String`, а также о том, каким образом собственная целочисленная переменная-член `itsSalary` хранит свое значение.

Фильтрация доступа к вложенным классам

Вы, наверное, уже заметили, что в классе `String` перегружается `operator+`. В классе `Employee` доступ к перегруженной функции `operator+` заблокирован. Дело в том, что в объявлениях методов доступа класса `Employee` указано, что все эти методы, такие как `GetFirstName()`, возвращают константные ссылки. Поскольку функция `operator+` не является (и не может быть) константой (она изменяет объект, для которого вызывается), попытка написать следующую строку приведет к сообщению об ошибке компиляции:

```
String buffer = Edie.GetFirstName() + Edie.GetLastName();
```

Функция GetFirstName() возвращает константную строку и вы не можете использовать operator+ с константным объектом.

Чтобы устранить эту проблему, следует перегрузить функцию GetFirstName() таким образом, чтобы она стала неконстантной:

```
const String & GetFirstName() const { return itsFirstName; }
String & GetFirstName() { return itsFirstName; }
```

Как видите, возвращаемое значение больше не является константой, также как и сама функция-член. Изменения возвращаемого значения недостаточно для перегрузки имени функции. Необходимо изменить константность самой функции.

Цена вложений

Важно отметить, что пользователю придется "расплачиваться" за каждый объект внешнего класса всякий раз при создании или копировании объекта Employee.

Снимите символы комментариев с операторов cout листинга 15.1 (строки 38, 51, 63, 75, 84 и 100), и вы увидите, как часто они вызываются. В листинге 15.3 представлена та же программа, что и в листинге 15.2, только в этом примере добавлены операторы печати, которые будут показывать сообщения на экране всякий раз при выполнении конструктора класса Employee. Это позволит наглядно увидеть весь процесс создания объектов в программе.

До компиляции этого листинга разблокируйте строки 38, 51, 63, 75, 84 и 100 в листинге 15.1.

Листинг 15.3. Конструкторы вложенных классов

```
1:   #include "String.hpp"
2:
3:   class Employee
4:   {
5:
6:   public:
7:    Employee();
8:    Employee(char *, char *, char *, long);
9:    ~Employee();
10:   Employee(const Employee&);
11:   Employee & operator= (const Employee &);
12:
13:   const String & GetFirstName() const
14:    { return itsFirstName; }
15:   const String & GetLastName() const { return itsLastName; }
16:   const String & GetAddress() const { return itsAddress; }
17:   long GetSalary() const { return itsSalary; }
18:
19:   void SetFirstName(const String & fName)
20:     { itsFirstName = fName; }
```

```
21:    void SetLastName(const String & lName)
22:     { itsLastName = lName; }
23:    void SetAddress(const String & address)
24:      { itsAddress = address; }
25:    void SetSalary(long salary) { itsSalary = salary; }
26:  private:
27:    String  itsFirstName;
28:    String  itsLastName;
29:    String  itsAddress;
30:    long   itsSalary;
31:  } ;
32:
33:  Employee::Employee():
34:    itsFirstName(""),
35:    itsLastName(""),
36:    itsAddress(""),
37:    itsSalary(0)
38:  { }
39:
40:  Employee::Employee(char * firstName, char * lastName,
41:    char * address, long salary):
42:    itsFirstName(firstName),
43:    itsLastName(lastName),
44:    itsAddress(address),
45:    itsSalary(salary)
46:  { }
47:
48:  Employee::Employee(const Employee & rhs):
49:    itsFirstName(rhs.GetFirstName()),
50:    itsLastName(rhs.GetLastName()),
51:    itsAddress(rhs.GetAddress()),
52:    itsSalary(rhs.GetSalary())
53:  { }
54:
55:  Employee::~Employee() { }
56:
57:  Employee & Employee::operator= (const Employee & rhs)
58:  {
59:    if (this == &rhs)
60:     return *this;
61:
62:    itsFirstName = rhs.GetFirstName();
63:    itsLastName = rhs.GetLastName();
64:    itsAddress = rhs.GetAddress();
65:    itsSalary = rhs.GetSalary();
66:
67:    return *this;
68:  }
69:
70:   int main()
```

```
71:   {
72:    cout << "Creating Edie...\ n";
73:    Employee Edie("Jane","Doe","1461 Shore Parkway", 20000);
74:    Edie.SetSalary(20000);
75:    cout << "Calling SetFirstName with char *...\ n";
76:    Edie.SetFirstName("Edythe");
77:    cout << "Creating temporary string LastName...\ n";
78:    String LastName("Levine");
79:    Edie.SetLastName(LastName);
80:
81:    cout << "Name: ";
82:    cout << Edie.GetFirstName().GetString();
83:    cout << " " << Edie.GetLastName().GetString();
84:    cout << "\ nAddress: ";
85:    cout << Edie.GetAddress().GetString();
86:    cout << "\ nSalary: " ;
87:    cout << Edie.GetSalary();
88:    cout << endl;
89:   return 0;
90:  }
```

РЕЗУЛЬТАТ

```
1:  Creating Edie...
2:       String(char*) constructor
3:       String(char*) constructor
4:       String(char*) constructor
5:  Calling SetFirstName with char *...
6:       String(char*) constructor
7:       String destructor
8:  Creating temporary string LastName...
9:       String(char*) constructor
10: Name: Edythe Levine
11: Address: 1461 Shore Parkway
12: Salary: 20000
13:      String destructor
14:      String destructor
15:      String destructor
16:      String destructor
```

АНАЛИЗ В листинге 15.3 используются классы, объявленные ранее в листингах 15.1 и 15.2. Единственное отличие состоит в том, что операторы `cout` разблокированы. Чтобы упростить обсуждение, строки, выводимые программой на экран, были пронумерованы.

В строке 72 листинга 15.3 выводится сообщение `Creating Edie...`, которому соответствует первая строка вывода. Для создания объекта `Edie` класса `Employee` задаются четыре параметра. Для инициализации трех из них задействуются конструкторы класса `String`, о чем свидетельствуют три следующие строки вывода.

Строка 75 информирует о вызове функции `SetFirstName`. Следующая строка программы, `Edie.SetFirstName("Edythe")`, создает временный объект класса `String` из строковой константы `"Edythe"`, для чего вновь задействуются соответствующие конструкто-

ры класса String (см. 6-ю строку вывода). Обратите внимание, что этот временный объект уничтожается сразу же после присвоения его значения переменной-члену, о чем свидетельствует вызов деструктора класса String (см. 7-ю строку вывода).

Присвоив имя, программа приступает к присвоению фамилии служащего. Это можно было бы выполнить так же, как и в случае с присвоением имени с помощью автоматически создаваемого временного объекта класса String. Но чтобы показать все возможности, в строке 78 явно создается объект класса String. Конструктор, создающий этот объект, дал о себе знать 9-й строкой вывода. Деструктор не вызывается, поскольку этот объект не удаляется до тех пор, пока не выйдет за границы своей области видимости в конце функции.

Наконец программа выводит на экран персональные сведения о служащем и выходит за область видимости объекта Employee, в результате чего вызываются четыре деструктора класса для удаления объектов этого класса, вложенных в объект Employee, и созданного ранее временного объекта LastName.

Передача объекта как значения

В листинге 15.3 показано, как создание одного объекта Employee приводит к вызову пяти конструкторов класса String. Листинг 15.4 — это еще один переписанный вариант программы. В нем нет дополнительных операторов вывода помимо представленных в листинге 15.1 (сейчас они разблокированы) и используется статическая переменная-член ConstructorCount, объявленная в классе String.

Как следует из объявления в листинге 15.1, значение переменной ConstructorCount увеличивается на единицу при каждом вызове конструктора класса String. В конце программы, представленной в листинге 15.4, объект Employee передается на печать сначала как ссылка, а затем как значение. Статическая переменная-член ConstructorCount отслеживает, сколько объектов класса String создается при разных способах передачи объекта Employee как параметра функции.

Перед компиляцией этого листинга в программе листинга 15.1 дополнительно к тем строкам, которые были разблокированы для листинга 15.3, следует снять символы комментариев со строк 23, 39, 52, 64, 76 и 153.

Листинг 15.4. Передача объекта как значения

```
1:   #include "String.hpp"
2:
3:   class Employee
4:   {
5:
6:   public:
7:    Employee();
8:    Employee(char *, char *, char *, long);
9:    ~Employee();
10:    Employee(const Employee&);
11:    Employee & operator= (const Employee &);
12:
13:    const String & GetFirstName() const
14:     { return itsFirstName; }
15:    const String & GetLastName() const { return itsLastName; }
```

```
16:    const String & GetAddress() const { return itsAddress; }
17:    long GetSalary() const { return itsSalary; }
18:
19:    void SetFirstName(const String & fName)
20:      { itsFirstName = fName; }
21:    void SetLastName(const String & lName)
22:     { itsLastName = lName; }
23:    void SetAddress(const String & address)
24:      { itsAddress = address; }
25:    void SetSalary(long salary) { itsSalary = salary; }
26:  private:
27:    String  itsFirstName;
28:    String  itsLastName;
29:    String  itsAddress;
30:    long   itsSalary;
31:  } ;
32:
33:  Employee::Employee():
34:    itsFirstName(""),
35:    itsLastName(""),
36:    itsAddress(""),
37:    itsSalary(0)
38:  { }
39:            .
40:  Employee::Employee(char * firstName, char * lastName,
41:    char * address, long salary):
42:    itsFirstName(firstName),
43:    itsLastName(lastName),
44:    itsAddress(address),
45:    itsSalary(salary)
46:  { }
47:
48:  Employee::Employee(const Employee & rhs):
49:    itsFirstName(rhs.GetFirstName()),
50:    itsLastName(rhs.GetLastName()),
51:    itsAddress(rhs.GetAddress()),
52:    itsSalary(rhs.GetSalary())
53:  { }
54:
55:  Employee::~Employee() { }
56:
57:  Employee & Employee::operator= (const Employee & rhs)
58:  {
59:    if (this == &rhs)
60:     return *this;
61:
62:    itsFirstName = rhs.GetFirstName();
63:    itsLastName = rhs.GetLastName();
64:    itsAddress = rhs.GetAddress();
65:    itsSalary = rhs.GetSalary();
```

```
66:
67:    return *this;
68:  }
69:
70:   void PrintFunc(Employee);
71:   void rPrintFunc(const Employee&);
72:
73:   int main()
74:   {
75:    Employee Edie("Jane","Doe","1461 Shore Parkway", 20000);
76:    Edie.SetSalary(20000);
77:    Edie.SetFirstName("Edythe");
78:    String LastName("Levine");
79:    Edie.SetLastName(LastName);
80:
81:    cout << "Constructor count: " ;
82:    cout << String::ConstructorCount << endl;
83:    rPrintFunc(Edie);
84:    cout << "Constructor count: ";
85:    cout << String::ConstructorCount << endl;
86:    PrintFunc(Edie);
87:    cout << "Constructor count: ";
88:    cout << String::ConstructorCount << endl;
89:   return 0;
90:  }
91:  void PrintFunc (Employee Edie)
92:  {
93:
94:    cout << "Name: ";
95:    cout << Edie.GetFirstName().GetString();
96:    cout << " " << Edie.GetLastName().GetString();
97:    cout << ".\ nAddress: ";
98:    cout << Edie.GetAddress().GetString();
99:    cout << ".\ nSalary: " ;
100:    cout << Edie.GetSalary();
101:    cout << endl;
102:
103:  }
104:
105:  void rPrintFunc (const Employee& Edie)
106:  {
107:    cout << "Name: ";
108:    cout << Edie.GetFirstName().GetString();
109:    cout << " " << Edie.GetLastName().GetString();
110:    cout << "\ nAddress: ";
111:    cout << Edie.GetAddress().GetString();
112:    cout << "\ nSalary: " ;
113:    cout << Edie.GetSalary();
114:    cout << endl;
115:  }
```

```
РЕЗУЛЬТАТ String(char*) constructor
    String(char*) constructor
    String(char*) constructor
    String(char*) constructor
    String destructor
    String(char*) constructor
Constructor count: 5
Name: Edythe Levine
Address: 1461 Shore Parkway
Salary: 20000
Constructor count: 5
    String(String&) constructor
    String(String&) constructor
    String(String&) constructor
Name: Edythe Levine.
Address: 1461 Shore Parkway.
Salary: 20000
    String destructor
    String destructor
    String destructor
Constructor count: 8
String destructor
    String destructor
    String destructor
    String destructor
```

АНАЛИЗ Как видно по данным, выводимым программой, в процессе создания одного объекта `Employee` создается пять объектов класса `String`. Когда объект `Employee` передается в функцию `rPrintFunc()` как ссылка, дополнительные объекты `Employee` не создаются. Соответственно не создаются и дополнительные объекты `String`. (Все они, кстати, также автоматически передаются как ссылки.)

Когда объект `Employee` передается в функцию `PrintFunc()` как значение, создается копия объекта `Employee` вместе с тремя объектами класса `String` (для этого используется конструктор-копировщик).

Различные пути передачи функциональности классу

В некоторых случаях одному классу необходимо передать некоторые свойства другого. Предположим, например, что вам необходимо создать класс каталога деталей `PartsCatalog`. На основе класса `PartsCatalog` предполагается создать коллекцию объектов, представляющую различные запчасти с уникальными номерами. В базе данных на основе класса `PartsCatalog` запрещается дублирование объектов, а для доступа к объекту необходимо указать его идентификационный номер.

Ранее, в обзоре за вторую неделю, уже был объявлен и детально проанализирован класс `PartsList`. Чтобы не начинать работу с нуля, можно взять этот класс за основу при объявлении класса `PartsCatalog`. Для этого можно вложить класс `PartsList` в класс `PartsCatalog`, чтобы затем делегировать классу `PartsCatalog` ответственность за поддержание связанного списка в класс `PartsList`.

Существует альтернативный путь. Можно произвести класс PartsCatalog от класса PartsList, таким образом унаследовав все свойства последнего. Помните однако, что открытое наследование (со спецификатором public) предполагает логическую принадлежность производного класса более общему базовому классу. Действительно ли в нашем случае класс PartsCatalog является частным проявлением класса PartList? Чтобы разобраться в этом, попробуйте ответить на ряд вопросов.

1. Содержит ли базовый класс PartsList методы, не применимые в классе PartsCatalog? Если да, то, вероятно, от открытого наследования лучше отказаться.
2. Будет ли один объект класса PartsCatalog соответствовать одному объекту класса PartsList? Если для создания объекта требуется не менее двух объектов PartsList, то, безусловно, необходимо применять вложение.
3. Обеспечит ли наследование от базового класса преимущества в работе благодаря использованию виртуальных функций или методов доступа к защищенным членам базового класса? В случае положительного ответа имеет смысл воспользоваться открытым или закрытым наследованием.

Ответив на приведенные выше вопросы, вы должны принять решение, использовать ли вам в программе открытое наследование, закрытое наследование (см. далее в этом занятии) или вложение. Познакомимся с некоторыми терминами, которые потребуются нам при дальнейшем обсуждении этой темы.

- *Вложение* — объект, относящийся к другому классу, используется в текущий класс.
- *Делегирование* — передача ответственности за выполнение специальных функций вложенному классу.
- *Выполнение средствами класса* — реализация специальных функций в классе за счет другого класса без использования открытого наследования.

Делегирование

Почему же класс PartsCatalog нельзя произвести от PartsList? Дело в том, что класс PartsCatalog должен обладать совершенно иными свойствами и его невозможно представить как частную реализацию класса PartsList. Посмотрите, класс PartsList — это коллекция объектов, упорядоченная по возрастанию номеров, элементы которой могут повторяться. Класс PartsCatalog представляет неупорядоченную коллекцию уникальных объектов.

Конечно, при желании можно произвести класс PartsList от класса PartsList со спецификатором public, после чего соответствующим образом заместить функцию Insert() и оператор индексирования ([]). Однако такие действия крайне нелогичны и противоречат самой сути наследования. Вместо этого следует создать новый класс PartsCatalog, в котором нет оператора индексирования, не разрешается дублирование записей и перегружается operator+ для суммирования наборов записей. Функцию управления связанным списком оставим классу PartsList.

Попробуем сначала решить эту задачу путем вложения одного класса в другой с делегированием ответственности от класса классу, как показано в листинге 15.5.

Листинг 15.5. Делегирование ответственности классу PartsList, **включенному в класс** PartsCatalog

```
1:  #include <iostream.h>
2:
3:  // *************** Класс Part ************
4:
5:  // Абстрактный базовый класс всех деталей
6:  class Part
7:  {
8:  public:
9:    Part():itsPartNumber(1) { }
10:   Part(int PartNumber):
11:     itsPartNumber(PartNumber){ }
12:   virtual ~Part(){ }
13:   int GetPartNumber() const
14:     { return itsPartNumber; }
15:   virtual void Display() const =0;
16: private:
17:   int itsPartNumber;
18: } ;
19:
20: // выполнение чистой виртуальной функции в
21: // стандартном виде для всех производных классов
22: void Part::Display() const
23: {
24: cout << "\ nPart Number: " << itsPartNumber << endl;
25: }
26:
27: // ************ Автомобильные детали **********
28:
29: class CarPart : public Part
30: {
31: public:
32:   CarPart():itsModelYear(94){ }
33:   CarPart(int year, int partNumber);
34:   virtual void Display() const
35:   {
36:     Part::Display();
37:     cout << "Model Year: ";
38:     cout << itsModelYear << endl;
39:   }
40: private:
41:   int itsModelYear;
42: } ;
43:
44: CarPart::CarPart(int year, int partNumber):
45:   itsModelYear(year),
46:   Part(partNumber)
47: { }
```

```
48:
49:
50:  // ************* Авиационные детали **********
51:
52:  class AirPlanePart : public Part
53:  {
54:  public:
55:    AirPlanePart():itsEngineNumber(1){ } ;
56:    AirPlanePart
57:      (int EngineNumber, int PartNumber);
58:    virtual void Display() const
59:    {
60:      Part::Display();
61:      cout << " Engine No.: ";
62:      cout << itsEngineNumber << endl;
63:    }
64:  private:
65:    int itsEngineNumber;
66:  } ;
67:
68:  AirPlanePart::AirPlanePart
69:    (int EngineNumber, int PartNumber):
70:    itsEngineNumber(EngineNumber),
71:    Part(PartNumber)
72:  { }
73:
74:  // ************** Узлы списка деталей *********
75:  class PartNode
76:  {
77:  public:
78:    PartNode (Part*);
79:    ~PartNode();
80:    void SetNext(PartNode * node)
81:     { itsNext = node; }
82:    PartNode * GetNext() const;
83:    Part * GetPart() const;
84:  private:
85:    Part *itsPart;
86:    PartNode * itsNext;
87:  } ;
88:   // Выполнение PartNode...
89:
90:   PartNode::PartNode(Part* pPart):
91:   itsPart(pPart),
92:   itsNext(0)
93:   { }
94:
95:   PartNode::~PartNode()
96:   {
97:    delete itsPart;
```

```
98:     itsPart = 0;
99:     delete itsNext;
100:     itsNext = 0;
101:    }
102:
103:    // Возвращается NULL, если нет следующего узла PartNode
104:    PartNode * PartNode::GetNext() const
105:    {
106:       return itsNext;
107:    }
108:
109:    Part * PartNode::GetPart() const
110:    {
111:     if (itsPart)
112:       return itsPart;
113:     else
114:       return NULL; //ошибка
115:    }
116:
117:
118:
119:   // *************** Список деталей ***********
120:   class PartsList
121:   {
122:   public:
123:     PartsList();
124:     ~PartsList();
125:     // необходимо, чтобы конструктор-копировщик и оператор соответствовали друг
          другу!
126:     void   Iterate(void (Part::*f)()const) const;
127:     Part*  Find(int & position, int PartNumber) const;
128:     Part*  GetFirst() const;
129:     void   Insert(Part *);
130:     Part*  operator[](int) const;
131:     int  GetCount() const { return itsCount;}
132:     static  PartsList& GetGlobalPartsList()
133:     {
134:       return GlobalPartsList;
135:     }
136:   private:
137:     PartNode * pHead;
138:     int itsCount;
139:     static PartsList GlobalPartsList;
140:   } ;
141:
142:   PartsList PartsList::GlobalPartsList;
143:
144:
145:    PartsList::PartsList():
146:     pHead(0),
```

```
147:     itsCount(0)
148:     { }
149:
150:    PartsList::~PartsList()
151:    {
152:     delete pHead;
153:    }
154:
155:    Part*  PartsList::GetFirst() const
156:    {
157:     if (pHead)
158:       return pHead->GetPart();
159:     else
160:       return NULL; // ловушка ошибок
161:    }
162:
163:    Part * PartsList::operator[](int offSet) const
164:    {
165:     PartNode* pNode = pHead;
166:
167:     if (!pHead)
168:       return NULL; // ловушка ошибок
169:
170:     if (offSet > itsCount)
171:       return NULL; // ошибка
172:
173:     for (int i=0;i<offSet; i++)
174:       pNode = pNode->GetNext();
175:
176:     return  pNode->GetPart();
177:    }
178:
179:    Part*  PartsList::Find(
180:      int & position,
181:      int PartNumber) const
182:    {
183:     PartNode * pNode = 0;
184:     for (pNode = pHead, position = 0;
185:        pNode!=NULL;
186:        pNode = pNode->GetNext(), position++)
187:     {
188:       if (pNode->GetPart()->GetPartNumber()== PartNumber)
189:        break;
190:     }
191:     if (pNode == NULL)
192:       return NULL;
193:     else
194:       return pNode->GetPart();
195:    }
196:
```

```
197:    void PartsList::Iterate(void (Part::*func)()const) const
198:    {
199:     if (!pHead)
200:       return;
201:     PartNode* pNode = pHead;
202:     do
203:       (pNode->GetPart()->*func)();
204:     while (pNode = pNode->GetNext());
205:    }
206:
207:    void PartsList::Insert(Part* pPart)
208:    {
209:     PartNode * pNode = new PartNode(pPart);
210:     PartNode * pCurrent = pHead;
211:     PartNode * pNext = 0;
212:
213:     int New = pPart->GetPartNumber();
214:     int Next = 0;
215:     itsCount++;
216:
217:     if (!pHead)
218:     {
219:       pHead = pNode;
220:       return;
221:     }
222:
223:     // если это значение меньше головного узла,
224:     // то текущий узел становится головным
225:     if (pHead->GetPart()->GetPartNumber() > New)
226:     {
227:       pNode->SetNext(pHead);
228:       pHead = pNode;
229:       return;
230:     }
231:
232:     for (;;)
233:     {
234:       // если нет следующего, вставляется текущий
235:       if (!pCurrent->GetNext())
236:       {
237:        pCurrent->SetNext(pNode);
238:        return;
239:       }
240:
241:       // если текущий больше предыдущего, но меньше следующего, то вставляем
242:       // здесь. Иначе присваиваем значение указателя Next
243:       pNext = pCurrent->GetNext();
244:       Next = pNext->GetPart()-> GetPartNumber();
245:       if (Next > New)
246:       {
```

```
247:        pCurrent->SetNext(pNode);
248:        pNode->SetNext(pNext);
249:        return;
250:       }
251:       pCurrent = pNext;
252:     }
253:    }
254:
255:
256:
257:   class PartsCatalog
258:   {
259:   public:
260:     void Insert(Part *);
261:     int Exists(int PartNumber);
262:     Part * Get(int PartNumber);
263:     operator+(const PartsCatalog &);
264:     void ShowAll() { thePartsList.Iterate(Part::Display); }
265:   private:
266:     PartsList thePartsList;
267:   } ;
268:
269:   void PartsCatalog::Insert(Part * newPart)
270:   {
271:     int partNumber = newPart->GetPartNumber();
272:     int offset;
273:
274:     if (!thePartsList.Find(offset, partNumber))
275:
276:      thePartsList.Insert(newPart);
277:     else
278:     {
279:      cout << partNumber << " был ";
280:      switch (offset)
281:      {
282:       case 0: cout << "first "; break;
283:       case 1: cout << "second "; break;
284:       case 2: cout << "third "; break;
285:       default: cout << offset+1 << "th ";
286:     }
287:     cout << "entry. Rejected!\ n";
288:     }
289:   }
290:
291:   int PartsCatalog::Exists(int PartNumber)
292:   {
293:     int offset;
294:     thePartsList.Find(offset,PartNumber);
295:      return offset;
296:   }
```

```
297:
298:   Part * PartsCatalog::Get(int PartNumber)
299:   {
300:     int offset;
301:     Part * thePart = thePartsList.Find(offset, PartNumber);
302:     return thePart;
303:   }
304:
305:
306:   int main()
307:   {
308:     PartsCatalog pc;
309:     Part * pPart = 0;
310:     int PartNumber;
311:     int value;
312:     int choice;
313:
314:     while (1)
315:     {
316:     cout << "(0)Quit (1)Car (2)Plane: ";
317:     cin >> choice;
318:
319:     if (!choice)
320:       break;
321:
322:     cout << "New PartNumber?: ";
323:     cin >> PartNumber;
324:
325:     if (choice == 1)
326:     {
327:       cout << "Model Year?: ";
328:       cin >> value;
329:       pPart = new CarPart(value,PartNumber);
330:     }
331:     else
332:     {
333:       cout << "Engine Number?: ";
334:       cin >> value;
335:       pPart = new AirPlanePart(value,PartNumber);
336:     }
337:     pc.Insert(pPart);
338:    }
339:    pc.ShowAll();
340:   return 0;
341:  }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4434
Model Year?: 93
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
1234 was the first entry. Rejected!
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2345
Model Year?: 93
(0)Quit (1)Car (2)Plane: 0

Part Number: 1234
Model Year: 94

Part Number: 2345
Model Year: 93

Part Number: 4434
Model Year: 93
```

ПРИМЕЧАНИЕ

Некоторые компиляторы не смогут откомпилировать строку 264, хотя она вполне соответствует стандартам C++. Если ваш компилятор возразит против записи этой строки, замените ее строкой

```
264:     void ShowAll() { thePartsList.Iterate(&Part::Display); }
```

(Обратите внимание на добавление амперсанта (знак &) перед `Part:Display`.) Если это сработает, свяжитесь с фирмой, поставившей вам этот компилятор, и поинтересуйтесь, где они его "откопали".

АНАЛИЗ

В листинге 15.5 используются классы `Part`, `PartNode` и `PartsList`, с которыми вы уже познакомились при подведении итогов второй недели.

Новый класс `PartsCatalog` объявляется в строках 257–267. Он использует `PartsList` как свою переменную-член, которой делегирует управление списком. Другими словами, класс `PartsCatalog` выполняется средствами классе `PartsList`.

Обратите внимание, что клиенты класса `PartsCatalog` не имеет прямого доступа к классу `PartsList`. Интерфейс класса `PartsList` реализуется методами класса `PartsCatalog`, что существенно изменяет его поведение. Например, метод `PartsCatalog::Insert()` не позволяет дублировать данные, вводимые в `PartsList`.

Определение выполнения функции `PartsCatalog::Insert()` начинается в строке 269. У объекта `Part`, передаваемого как параметр, запрашивается значение его переменной-члена `itsPartNumber`.

Это значение передается методу `Find()` класса `PartsList`, и объект добавляется в список, если только в списке не найден другой объект с таким же номером. В противном случае возвращается сообщение об ошибке.

Обратите внимание, что в методе `Insert()` класса `PartCatolog` используется переменная-член этого класса `thePartList`, являющаяся объектом класса `PartList`. Процедура поддержания связного списка и добавления объектов в него, а также поиска и возвращения данных из списка полностью реализуется вложенным классом `PartsList`, объект которого является переменной-членом класса `PartsCatalog`. Вместо того чтобы повторять все процедуры обработки записей списка в классе `PartsCatalog`, методами этого класса просто создается удобный интерфейс для уже существующего класса `PartsList`.

Именно в этом и состоит суть модульности программирования на C++. Удачно созданный однажды модуль, такой как PartsLists, можно многократно использовать в других программах, например с классом PartsCatalog. При этом разработчиков нового класса PartsCatalog могут совершенно не интересовать детали выполнения модуля PartsList. Интерфейс класса PartsList (в данном случае под интерфейсом понимается его объявление) предоставляет всю информацию, необходимую разработчику нового класса PartsCatalog.

Закрытое наследование

Если бы для PartsCatalog был необходим доступ к защищенным членам PartsList (в данном примере таковых нет) или в PartsCatalog использовались замещенные методы PartsList, то его можно было бы просто унаследовать от PartsList.

Однако, поскольку PartsCatalog не является объектом PartsList и нежелательно предоставлять весь набор функциональных возможностей PartsList клиентам PartsCatalof, следует применить закрытое наследование.

Первое, что необходимо знать: при закрытом наследовании все переменные и функции-члены базового класса трактуются так, как если бы они были объявлены закрытыми, независимо от установок доступа в базовом классе. Таким образом, для любой функции, не являющейся функцией-членом PartsCatalog, недоступны функции, унаследованные из PartsList. Это очень важно: закрытое наследование не передает в производный класс интерфейс базового класса.

Класс PartsList невидим для клиентов класса PartsCatalog. Поэтому последним недоступен интерфейс класса PartsList и они не могут вызывать его методы. Однако пользователям будут доступны все методы класса PartsCatalog, имеющие доступ ко всем членам класса PartsList, так как класс PartsCatalog является производным от PartList. Важно также то, что объекты PartsCatalog не являются объектами PartsList, как было бы при использовании открытого наследования. Класс PartsCatalog выполняется методами класса PartsList, как в случае с вложением. Применение закрытого наследования не менее удобно.

Использование закрытого наследования показано в листинге 15.6. Класс PartsCatalog производится как private от класса PartsList.

Листинг 15.6. Закрытое наследование

```
1:   //Листинг 15.6. Закрытое наследование
2:   #include <iostream.h>
3:
4:   // ***************** Класс Part ************
5:
6:   // Абстрактный базовый класс всех деталей
7:   class Part
8:   {
9:   public:
10:     Part():itsPartNumber(1) { }
11:     Part(int PartNumber):
12:       itsPartNumber(PartNumber){ }
13:     virtual ~Part(){ }
14:     int GetPartNumber() const
```

```
15:       { return itsPartNumber; }
16:     virtual void Display() const =0;
17:   private:
18:     int itsPartNumber;
19:   } ;
20:
21:   // выполнение чистой виртуальной функции в
22:   // стандартном виде для всех производных классов
23:   void Part::Display() const
24:   {
25:     cout << "\ nPart Number: " << itsPartNumber << endl;
26:   }
27:
28:   // *************** Car Part ************
29:
30:   class CarPart : public Part
31:   {
32:   public:
33:     CarPart():itsModelYear(94){ }
34:     CarPart(int year, int partNumber);
35:     virtual void Display() const
36:     {
37:       Part::Display();
38:       cout << "Model Year: ";
39:       cout << itsModelYear << endl;
40:     }
41:   private:
42:     int itsModelYear;
43:   } ;
44:
45:   CarPart::CarPart(int year, int partNumber):
46:     itsModelYear(year),
47:     Part(partNumber)
48:   { }
49:
50:
51:   // *********** Класс AirPlane Part ************
52:
53:   class AirPlanePart : public Part
54:   {
55:   public:
56:     AirPlanePart():itsEngineNumber(1){ } ;
57:     AirPlanePart
58:       (int EngineNumber, int PartNumber);
59:     virtual void Display() const
60:     {
61:       Part::Display();
62:       cout << "Engine No.: ";
63:       cout << itsEngineNumber << endl;
64:     }
```

```
65:   private:
66:     int itsEngineNumber;
67:   } ;
68:
69:   AirPlanePart::AirPlanePart
70:     (int EngineNumber, int PartNumber):
71:     itsEngineNumber(EngineNumber),
72:     Part(PartNumber)
73:   { }
74:
75:   // ************ Класс Part Node ************
76:   class PartNode
77:   {
78:   public:
79:     PartNode (Part*);
80:     ~PartNode();
81:     void SetNext(PartNode * node)
82:      { itsNext = node; }
83:     PartNode * GetNext() const;
84:     Part * GetPart() const;
85:   private:
86:     Part *itsPart;
87:     PartNode * itsNext;
88:   } ;
89:    // Выполнения PartNode...
90:
91:    PartNode::PartNode(Part* pPart):
92:    itsPart(pPart),
93:    itsNext(0)
94:    { }
95:
96:    PartNode::~PartNode()
97:    {
98:     delete itsPart;
99:     itsPart = 0;
100:     delete itsNext;
101:     itsNext = 0;
102:    }
103:
104:    // Возвращает NULL NULL, если нет следующего узла PartNode
105:    PartNode * PartNode::GetNext() const
106:    {
107:       return itsNext;
108:    }
109:
110:    Part * PartNode::GetPart() const
111:    {
112:     if (itsPart)
113:       return itsPart;
114:     else
```

```
115:       return NULL; //ошибка
116:     }
117:
118:
119:
120:   // ************ Класс Part List ************
121:   class PartsList
122:   {
123:   public:
124:     PartsList();
125:     ~PartsList();
126:     // Необходимо, чтобы конструктор-копировщик и оператор соответствовали друг
          другу!
127:     void   Iterate(void (Part::*f)()const) const;
128:     Part*  Find(int & position, int PartNumber) const;
129:     Part*  GetFirst() const;
130:     void   Insert(Part *);
131:     Part*  operator[](int) const;
132:     int  GetCount() const { return itsCount; }
133:     static  PartsList& GetGlobalPartsList()
134:     {
135:       return GlobalPartsList;
136:     }
137:   private:
138:     PartNode * pHead;
139:     int itsCount;
140:     static PartsList GlobalPartsList;
141:   } ;
142:
143:   PartsList PartsList::GlobalPartsList;
144:
145:
146:    PartsList::PartsList():
147:     pHead(0),
148:     itsCount(0)
149:     { }
150:
151:    PartsList::~PartsList()
152:    {
153:     delete pHead;
154:    }
155:
156:    Part*  PartsList::GetFirst() const
157:    {
158:     if (pHead)
159:       return pHead->GetPart();
160:     else
161:       return NULL; // ловушка ошибок
162:    }
163:
```

```
164:    Part * PartsList::operator[](int offSet) const
165:    {
166:     PartNode* pNode = pHead;
167:
168:     if (!pHead)
169:       return NULL; // ловушка ошибок
170:
171:     if (offSet > itsCount)
172:       return NULL; // ошибка
173:
174:     for (int i=0;i<offSet; i++)
175:       pNode = pNode->GetNext();
176:
177:     return  pNode->GetPart();
178:    }
179:
180:    Part*  PartsList::Find(
181:      int & position,
182:      int PartNumber) const
183:    {
184:     PartNode * pNode = 0;
185:     for (pNode = pHead, position = 0;
186:        pNode!=NULL;
187:        pNode = pNode->GetNext(), position++)
188:     {
189:       if (pNode->GetPart()->GetPartNumber() == PartNumber)
190:        break;
191:     }
192:     if (pNode == NULL)
193:       return NULL;
194:     else
195:       return pNode->GetPart();
196:    }
197:
198:    void PartsList::Iterate(void (Part::*func)()const) const
199:    {
200:     if (!pHead)
201:       return;
202:     PartNode* pNode = pHead;
203:     do
204:       (pNode->GetPart()->*func)();
205:     while (pNode = pNode->GetNext());
206:    }
207:
208:    void PartsList::Insert(Part* pPart)
209:    {
210:     PartNode * pNode = new PartNode(pPart);
211:     PartNode * pCurrent = pHead;
212:     PartNode * pNext = 0;
213:
```

```
214:     int New = pPart->GetPartNumber();
215:     int Next = 0;
216:     itsCount++;
217:
218:     if (!pHead)
219:     {
220:       pHead = pNode;
221:       return;
222:     }
223:
224:     // если это значение меньше головного узла,
225:     // то текущий узел становится головным
226:     if (pHead->GetPart()->GetPartNumber() > New)
227:     {
228:       pNode->SetNext(pHead);
229:       pHead = pNode;
230:       return;
231:     }
232:
233:     for (;;)
234:     {
235:       // если нет следующего, вставляется текущий
236:       if (!pCurrent->GetNext())
237:       {
238:        pCurrent->SetNext(pNode);
239:        return;
240:       }
241:
242:       // если текущий больше предыдущего, но меньше следующего, то вставляем
243:       // здесь. Иначе присваиваем значение указателя Next
244:       pNext = pCurrent->GetNext();
245:       Next = pNext->GetPart()->GetPartNumber();
246:       if (Next > New)
247:       {
248:        pCurrent->SetNext(pNode);
249:        pNode->SetNext(pNext);
250:        return;
251:       }
252:       pCurrent = pNext;
253:     }
254:   }
255:
256:
257:
258:   class PartsCatalog : private PartsList
259:   {
260:   public:
261:    void Insert(Part *);
262:    int Exists(int PartNumber);
263:    Part * Get(int PartNumber);
```

```
264:    operator+(const PartsCatalog &);
265:    void ShowAll() { Iterate(Part::Display); }
266:  private:
267:  } ;
268:
269:  void PartsCatalog::Insert(Part * newPart)
270:  {
271:    int partNumber = newPart->GetPartNumber();
272:    int offset;
273:
274:    if (!Find(offset, partNumber))
275:     PartsList::Insert(newPart);
276:    else
277:    {
278:     cout << partNumber << " was the ";
279:     switch (offset)
280:     {
281:       case 0: cout << "first "; break;
282:       case 1: cout << "second "; break;
283:       case 2: cout << "third "; break;
284:       default: cout << offset+1 << "th ";
285:     }
286:     cout << "entry. Rejected!\ n";
287:    }
288:  }
289:
290:  int PartsCatalog::Exists(int PartNumber)
291:  {
292:    int offset;
293:    Find(offset,PartNumber);
294:    return offset;
295:  }
296:
297:  Part * PartsCatalog::Get(int PartNumber)
298:  {
299:    int offset;
300:    return (Find(offset, PartNumber));
301:
302:  }
303:
304:  int main()
305:  {
306:    PartsCatalog pc;
307:    Part * pPart = 0;
308:    int PartNumber;
309:    int value;
310:    int choice;
311:
312:    while (1)
313:    {
```

```
314:     cout << "(0)Quit (1)Car (2)Plane: ";
315:     cin >> choice;
316:
317:     if (!choice)
318:       break;
319:
320:     cout << "New PartNumber?: ";
321:     cin >> PartNumber;
322:
323:     if (choice == 1)
324:     {
325:       cout << "Model Year?: ";
326:       cin >> value;
327:       pPart = new CarPart(value,PartNumber);
328:     }
329:     else
330:     {
331:       cout << "Engine Number?: ";
332:       cin >> value;
333:       pPart = new AirPlanePart(value,PartNumber);
334:     }
335:     pc.Insert(pPart);
336:    }
337:    pc.ShowAll();
338:   return 0;
339:  }
```

Результат

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4434
Model Year?: 93
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
1234 was the first entry. Rejected!
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2345
Model Year?: 93
(0)Quit (1)Car (2)Plane: 0

Part Number: 1234
Model Year: 94

Part Number: 2345
Model Year: 93

Part Number: 4434
Model Year: 93
```

АНАЛИЗ В листинге 15.6 был изменен интерфейс класса PartsCatalog и переписана функция main(). Интерфейсы других классов остались такими же, как и в листинге 15.5.

В строке 258 листинга 15.6 класс PartsCatalog производится как private от класса PartsList. Интерфейс класса PartsCatalog остался таким же, как и в листинге 15.5, хотя, конечно же, необходимость в объектах класса PartsList как переменных-членах отпала.

Функция-член ShowAll() класса PartsCatalog вызывает функцию Iterate() из PartsList, параметром которой задается указатель на функцию-член класса Part. Таким образом, функция ShowAll() выполняет роль открытого интерфейса, позволяя пользователям получать информацию, не обращаясь напрямую к закрытой функции Iterate(), прямой доступ к которой закрыт для клиентов класса PartsCatalog.

Функция Insert()тоже изменилась. Обратите внимание, в строке 274 функция Find() теперь вызывается непосредственно, поскольку она наследуется из базового класса. Чтобы при вызове функции Insert() не возникло зацикливания функции на самое себя, в строке 275 делается явный вызов функции с указанием имени класса.

Таким образом, если методам класса PartsCatalog необходимо вызвать методы PartsList, они могут делать это напрямую. Единственное исключение состоит в том, что при необходимости заместить метод базового класса в классе PartsList следует явно указать класс и имя функции.

Закрытое наследование позволяет PartsCatalog унаследовать функциональность базового класса и создавать интерфейс, косвенно открывающий доступ к его методам, которые нельзя вызывать напрямую.

Рекомендуется

Применяйте открытое наследование, когда производный класс является разновидностью базового.

Используйте вложение классов, когда необходимо делегировать выполнение задач другому классу, ограничив при этом доступ к его защищенным членам.

Применяйте закрытое наследование, если необходимо реализовать один класс в пределах другого и обеспечить доступ к защищенным членам базового класса.

Не рекомендуется

Не применяйте закрытое наследование, если необходимо использовать более одного объекта базового класса. Для этих целей больше подойдет вложение классов. Например, если для одного объекта PartsCatalog необходимы два объекта PartsList, вы не сможете использовать закрытое наследование.

Не используйте открытое наследование, если необходимо закрыть клиентам производного класса прямой доступ к методам базового класса.

Классы-друзья

Иногда для выполнения задач, поставленных перед программой, необходимо обеспечить взаимодействие нескольких независимых классов. Например, классы PartNode и PartsList тесно взаимосвязаны, и было бы удобно, если бы в PartsList можно было напрямую использовать указатель itsPart класса PartNode.

Конечно, можно было бы объявить itsPart как открытую или хотя бы защищенную переменную-член, но это далеко не лучший путь, противоречащий самой идее использования классов. Поскольку указатель itsPart является специфическим членом класса PartNode, его следует оставить недоступным для внешних классов.

Однако, если вы хотите предоставить данные или закрытые методы какому-либо иному классу, достаточно объявить этот класс другом. Это расширит интерфейс вашего класса возможностями класса-друга.

После того как в PartsNode класс PartsList будет объявлен другом, переменные-члены и методы класса PartsNode станут доступными для PartsList.

Важно заметить, что дружественность класса не передается на другие классы. Иными словами, если вы мой друг, а Ваня — ваш друг, это вовсе не значит, что Ваня также и мой друг. Кроме того, дружба не наследуется. Опять же, хотя вы мой друг и я хочу рассказать вам свои секреты, это не означает, что я желаю поделиться ими с вашими детьми.

Наконец, дружественность классов односторонняя. Объявление одного класса другом какого-либо иного класса не делает последний другом первого. Вы при желании может поделиться своими секретами со мной, но это не значит, что я должен рассказать вам свои секреты.

В листинге 15.7 представлена версия листинга 15.6, в которой используется объявление класса друга. Так, класс PartsList объявляется как друг класса PartNode. Еще раз напомним, что это объявление не делает класс PartNode другом класса PartsList.

Листинг 15.7. Использование классов-друзей

```
1:  #include <iostream.h>
2:
3:
4:
5:
6:  // *************** Класс Part ************
7:
8:  // Абстрактный базовый класс всех деталей
9:  class Part
10:   {
11:   public:
12:    Part():itsPartNumber(1) { }
13:    Part(int PartNumber):
14:      itsPartNumber(PartNumber){ }
15:    virtual ~Part(){ }
16:    int GetPartNumber() const
17:      { return itsPartNumber; }
18:    virtual void Display() const =0;
19:   private:
20:    int itsPartNumber;
21:   } ;
22:
23:   // выполнение чистой виртуальной функции в
24:   // стандартном виде для всех производных классов
25:   void Part::Display() const
26:   {
27:     cout << "\ nPart Number: ";
28:     cout << itsPartNumber << endl;
29:   }
30:
31:   // ************** Класс Car Part ************
32:
```

```
33:    class CarPart : public Part
34:    {
35:    public:
36:     CarPart():itsModelYear(94){ }
37:     CarPart(int year, int partNumber);
38:     virtual void Display() const
39:     {
40:       Part::Display();
41:       cout << "Model Year: ";
42:       cout << itsModelYear << endl;
43:     }
44:    private:
45:     int itsModelYear;
46:    } ;
47:
48:    CarPart::CarPart(int year, int partNumber):
49:     itsModelYear(year),
50:     Part(partNumber)
51:    { }
52:
53:
54:    // ********** Класс AirPlane Part **********
55:
56:    class AirPlanePart : public Part
57:    {
58:    public:
59:     AirPlanePart():itsEngineNumber(1){ } ;
60:     AirPlanePart
61:       (int EngineNumber, int PartNumber);
62:     virtual void Display() const
63:     {
64:       Part::Display();
65:       cout << "Engine No.: ";
66:       cout << itsEngineNumber << endl;
67:     }
68:    private:
69:     int itsEngineNumber;
70:    } ;
71:
72:    AirPlanePart::AirPlanePart
73:      (int EngineNumber, int PartNumber):
74:     itsEngineNumber(EngineNumber),
75:     Part(PartNumber)
76:    { }
77:
78:   // *************** Класс Part Node ***********
79:   class PartNode
80:   {
81:   public:
82:    friend class PartsList;
```

```
83:    PartNode (Part*);
84:    ~PartNode();
85:     void SetNext(PartNode * node)
86:      { itsNext = node; }
87:    PartNode * GetNext() const;
88:    Part * GetPart() const;
89:   private:
90:    Part *itsPart;
91:    PartNode * itsNext;
92:   } ;
93:
94:
95:     PartNode::PartNode(Part* pPart):
96:     itsPart(pPart),
97:     itsNext(0)
98:     { }
99:
100:     PartNode::~PartNode()
101:     {
102:       delete itsPart;
103:       itsPart = 0;
104:       delete itsNext;
105:       itsNext = 0;
106:      }
107:
108:      // Возвращается NULL, если нет следующего узла PartNode
109:      PartNode * PartNode::GetNext() const
110:      {
111:         return itsNext;
112:      }
113:
114:      Part * PartNode::GetPart() const
115:      {
116:       if (itsPart)
117:         return itsPart;
118:       else
119:         return NULL; //ошибка
120:      }
121:
122:
123:   // ************** Класс Part List ************
124:   class PartsList
125:   {
126:   public:
127:    PartsList();
128:    ~PartsList();
129:    // Необходимо, чтобы конструктор-копировщик и оператор соответствовали друг дру-
гу!
130:    void   Iterate(void (Part::*f)()const) const;
131:    Part*  Find(int & position, int PartNumber) const;
```

```
132:    Part*  GetFirst() const;
133:    void    Insert(Part *);
134:    Part*  operator[](int) const;
135:    int  GetCount() const { return itsCount; }
136:    static  PartsList& GetGlobalPartsList()
137:        {
138:          return GlobalPartsList;
139:        }
140:   private:
141:    PartNode * pHead;
142:    int itsCount;
143:    static PartsList GlobalPartsList;
144:   } ;
145:
146:   PartsList PartsList::GlobalPartsList;
147:
148:   // Implementations for Lists...
149:
150:   PartsList::PartsList():
151:    pHead(0),
152:    itsCount(0)
153:    { }
154:
155:   PartsList::~PartsList()
156:   {
157:    delete pHead;
158:   }
159:
160:   Part*  PartsList::GetFirst() const
161:   {
162:    if (pHead)
163:      return pHead->itsPart;
164:    else
165:      return NULL; // ловушка ошибок
166:   }
167:
168:   Part * PartsList::operator[](int offSet) const
169:   {
170:    PartNode* pNode = pHead;
171:
172:    if (!pHead)
173:      return NULL; // ловушка ошибок
174:
175:    if (offSet > itsCount)
176:      return NULL; // ошибка
177:
178:    for (int i=0;i<offSet; i++)
179:      pNode = pNode->itsNext;
180:
181:    return  pNode->itsPart;
```

```
182:  }
183:
184:  Part* PartsList::Find(int & position, int PartNumber) const
185:  {
186:   PartNode * pNode = 0;
187:   for (pNode = pHead, position = 0;
188:      pNode!=NULL;
189:      pNode = pNode->itsNext, position++)
190:   {
191:     if (pNode->itsPart->GetPartNumber() == PartNumber)
192:      break;
193:   }
194:   if (pNode == NULL)
195:     return NULL;
196:   else
197:     return pNode->itsPart;
198:  }
199:
200:  void PartsList::Iterate(void (Part::*func)()const) const
201:  {
202:   if (!pHead)
203:     return;
204:   PartNode* pNode = pHead;
205:   do
206:     (pNode->itsPart->*func)();
207:   while (pNode = pNode->itsNext);
208:  }
209:
210:  void PartsList::Insert(Part* pPart)
211:  {
212:   PartNode * pNode = new PartNode(pPart);
213:   PartNode * pCurrent = pHead;
214:   PartNode * pNext = 0;
215:
216:   int New = pPart->GetPartNumber();
217:   int Next = 0;
218:   itsCount++;
219:
220:   if (!pHead)
221:   {
222:     pHead = pNode;
223:     return;
224:   }
225:
226:   // если это значение меньше головного узла,
227:   // то текущий узел становится головным
228:   if (pHead->itsPart->GetPartNumber() > New)
229:   {
230:     pNode->itsNext = pHead;
231:     pHead = pNode;
```

```
232:       return;
233:     }
234:
235:     for (;;)
236:     {
237:       // если нет следующего, вставляется текущий
238:       if (!pCurrent->itsNext)
239:       {
240:        pCurrent->itsNext = pNode;
241:        return;
242:       }
243:
244:       // если текущий больше предыдущего, но меньше следующего, то вставляем
245:       // здесь. Иначе присваиваем значение указателя Next
246:       pNext = pCurrent->itsNext;
247:       Next = pNext->itsPart->GetPartNumber();
248:       if (Next > New)
249:       {
250:        pCurrent->itsNext = pNode;
251:        pNode->itsNext = pNext;
252:        return;
253:       }
254:       pCurrent = pNext;
255:     }
256:   }
257:
258:   class PartsCatalog : private PartsList
259:   {
260:   public:
261:    void Insert(Part *);
262:    int Exists(int PartNumber);
263:    Part * Get(int PartNumber);
264:    operator+(const PartsCatalog &);
265:    void ShowAll() { Iterate(Part::Display); }
266:   private:
267:   } ;
268:
269:   void PartsCatalog::Insert(Part * newPart)
270:   {
271:    int partNumber = newPart->GetPartNumber();
272:    int offset;
273:
274:    if (!Find(offset, partNumber))
275:      PartsList::Insert(newPart);
276:    else
277:    {
278:      cout << partNumber << " was the ";
279:      switch (offset)
280:      {
281:       case 0: cout << "first "; break;
```

```
282:       case 1: cout << "second "; break;
283:       case 2: cout << "third "; break;
284:       default: cout << offset+1 << "th ";
285:      }
286:      cout << "entry. Rejected!\ n";
287:    }
288:   }
289:
290:   int PartsCatalog::Exists(int PartNumber)
291:   {
292:    int offset;
293:    Find(offset,PartNumber);
294:    return offset;
295:   }
296:
297:   Part * PartsCatalog::Get(int PartNumber)
298:   {
299:    int offset;
300:    return (Find(offset, PartNumber));
301:
302:   }
303:
304:   int main()
305:   {
306:    PartsCatalog pc;
307:    Part * pPart = 0;
308:    int PartNumber;
309:    int value;
310:    int choice;
311:
312:    while (1)
313:    {
314:      cout << "(0)Quit (1)Car (2)Plane: ";
315:      cin >> choice;
316:
317:      if (!choice)
318:       break;
319:
320:      cout << "New PartNumber?: ";
321:      cin >> PartNumber;
322:
323:      if (choice == 1)
324:      {
325:       cout << "Model Year?: ";
326:       cin >> value;
327:       pPart = new CarPart(value,PartNumber);
328:      }
329:      else
330:      {
331:       cout << "Engine Number?: ";
```

```
332:        cin >> value;
333:        pPart = new AirPlanePart(value,PartNumber);
334:       }
335:       pc.Insert(pPart);
336:     }
337:     pc.ShowAll();
338:     return 0;
339:   }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4434
Model Year?: 93
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
1234 was the first entry. Rejected!
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2345
Model Year?: 93
(0)Quit (1)Car (2)Plane: 0

Part Number: 1234
Model Year: 94

Part Number: 2345
Model Year: 93

Part Number: 4434
Model Year: 93
```

АНАЛИЗ

В строке 82 класс `PartsList` объявляется другом класса `PartNode`.

В данном случае объявление класса другом происходит в разделе `public` объявления класса `PartNode`, но так поступать вовсе не обязательно. Это объявление можно размещать в любом месте объявления класса, что не изменит его суть. В результате объявления класса как друга все закрытые методы и переменные-члены класса `PartNode` становятся доступными любой функции-члену класса `PartsList`.

В строке 160 были внесены изменения в вызове функции-члена `GetFirst()` с учетом появившихся новых возможностей. Теперь вместо возвращения `pHead->GetPart` эта функция может возвращать закрытую переменную-член `pHead->itsPart`. Аналогичным образом в функции `Insert()` можно написать `pNode->itsNext = pHead` вместо переменной-члена `pHead->SetNext(pHead)`.

В данном случае внесенные изменения существенно не улучшили код программы, поэтому нет особых причин делать класс `PartsList` другом `PartNode`. В данном примере просто хотелось проиллюстрировать, как работает ключевое слово `friend`.

Объявление классов-друзей следует применять с осторожностью. Класс объявляется как друг какого-либо иного класса в том случае, когда два класса тесно взаимодей-

ствуют друг с другом и открытие доступа одного класса к данным и методам другого класса существенно упрощает код программы. Однако зачастую проще организовать взаимодействие между классами с помощью открытых методов доступа.

От начинающих программистов C++ часто можно услышать замечание, что объявление классов-друзей противоречит принципу инкапсуляции, лежащему в основе объектно-ориентированного программирования. Это, честно говоря, довольно широко распространенная бессмыслица. Объявление класса-друга просто расширяет интерфейс другого класса, что влияет на инкапсуляцию не больше, чем открытое наследование классов.

Дружественный класс

Объявление одного класса другом какого-либо иного с помощью ключевого слова `friend` в объявлении второго класса открывает первому классу доступ к членам второго класса. Иными словами, я могу объявить вас своим другом, но вы не можете объявить себя моим другом.

Пример:

```
class PartNode{
   public:
   friend class PartsList;   // объявление класса PartsList другом PartNode
```

Функции-друзья

Иногда бывает необходимо предоставить права доступа не всему классу, а только одной или нескольким функциям-членам. Это реализуется посредством объявления друзьями функций-членов другого класса. Причем объявлять другом весь класс вовсе не обязательно. Фактически другом можно объявить любую функцию, независимо от того, является ли она функцией-членом другого класса или нет.

Функции-друзья и перегрузка оператора

В листинге 15.1 представлен класс `String`, в котором перегружается `operator+`. В нем также объявляется конструктор, принимающий указатель на константную строку, поэтому объект класса `String` можно создавать из строки с концевым нулевым символом.

Строки в C и C++ представляют собой массивы символов, заканчивающиеся концевым нулевым символом. Такая строка получается, например, в следующем выражении присвоения: `myString[] = "Hello World"`.

Но чего невозможно сделать в классе String, так это получить новую строку в результате сложения объекта этого класса с массивом символов:

```
char cString[] = { "Hello"} ;
String sString(" World");
String sStringTwo = cString + sString; //ошибка!
```

Строки нельзя использовать с перегруженной функции operator+. Как объяснялось на занятии 10, выражение cString + sString на самом деле вызывает функцию cString.operator+(sString). Поскольку функция operator+() не может вызываться для символьной строки, данная попытка приведет к ошибке компиляции.

Эту проблему можно решить, объявив функцию-друга в классе String, которая перегружает operator+ таким образом, чтобы суммировать два объекта String. Соответствующий конструктор класса String преобразует строки в объекты String, после чего вызывается функция-друг operator+, выполняющая конкатенацию двух объектов.

Листинг 15.8. Функция-друг operator+

```
1:  //Листинг 15.8. Операторы друзья
2:
3:  #include <iostream.h>
4:  #include <string.h>
5:
6:  // Рудиментарный класс string
7:  class String
8:  {
9:   public:
10:    // constructors
11:    String();
12:    String(const char *const);
13:    String(const String &);
14:    ~String();
15:
16:    // перегруженные операторы
17:    char & operator[](int offset);
18:    char operator[](int offset) const;
19:    String operator+(const String&);
20:    friend String operator+(const String&, const String&);
21:    void operator+=(const String&);
22:    String & operator= (const String &);
23:
24:    // методы общего доступа
25:    int GetLen()const { return itsLen; }
26:    const char * GetString() const { return itsString; }
27:
28:  private:
29:    String (int);      // закрытый конструктор
30:    char * itsString;
31:    unsigned short itsLen;
32: } ;
33:
```

```
34:  // конструктор, заданный по умолчанию, создает строку длиной 0 байт
35:  String::String()
36:  {
37:    itsString = new char[1];
38:    itsString[0] = '\ 0';
39:    itsLen=0;
40:    // cout << "\ tDefault string constructor\ n";
41:    // ConstructorCount++;
42:  }
43:
44:  // закрытый конструктор, используемый только
45:  // методами класса для создания новой строки
46:  // указанного размера, заполненной нулями.
47:  String::String(int len)
48:  {
49:    itsString = new char[len+1];
50:    for (int i = 0; i<=len; i++)
51:     itsString[i] = '\ 0';
52:    itsLen=len;
53:    // cout << "\ tString(int) constructor\ n";
54:    // ConstructorCount++;
55:  }
56:
57:  // Преобразует массив символов в строку
58:  String::String(const char * const cString)
59:  {
60:    itsLen = strlen(cString);
61:    itsString = new char[itsLen+1];
62:    for (int i = 0; i<itsLen; i++)
63:     itsString[i] = cString[i];
64:    itsString[itsLen]='\ 0';
65:    // cout << "\ tString(char*) constructor\ n";
66:    // ConstructorCount++;
67:  }
68:
69:  // конструктор-копировщик
70:  String::String (const String & rhs)
71:  {
72:    itsLen=rhs.GetLen();
73:    itsString = new char[itsLen+1];
74:    for (int i = 0; i<itsLen;i++)
75:     itsString[i] = rhs[i];
76:    itsString[itsLen] = '\ 0';
77:    // cout << "\ tString(String&) constructor\ n";
78:    // ConstructorCount++;
79:  }
80:
81:  // деструктор, освобождает занятую память
82:  String::~String ()
83:  {
```

```
84:    delete [] itsString;
85:    itsLen = 0;
86:    // cout << "\ tString destructor\ n";
87:  }
88:
89:  // этот оператор освобождает память, а затем
90:  // копирует строку и размер
91:  String& String::operator=(const String & rhs)
92:  {
93:    if (this == &rhs)
94:     return *this;
95:    delete [] itsString;
96:    itsLen=rhs.GetLen();
97:    itsString = new char[itsLen+1];
98:    for (int i = 0; i<itsLen;i++)
99:     itsString[i] = rhs[i];
100:    itsString[itsLen] = '\ 0';
101:    return *this;
102:   // cout << "\ tString operator=\ n";
103:  }
104:
105:  // неконстантный оператор индексирования,
106:  // возвращает ссылку на символ, который можно
107:  // изменить!
108:  char & String::operator[](int offset)
109:  {
110:   if (offset > itsLen)
111:     return itsString[itsLen-1];
112:   else
113:     return itsString[offset];
114:  }
115:
116:  // константный оператор индексирования,
117:  // используется для константных объектов (см. конструктор-копировщик!)
118:  char String::operator[](int offset) const
119:  {
120:   if (offset > itsLen)
121:     return itsString[itsLen-1];
122:   else
123:     return itsString[offset];
124:  }
125:  // создает новый объект String, добавляя
126:  // текущий объект к rhs
127:  String String::operator+(const String& rhs)
128:  {
129:    int totalLen = itsLen + rhs.GetLen();
130:    String temp(totalLen);
131:    int i, j;
132:    for (i = 0; i<itsLen; i++)
133:     temp[i] = itsString[i];
```

```
134:    for (j = 0, i = itsLen; j<rhs.GetLen(); j++, i++)
135:     temp[i] = rhs[j];
136:    temp[totalLen]='\ 0';
137:    return temp;
138:  }
139:
140:  // создает новый объект String
141:  // из двух объектов класса String
142:  String operator+(const String& lhs, const String& rhs)
143:  {
144:    int totalLen = lhs.GetLen() + rhs.GetLen();
145:    String temp(totalLen);
146:    int i, j;
147:    for (i = 0; i<lhs.GetLen(); i++)
148:     temp[i] = lhs[i];
149:    for (j = 0, i = lhs.GetLen();; j<rhs.GetLen(); j++, i++)
150:     temp[i] = rhs[j];
151:    temp[totalLen]='\ 0';
152:    return temp;
153:  }
154:
155:  int main()
156:  {
157:    String s1("String One ");
158:    String s2("String Two ");
159:    char *c1 = { "C-String One " } ;
160:    String s3;
161:    String s4;
162:    String s5;
163:
164:    cout << "s1: " << s1.GetString() << endl;
165:    cout << "s2: " << s2.GetString() << endl;
166:    cout << "c1: " << c1 << endl;
167:    s3 = s1 + s2;
168:    cout << "s3: " << s3.GetString() << endl;
169:    s4 = s1 + c1;
170:    cout << "s4: " << s4.GetString() << endl;
171:    s5 = c1 + s2;
172:    cout << "s5: " << s5.GetString() << endl;
173:   return 0;
174:  }
```

РЕЗУЛЬТАТ

```
s1: String One
s2: String Two
c1: C-String One
s3: String One String Two
s4: String One C-String One
s5: C-String One String Two
```

Анализ: Объявления всех методов класса String, за исключением operator+, остались такими же, как в листинге 15.1. В строке 20 листинга 15.8 перегружается новый operator+, который принимает две ссылки на константные строки и возвращает строку, полученную в результате конкатенации исходных строк. Эта функция объявлена как друг класса String.

Обратите внимание, что функция operator+ не является функцией-членом этого или любого другого класса. Она объявляется среди функций-членов класса String как друг, но не как член класса. Тем не менее это все же полноценное объявление функции, и нет необходимости еще раз объявлять в программе прототип этой функции.

Выполнение функции operator+ определяется в строках 142–153. Определение выполнения функции аналогично приведенному в версии программы, представленной в листинге 15.1, за тем исключением что функция принимает в качестве аргументов две строки, обращаясь к ними с помощью открытых методов доступа класса.

Перегруженный оператор применяется в строке 171, где выполняется конкатенация двух строк.

Функции-друзья

Для объявления функции как друга класса используется ключевое слово friend, за которым следует объявление функции. Это не предоставляет функции доступ к указателю this, но обеспечивает доступ ко всем закрытым и защищенным данным и функциям-членам.

Пример:

```
class PartNode
  {  // ...
    // сделаем функцию-член другого класса другом этого класса
    friend void PartsList::Insert(Part *);
    // сделаем другом глобальную функцию
    friend int SomeFunction();
    // ...
  };
```

Перегрузка оператора вывода

Настало время снабдить наш класс String возможностью использовать объект cout для вывода своих данных так же, как при выводе данных базовых типов. До сих пор для вывода значения переменной-члена приходилось использовать следующее выражение:

```
cout << theString.GetString();
```

Неплохо было бы иметь возможность написать просто

```
cout << theString;
```

Для этого необходимо перегрузить функцию operator<<(). Более подробно использование потоков iostreams для вывода данных обсуждается на занятии 16. А в листинге 15.9 объявляется перегрузка функции operator<< как друга.

Листинг 15.9. Перегрузка `operator<<()`

```
1:  #include <iostream.h>
2:  #include <string.h>
3:
4:  class String
5:  {
6:   public:
7:     // конструкторы
8:     String();
9:     String(const char *const);
10:     String(const String &);
11:    ~String();
12:
13:    // перегруженные операторы
14:    char & operator[](int offset);
15:    char operator[](int offset) const;
16:    String operator+(const String&);
17:    void operator+=(const String&);
18:    String & operator= (const String &);
19:    friend ostream& operator<<
20:      ( ostream& theStream,String& theString);
21:    // Общие методы доступа
22:    int GetLen()const { return itsLen; }
23:    const char * GetString() const { return itsString; }
24:
25:  private:
26:    String (int);      // закрытый конструктор
27:    char * itsString;
28:    unsigned short itsLen;
29: } ;
30:
31:
32: // конструктор, заданный по умолчанию, создает строку длиной 0 байт
33: String::String()
34: {
35:   itsString = new char[1];
36:   itsString[0] = '\ 0';
37:   itsLen=0;
38:   // cout << "\ tDefault string constructor\ n";
39:   // ConstructorCount++;
40: }
41:
42: // закрытый конструктор, используемый только
43: // методами класса для создания новой строки
44: // указанного размера, заполненной значениями NULL.
45: String::String(int len)
46: {
47:   itsString = new char[len+1];
48:   for (int i = 0; i<=len; i++)
```

```
49:     itsString[i] = '\ 0';
50:    itsLen=len;
51:    // cout << "\ tString(int) constructor\ n";
52:    // ConstructorCount++;
53:  }
54:
55:  // Преобразует массив символов в строку
56:  String::String(const char * const cString)
57:  {
58:    itsLen = strlen(cString);
59:    itsString = new char[itsLen+1];
60:    for (int i = 0; i<itsLen; i++)
61:     itsString[i] = cString[i];
62:    itsString[itsLen]='\ 0';
63:    // cout << "\ tString(char*) constructor\ n";
64:    // ConstructorCount++;
65:  }
66:
67:  // конструктор-копировщик
68:  String::String (const String & rhs)
69:  {
70:    itsLen=rhs.GetLen();
71:    itsString = new char[itsLen+1];
72:    for (int i = 0; i<itsLen;i++)
73:     itsString[i] = rhs[i];
74:    itsString[itsLen] = '\ 0';
75:    // cout << "\ tString(String&) constructor\ n";
76:    // ConstructorCount++;
77:  }
78:
79:  // деструктор освобождает занятую память
80:  String::~String ()
81:  {
82:    delete [] itsString;
83:    itsLen = 0;
84:    // cout << "\ tString destructor\ n";
85:  }
86:
87:  // оператор равенства освобождает память, а затем
88:  // копирует строку и размер
89:  String& String::operator=(const String & rhs)
90:  {
91:    if (this == &rhs)
92:     return *this;
93:    delete [] itsString;
94:    itsLen=rhs.GetLen();
95:    itsString = new char[itsLen+1];
96:    for (int i = 0; i<itsLen;i++)
97:     itsString[i] = rhs[i];
98:    itsString[itsLen] = '\ 0';
```

```
99:    return *this;
100:   // cout << "\ tString operator=\ n";
101:  }
102:
103:  // неконстантный оператор индексирования,
104:  // возвращает ссылку на символ, который можно
105:  // изменить!
106:  char & String::operator[](int offset)
107:  {
108:   if (offset > itsLen)
109:     return itsString[itsLen-1];
110:   else
111:     return itsString[offset];
112:  }
113:
114:  // константный оператор индексирования,
115:  // используется для константных объектов (см. конструктор-копировщик!)
116:  char String::operator[](int offset) const
117:  {
118:   if (offset > itsLen)
119:     return itsString[itsLen-1];
120:   else
121:     return itsString[offset];
122:  }
123:
124:  // создает новую строку, добавляя текущую
125:  // строку к rhs
126:  String String::operator+(const String& rhs)
127:  {
128:   int totalLen = itsLen + rhs.GetLen();
129:   String temp(totalLen);
130:   int i, j;
131:   for (i = 0; i<itsLen; i++)
132:     temp[i] = itsString[i];
133:   for (j = 0; j<rhs.GetLen(); j++, i++)
134:     temp[i] = rhs[j];
135:   temp[totalLen]='\ 0';
136:   return temp;
137:  }
138:
139:  // изменяет текущую строку, ничего не возвращая
140:  void String::operator+=(const String& rhs)
141:  {
142:   unsigned short rhsLen = rhs.GetLen();
143:   unsigned short totalLen = itsLen + rhsLen;
144:   String temp(totalLen);
145:   int i, j;
146:   for (i = 0; i<itsLen; i++)
147:     temp[i] = itsString[i];
148:   for (j = 0, i = 0; j<rhs.GetLen(); j++, i++)
```

```
149:     temp[i] = rhs[i-itsLen];
150:    temp[totalLen]='\ 0';
151:    *this = temp;
152:  }
153:
154:  // int String::ConstructorCount =
155:  ostream& operator<< ( ostream& theStream,String& theString)
156:  {
157:    theStream << theString.itsString;
158:    return theStream;
159:  }
160:
161:  int main()
162:  {
163:    String theString("Hello world.");
164:    cout << theString;
165:   return 0;
166:  }
```

РЕЗУЛЬТАТ

```
Hello world.
```

АНАЛИЗ

В строке 19 `operator<<` объявляется как функция-друг, которая принимает ссылки на `ostream` и `String` и возвращает ссылку на `ostream`. Обратите внимание, что она не является функцией-членом класса `String`. Поскольку эта функция возвращает ссылку на `ostream`, можно конкатенировать вызовы `operator<<` следующим образом:

```
cout << "myAge: " << itsAge << " years. ";
```

Выполнение этой функции-друга представлено строками 155—159. Основное назначение функции состоит в том, чтобы скрыть детали процедуры передачи строки в `iostream`. Больше ничего и не требуется. Более подробно о функции ввода и перегрузке `operator>>` вы узнаете на следующем занятии.

Резюме

Сегодня вы узнали, как делегировать ответственность за выполнение специальных задач вложенным объектам, а также выполнять один класс в пределах другого с помощью вложения или открытого наследования. Основное ограничение вложения — отсутствие у нового класса доступа к защищенным членам вложенного класса и возможности замещения функций-членов вложенного объекта. Вложение гораздо проще в использовании, чем закрытое наследование, поэтому по возможности следует применять этот подход.

Вы также узнали, как объявлять классы и функции-друзьями другого класса. Объявление функции друга позволяет перегрузить оператор ввода таким образом, что появляется возможность использовать объект `cout` в пользовательском классе точно так же, как в стандартных встроенных классах.

Напомним, что открытое наследования определяет производный класс как *уточнение* базового класса; вложение подразумевает *обладание* одним классом объектами другого класса, а закрытое наследование состоит в выполнении одного класса *средствами* другого класса. *Делегирование ответственности* реализуется либо вложением, либо закрытым наследованием, хотя первое предпочтительнее.

Вопросы и ответы

Почему так важно разбираться в особенностях отношений между классами при выборе различных подходов установки взаимосвязей между ними?

Язык программирования C++ создавался специально для разработки объектно-ориентированных программ. Характерной особенностью объектно-ориентированного программирования является моделирование в программе реальных отношений между объектами и явлениями окружающего мира, причем при выборе подходов программирования следует учитывать особенности этих отношений, чтобы максимально точно смоделировать реальность.

Почему вложение предпочтительнее закрытого наследования?

Современное программирование — это разрешение противоречий между достижением максимальной точности моделирования событий и предупреждением чрезвычайного усложнения программ. Поэтому чем больше объектов программы будут использоваться как "черные ящики", тем меньше всевозможных параметров нужно отслеживать при отладке или модернизации программы. Методы вложенных классов скрыты от пользователей, что нельзя сказать о закрытом наследовании.

Почему бы не описать все классы, объекты которых используются в других классах, друзьями этих классов?

Объявление одного класса другом какого-либо иного открывает закрытые методы и данные класса, что снижает инкапсуляцию класса. Лучше всего держать как можно больше членов одного класса закрытыми от всех остальных классов.

Если функция перегружается, нужно ли описывать каждый вариант этой функции другом класса?

Да. Если вы перегружаете функцию и хотите представить все варианты этой функции друзьями другого класса, то в описании класса каждый вариант функции должен сопровождаться ключевым словом `friend`.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из вопросов, предложенных ниже.

Контрольные вопросы

1. Как объявить класс, являющийся частным проявлением другого класса?
2. Как объявить класс, объекты которого должны использоваться в качестве переменных-членов другого класса?
3. В чем состоят различия между вложением и делегированием?
4. В чем состоят различия между делегированием и выполнением класса в пределах другого класса?

5. Что такое функция-друг?
6. Что такое класс-друг?
7. Если класс Dog является другом Boy, то можно ли сказать, что Boy — друг Dog?
8. Если класс Dog является другом Boy, а Terrier произведен от Dog, является ли Terrier другом Boy?
9. Если класс Dog является другом Boy, а Boy — другом House, можно ли считать Dog другом House?
10. Где необходимо размещать объявление функции-друга?

Упражнения

1. Объявите класс Animal, который содержит переменную-член, являющуюся объектом класса String.
2. Объявите класс BoundedArray, являющийся массивом.
3. Объявите класс Set, выполняемый в пределах массива BoundedArray.
4. Измените листинг 15.1 таким образом, чтобы класс String включал перегруженный оператор вывода (>>).
5. **Жучки**: найдите ошибки в этом коде:

```
1:  #include <iostream.h>
2:
3:  class Animal;
4:
5:  void setValue(Animal& , int);
6:
7:
8:  class Animal
9:  {
10: public:
11:   int GetWeight()const { return itsWeight; }
12:   int GetAge() const { return itsAge; }
13: private:
14:   int itsWeight;
15:   int itsAge;
16: } ;
17:
18: void setValue(Animal& theAnimal, int theWeight)
19: {
20:   friend class Animal;
21:   theAnimal.itsWeight = theWeight;
22: }
23:
24: int main()
25: {
26:   Animal peppy;
27:   setValue(peppy,5);
28: }
```

6. Исправьте листинг, приведенный в упражнении 5, и откомпилируйте его.
7. **Жучки**: найдите ошибки в этом коде:

```
1:  #include <iostream.h>
2:
3:  class Animal;
4:
5:  void setValue(Animal& , int);
6:  void setValue(Animal& ,int,int);
7:
8:  class Animal
9:  {
10: friend void setValue(Animal& ,int);
11: private:
12:   int itsWeight;
13:   int itsAge;
14: } ;
15:
16: void setValue(Animal& theAnimal, int theWeight)
17: {
18:   theAnimal.itsWeight = theWeight;
19: }
20:
21:
22: void setValue(Animal& theAnimal, int theWeight, int theAge)
23: {
24:   theAnimal.itsWeight = theWeight;
25:   theAnimal.itsAge = theAge;
26: }
27:
28: int main()
29: {
30:   Animal peppy;
31:   setValue(peppy,5);
32:   setValue(peppy,7,9);
33: }
```

8. Исправьте листинг, приведенный в упражнении 7, и откомпилируйте его.

Потоки

Ранее для вывода на экран и считывания с клавиатуры мы использовали объекты `cout` и `cin`, не понимая до конца принципов их работы. Сегодня вы узнаете:

- Что такое потоки ввода-вывода и как они используются
- Как с помощью потоков управлять вводом и выводом данных
- Как с помощью потоков записывать информацию в файл и затем считывать ее

Знакомство с потоками

Язык программирования C++ специально не определяет, каким образом данные выводятся на экран или в файл либо как они считываются программой. Тем не менее эти особенности являются важной частью работы программиста, поэтому стандартная библиотека C++ включает библиотеку `iostream`, упрощающую ввод-вывод (I/O).

Благодаря выделению операций ввода-вывода в отдельную библиотеку упрощается создание аппаратно независимого языка разработки программ для разных платформ. Это позволяет создать программу на C++ для компьютеров PC, а затем откомпилировать ее для рабочей станции Sun. Разработчики снабдили компилятор библиотеками для всех случаев. Так, по крайней мере, должно быть теоретически.

> Библиотека — это набор файлов OBJ, которые можно подключать к программе для получения дополнительных функциональных возможностей. Это наиболее распространенная форма многократного использования кода, и можно сказать, что она существует еще с тех пор, как первобытные программисты каменного века выбивали первые нули и единицы на стенах своих пещер.

Инкапсуляция

Классы `iostream` рассматривают информацию, выводимую программой на экран, как побитовый поток данных. Если данные выводятся в файл или на экран, то источник потока, как правило, содержится в программе. Если же поток направлен в проти-

воположную сторону, данные могут поступать с клавиатуры или файла на диске. В этом случае они заносятся в переменные.

Одна из основных целей использования потоков состоит в инкапсуляции процедуры обмена данными с диском или дисплеем компьютера. Сама программа работает только с потоками, которые реализуют эти процессы. Схематически эта идея проиллюстрирована на рис. 16.1.

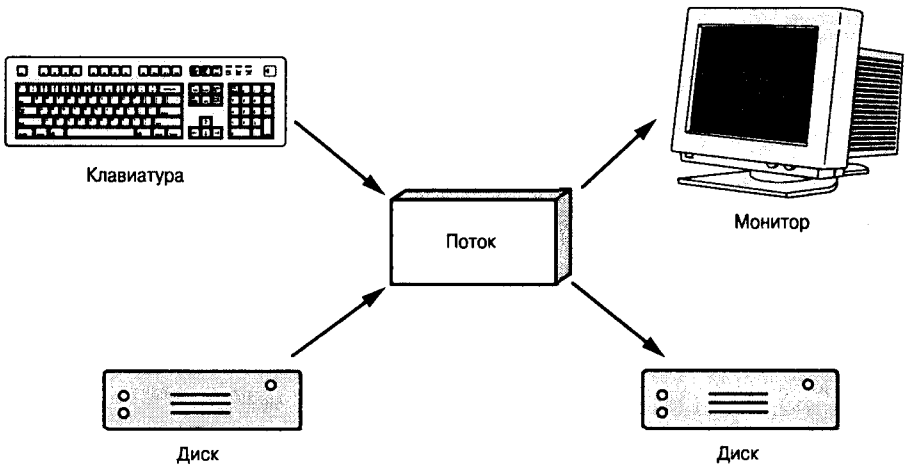


Рис. 16.1. Инкапсуляция с помощью потоков

Буферизация

Запись на диск (и в меньшей степени вывод на экран) обходится очень дорого. Запись данных на диск и считывание их с диска требует довольно много времени, что может надолго заблокировать выполнение программы. Для решения этой проблемы потоки обеспечивают *буферизацию*. Данные сначала записываются в буфер потока, а после его наполнения вся информация разом записывается на диск.

Суть идеи проиллюстрирована на примере знакомого со школьной скамьи бака с водой (рис. 16.2). Вода заливается сверху, и бак постепенно наполняется, поскольку нижний вентиль закрыт.

Когда вода (данные) достигает верха, нижний вентиль автоматически открывается и вся вода выливается (рис. 16.3).

Как только бак опустеет, нижний вентиль закрывается, а верхний открывается вновь, и вода снова поступает в бак (рис. 16.4).

В некоторых случаях необходимо, чтобы вода сразу же выливалась из бака, не дожидаясь его наполнения. В программировании такая ситуация называется очисткой буфера (рис. 16.5).

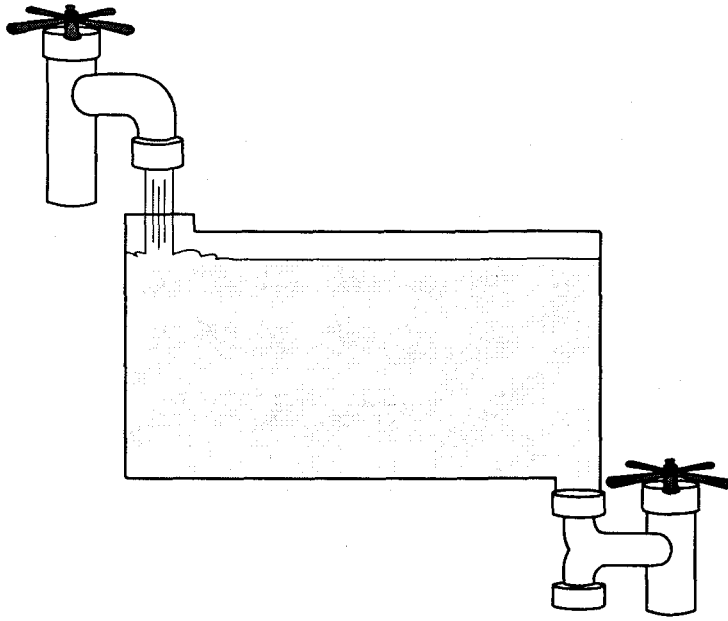


Рис. 16.2. Буфер наполняется данными, как закрытый бак — водой

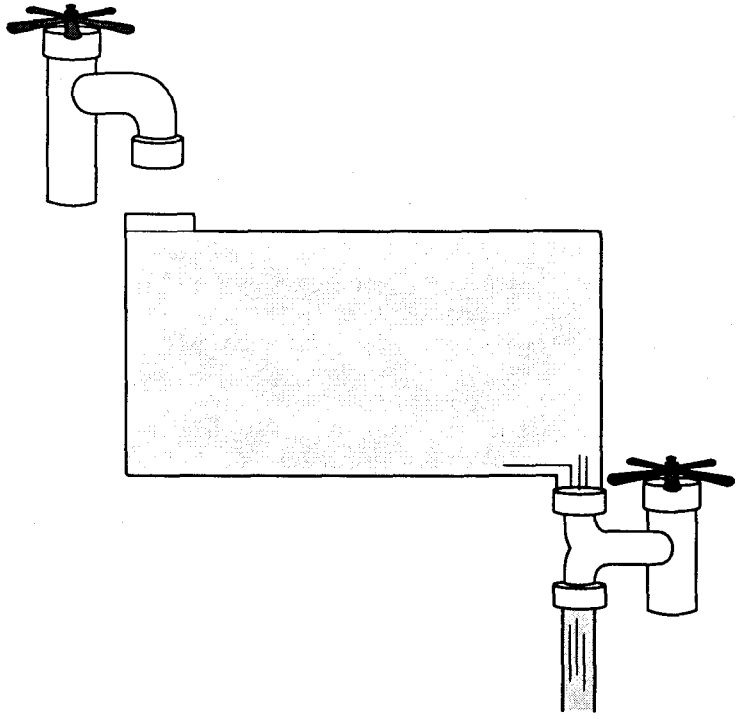


Рис. 16.3. Открывается сливной вентиль, и вода (данные) сливается из бака

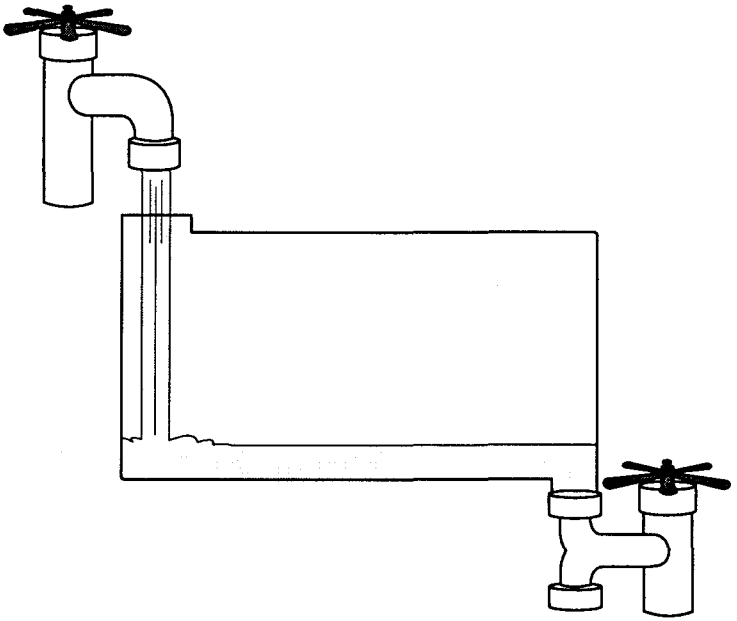


Рис. 16.4. Повторное наполнение бака

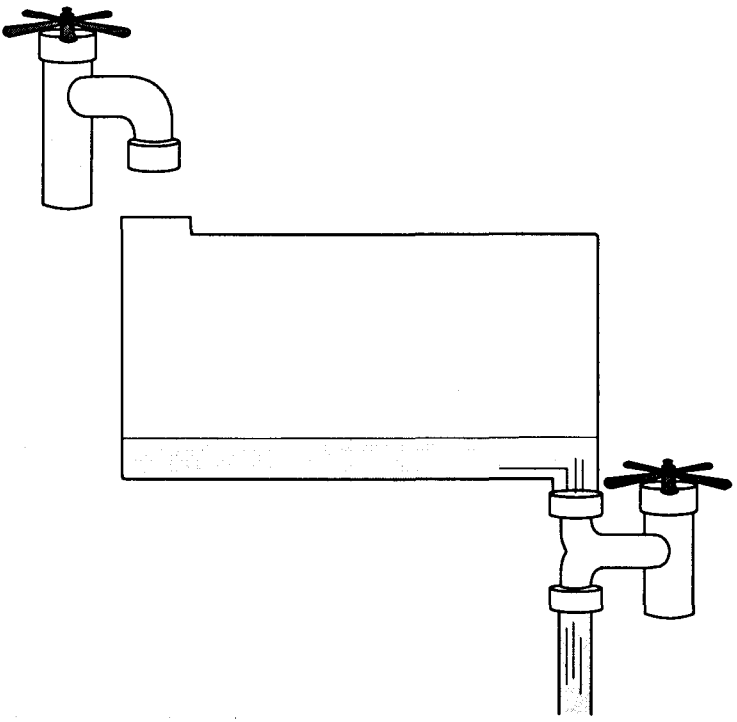


Рис. 16.5. Очистка буфера подобна экстренному сливу воды

Потоки и буферы

В C++ применяется объектно-ориентированный подход к реализации обмена данными с буферизированными потоками.

- Класс `streambuf` управляет буфером, поэтому его функции предоставляют возможность наполнять, опорожнять и очищать буфер, а также выполнять с ним другие операции.
- Класс `ios` является базовым для классов потоков ввода-вывода. В качестве переменной-члена класса `ios` выступает объект `streambuf`.
- Классы `istream` и `ostream` являются производными от класса `ios` и отвечают соответственно за потоковый ввод и вывод данных.
- Класс `iosteam` является производным от классов `istream` и `ostream` и обеспечивает методы ввода-вывода для печати на экран.
- Классы `fstream` используются для ввода-вывода из файлов.

Стандартные объекты ввода-вывода

При запуске программы, включающей классы `iostreams`, создаются и инициируются четыре объекта.

> Библиотека класса `iostream` встроена в компилятор. Чтобы добавить в свою программу методы этого класса, достаточно в первых строках программы включить выражение `#include<iostream>`.

- Объект `cin` (произносится как "си-ин" от английского "see-in") обрабатывает ввод с клавиатуры.
- Объект `cout` (произносится как "си-аут" от английского "see-out") обрабатывает вывод на экран.
- Объект `cerr` (произносится как "си-эр" от английского "see-err") обрабатывает не буферизированный вывод ошибок на стандартное устройство вывода сообщений об ошибках, т.е. на экран. Поскольку вывод не буферизированный, то все данные, направляемые в `cerr`, сразу же выводятся устройством вывода.
- Объект `clog` (произносится как "си-лог" от английского "see-log") обрабатывает буферизированные сообщения об ошибках, которые выводятся на стандартное устройство вывода сообщений об ошибках (экран). Зачастую эти сообщения переадресуются в файл регистрации. Об этом вы узнаете далее в главе.

Переадресация

Каждое стандартное устройство ввода и вывода, в том числе устройство вывода сообщений об ошибках, может осуществлять переадресацию на другие устройства. Например, системные сообщения об ошибках часто переадресуются в файл регистрации. Для ввода и вывода данных программой также можно использовать файлы, для чего служат специальные команды операционной системы.

Под переадресацией понимают пересылку выводимых данных в устройство, либо считывание данных с устройства, отличное от установленного по умолчанию. В операционных системах DOS и UNIX используются специальные операторы переадресации ввода (<) и вывода (>).

Пайпингом называется использование вывода одной программы в качестве ввода для другой.

Операционная система DOS содержит ограниченный набор команд переадресации для вывода (>) и ввода (<). Команды переадресации системы UNIX более разнообразны, однако основная идея остается той же: данные выводятся на экран, записываются в файл или передаются другой программе. Ввод в программу осуществляется из файлов или с клавиатуры.

В целом переадресация больше относится к функциям операционной системы, а не библиотек iosream. Язык C++ предоставляет доступ к четырем стандартным устройствам и необходимый набор команд для переадресации устройств ввода-вывода.

Вывод данных с помощью cin

Глобальный объект cin отвечает за ввод данных и становится доступным при включении в программу класса iostream. В предыдущих примерах используется перегруженный оператор ввода (>>) для присвоения вводимых данных переменным программы. Для ввода данных используется следующий синтаксис:

```
int someVariable;
cout << "Enter a number: ";
cin >> someVariable;
```

Другой глобальный объект, cout, и его использование для вывода данных обсуждается несколько ниже. Сейчас же остановимся на третьей строке: cin >> someVariable;. Что же представляет собой объект cin?

На глобальность этого объекта указывает тот факт, что его не нужно объявлять в коде программы. Объект cin включает перегруженный оператор ввода (>>), который записывает данные, хранимые в буфере cin, в локальную переменную someVariable. Причем оператор ввода перегружен таким образом, что подходит для ввода данных всех базовых типов, включая int&, short&, long&, double&, float&, char&, char* и т.п. Когда компилятор встречает выражение cin >> someVariable, то вызывается вариант оператора ввода, соответствующий типу переменной someVariable. В приведенным выше примере someVariable имеет тип int, поэтому вызывается следующий вариант перегруженной функции:

```
istream & operator>> (int&)
```

Обратите внимание, поскольку параметр передается как ссылка, оператор ввода может изменять исходную переменную. Использование cin показано в листинге 16.1.

Листинг 16.1. Использование cin для ввода данных разных типов

```
1:    //Листинг 16.1. Ввод даннах с помощью cin
2:
3:    #include <iostream.h>
4:
5:    int main()
6:    {
7:       int myInt;
```

```
8:      long myLong;
9:      double myDouble;
10:     float myFloat;
11:     unsigned int myUnsigned;
12:
13:     cout << "int: ";
14:     cin >> myInt;
15:     cout << "Long: ";
16:     cin >> myLong;
17:     cout << "Double: ";
18:     cin >> myDouble;
19:     cout << "Float: ";
20:     cin >> myFloat;
21:     cout << "Unsigned: ";
22:     cin >> myUnsigned;
23:
24:     cout << "\ n\ nInt:\ t" << myInt << endl;
25:     cout << "Long:\ t" << myLong << endl;
26:     cout << "Double:\ t" << myDouble << endl;
27:     cout << "Float:\ t" << myFloat << endl;
28:     cout << "Unsigned:\ t" << myUnsigned << endl;
29:   return 0;
30:  }
```

РЕЗУЛЬТАТ

```
int: 2
Long: 70000
Double: 987654321
Float: 3.33
Unsigned: 25

Int:    2
Long:   70000
Double: 9.87654e+08
Float:  3.33
Unsigned:       25
```

АНАЛИЗ В строках 7–11 объявляются переменные разных типов. В строках 13–22 пользователю предлагается ввести значения для этих переменных, после чего результаты выводятся в строках 24–28 (с помощью `cin`).

Выводимая программой информация говорит о том, что переменные записываются и выводятся в соответствии с их типом.

Строки

Объект `cin` также может принимать в качестве аргумента указатель на строку символов (`char*`), что позволяет создавать буфер символов и заполнять его с помощью `cin`. Например, можно написать следующее:

```
char YourName[50]
cout << "Enter your name: ";
cin >> YourName;
```

Если ввести имя Jesse, переменная YourName заполнится символами J, e, s, s, e и \0. Последним будет концевой нулевой символ, так как cin автоматически вставляет его. Поэтому при определении размера буфера нужно позаботиться о том, чтобы он был достаточно большим и мог вместить все символы строки, включая концевой нулевой символ. Более подробно о поддержке концевого нулевого символа стандартными библиотечными строковыми функциями речь пойдет на занятии 21.

Проблемы, возникающие при вводе строк

Успешно выполнив все описанные ранее операции с объектом cin, вы будете неприятно удивлены, если попытаетесь ввести в строке полное имя. Дело в том, что cin рассматривает пробел как заданный по умолчанию разделитель строк. После того как в строке обнаруживается пробел, ввод строки завершается добавлением концевого нулевого символа. Эта проблема показана в листинге 16.2.

Листинг 16.2. Попытка ввода более одного слова с помощью `cin`

```
1:    //Листинг 16.2. Проблемы с вводом строки с помощью cin
2:
3:    #include <iostream.h>
4:
5:    int main()
6:    {
7:       char YourName[50];
8:       cout << "Your first name: ";
9:       cin >> YourName;
10:      cout << "Here it is: " << YourName << endl;
11:      cout << "Your entire name: ";
12:      cin >> YourName;
13:      cout << "Here it is: " << YourName << endl;
14:    return 0;
15:   }
```

РЕЗУЛЬТАТ

```
Your first name: Jesse
Here it is: Jesse
Your entire name: Jesse Liberty
Here it is: Jesse .
```

АНАЛИЗ

Строкой 7 для хранения вводимой пользователем строки создается массив символов. В строке 8 пользователю предлагается ввести имя, и, как видно из вывода, это имя сохраняется правильно.

В строке 11 пользователю предлагается ввести не только имя, но и фамилию. Ввод осуществляется только до тех пор, пока cin не обнаружит пробел между именем и фамилией. После этого ввод строки прекращается и оставшаяся информация теряется. Это не совсем то, что было нужно.

Чтобы понять, почему cin работает именно так, проанализируйте листинг 16.3, в котором показан пример ввода строки значений.

Листинг 16.3. Ввод строки значений

```
1:    //Листинг 16.3. Ввод строки значений с помощью cin
2:
3:    #include <iostream.h>
4:
5:    int main()
6:    {
7:       int myInt;
8:       long myLong;
9:       double myDouble;
10:      float myFloat;
11:      unsigned int myUnsigned;
12:      char myWord[50];
13:
14:      cout << "int: ";
15:      cin >> myInt;
16:      cout << "Long: ";
17:      cin >> myLong;
18:      cout << "Double: ";
19:      cin >> myDouble;
20:      cout << "Float: ";
21:      cin >> myFloat;
22:      cout << "Word: ";
23:      cin >> myWord;
24:      cout << "Unsigned: ";
25:      cin >> myUnsigned;
26:
27:      cout << "\ n\ nInt:\ t" << myInt << endl;
28:      cout << "Long:\ t" << myLong << endl;
29:      cout << "Double:\ t" << myDouble << endl;
30:      cout << "Float:\ t" << myFloat << endl;
31:      cout << "Word: \ t" << myWord << endl;
32:      cout << "Unsigned:\ t" << myUnsigned << endl;
33:
34:      cout << "\ n\ nInt, Long, Double, Float, Word, Unsigned: ";
35:      cin >> myInt >> myLong >> myDouble;
36:      cin >> myFloat >> myWord >> myUnsigned;
37:      cout << "\ n\ nInt:\ t" << myInt << endl;
38:      cout << "Long:\ t" << myLong << endl;
39:      cout << "Double:\ t" << myDouble << endl;
40:      cout << "Float:\ t" << myFloat << endl;
41:      cout << "Word: \ t" << myWord << endl;
42:      cout << "Unsigned:\ t" << myUnsigned << endl;
43:
44:
45:    return 0;
46:    }
```

РЕЗУЛЬТАТ

```
Int: 2
Long: 30303
Double: 393939397834
Float: 3.33
Word: Hello
Unsigned: 85

Int:    2
Long:   30303
Double: 3.93939e+11
Float:  3.33
Word:   Hello
Unsigned:       85

Int, Long, Double, Float, Word, Unsigned: 3 304938 393847473 6.66 bye -2

Int:    3
Long:   304938
Double: 3.93847e+08
Float:  6.66
Word:   bye
Unsigned: 4294967294
```

Вновь в программе объявляются переменные разных типов и массив символов. Пользователю предлагается последовательно ввести данные разных типов, чтобы убедиться что программа поддерживает ввод данных любого типа.

АНАЛИЗ

В строке 34 пользователю предлагается ввести все данные сразу в определенном порядке, после чего каждое введенное значение присваивается соответствующей переменной. Благодаря тому что `cin` рассматривает пробелы между словами как разделители, становится возможной инициализация всех переменных. В противном случае программа пыталась бы ввести всю строку в одну переменную, что было бы ошибкой.

Обратите внимание на строку 42, в которой выводится беззнаковое целое число. Пользователь ввел значение `-2`. Поскольку программа была проинструктирована, что вводится беззнаковое целое число, то вместо знакового `-2` будет введено беззнаковое двоичное представление этого числа. Поэтому при выводе с помощью `cout` на экране отображается значение `4294967294`, являющееся двоичным представлением числа `-2`.

Позже вы узнаете, как вводить в буфер строки, содержащие несколько слов, разделенных пробелами. Сейчас же рассмотрим подробнее использование `cin` для ввода данных сразу в несколько переменных, как в строках 35–36.

Оператор >> возвращает ссылку на объект istream

Оператор `>>` возвращает ссылку на объект `istream`. Но поскольку `cin` сам является объектом `istream`, результат выполнения одной операции ввода может быть началом следующей операции ввода, как показано ниже:

```
Int VarOne, varTwo, varThree;
cout << "Enter three numbers: "
cin >> VarOne >> varTwo >> varThree;
```

В строке `cin >> VarOne >> varTwo >> varThree;` сначала выполняется первый ввод `cin >> VarOne`, в результате чего возвращается объект `istream`, позволяющий выполнить присвоение второго значения переменной `varTwo`. Это равносильно следующей записи:

```
((cin >> VarOne) >> varTwo) >> varThree;
```

Аналогичный подход используется с объектом `cout`, но речь об этом пойдет дальше.

Другие методы объекта cin

В дополнение к перегружаемому оператору `>>` объект `cin` имеет множество других встроенных методов. Они используются в тех случаях, когда необходим более совершенный контроль над вводом данных.

Ввод одного символа

Вариант `operator>>`, принимающий ссылку на символ, может использоваться для считывания одного символа со стандартного устройства ввода. Для этого используется функция-член `get()`. При этом можно применять `get()` без параметров или использовать вариант этой же функции, принимающей в качестве параметра ссылку на символ.

Использование функции get() без параметров

Сначала рассмотрим использование функции `get()` без параметров. В этом случае функция возвращает значение найденного символа или `EOF` (end of file — конец файла) при достижении конца файла. Функция `get()` без параметров используется редко. Так, `cin.get()` нельзя использовать для последовательной инициализации ряда переменных, поскольку возвращаемое функцией значение не является объектом `iostream`. Именно поэтому следующая запись работать не будет:

```
cin.get() >>myVarOne >> myVarTwo // ошибка
```

Запись `cin.get() >> myVarOne` возвращает значение типа `int`, а не объект `iostream`.

Пример использования функции `get()` без параметров показан в листинге 16.4.

Листинг 16.4. Использование функции `get()` без параметров

```
1:    // Листинг 16.4. Использование get() без параметров
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       char ch;
7:       while ( (ch = cin.get()) != EOF)
8:       {
9:          cout << "ch: " << ch << endl;
10:      }
11:      cout << "\ nDone!\ n";
12:    return 0;
13:   }
```

ПРИМЕЧАНИЕ

Для выхода из этой программы придется ввести символ конца файла с клавиатуры. С этой целью в операционной системе DOS используется комбинация клавиш <Ctrl+Z>, а в UNIX — <Ctrl+D>.

РЕЗУЛЬТАТ

```
Hello

ch: H
ch: e
ch: l
ch: l
ch: o
ch:

World
ch: W
ch: o
ch: r
ch: l
ch: d
ch:

(ctrl-z)
Done!
```

АНАЛИЗ В строке 6 объявляется локальная символьная переменная. В цикле `while` символ, полученный от `cin.get()`, присваивается `ch`, и если возвращенный символ не `EOF`, то он выводится на печать. Цикл завершается вводом `EOF`, комбинацией клавиш <Ctrl+Z> в DOS или <Ctrl+D> в UNIX.

Следует отметить, что не во всех версиях библиотеки `istream` поддерживается функция-член `get()`, *хотя сейчас она является частью стандарта ANSI/ISO.*

Использование функции get() с параметром

При установке в функции `get()` параметра, указывающего на символьную переменную, этой переменной присваивается очередной символ потока ввода. При этом возвращается объект `iostream`, что позволяет вводить последовательный ряд значений, как показано в листинге 16.5.

Листинг 16.5. Использование функции `get()` с параметрами

```
1:    // Листинг 16.5. Использование get() с параметрами
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       char a, b, c;
7:
8:       cout << "Enter three letters: ";
9:
10:      cin.get(a).get(b).get(c);
```

```
11:
12:       cout << "a: " << a << "\ nb: " << b << "\ nc: " << c << endl;
13:     return 0;
14:   }
```

Результат:

```
Enter three letters: one

a: o
b: n
c: e
```

Анализ: В строке 6 объявляются символьные переменные `a`, `b` и `c`. В строке 10 трижды последовательно вызывается функция `cin.get()`. Сначала вызывается `cin.get(a)`, в результате первый символ буфера ввода заносится в `a` и возвращается объект `cin`, после чего происходит вызов `cin.get(b)`, присваивающий очередной символ буфера переменной `b`. Аналогичным образом вызывается функция `cin.get(c)`, присваивающая следующий символ переменной `c`.

Поскольку `cin.get()` возвращает `cin`, можно было записать это следующим образом:

```
cin.get(a) >> b;
```

В этом случае `cin.get(a)` возвратит `cin`, поэтому следующее выражение будет иметь вид: `cin >> b;`.

Рекомендуется

Используйте оператор ввода >>, когда необходимо вводить значения, разделенные пробелами в строке.

Рекомендуется

Используйте функцию `get()` с символьным параметром, если нужно последовательно вводить все символы строки, включая пробелы.

Ввод строк со стандартного устройства ввода

Для заполнения массива символов можно использовать как оператор ввода (>>), так и методы `get()` и `getline()`.

Еще один вариант перегруженной функции `get()` принимает три параметра. Первый параметр — это указатель на массив символов, второй указывает максимальное число символов в строке с учетом концевого нулевого символа, добавляемого автоматически, и третий задает разделитель строк.

Если для второго параметра установлено значение 20, функция `get()` введет 19 символов и оборвет ввод строки, на которую указывал первый параметр, после чего добавит концевой нулевой символ. Третий параметр по умолчанию устанавливается как символ разрыва строки (`'\n'`). Если этот символ повстречается раньше, чем будет введен последний допустимый символ строки, функция вставит в этом месте концевой нулевой символ, но символ разрыва строки при этом останется в буфере и будет считан очередной функцией ввода.

Реализация этого метода ввода показана в листинге 16.6.

Листинг 16.6. Использование функции `get()` для заполнения массива символов

```
1:    // Листинг 16.6. Использование get()с массивом символов
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       char stringOne[256];
7:       char stringTwo[256];
8:
9:       cout << "Enter string one: ";
10:      cin.get(stringOne,256);
11:      cout << "stringOne: " << stringOne << endl;
12:
13:      cout << "Enter string two: ";
14:      cin >> stringTwo;
15:      cout << "StringTwo: " << stringTwo << endl;
16:    return 0;
17:   }
```

РЕЗУЛЬТАТ

```
Enter string one: Now is the time
stringOne: Now is the time
Enter string two: For all good
StringTwo: For
```

АНАЛИЗ В строках 6 и 7 создаются два массива символов. Строка 9 предлагает пользователю ввести строку, после чего в строке 10 вызывается функция `cin.get()` с тремя параметрами. Первый параметр ссылается на заполняемый массив символов, второй задает максимально возможное количество символов в строке с учетом нулевого концевого символа (`'\0'`). Третий параметр не установлен, и используется заданный по умолчанию символ разрыва строки.

Пользователь вводит строку `Now is the time`. Вся строка вместе с концевым нулевым символом помещается в массив `stringOne`.

Вторую строку пользователю предлагается ввести в строке 13, однако в этом случае уже используется оператор ввода. Поскольку он считывает строку до первого пробела, во втором случае в буфер заносится строка `Все`, что, конечно же, неправильно.

Один из способов решения этой проблемы заключается в использовании функции `getline()`, как показано в листинге 16.7.

Листинг 16.7. Использование функции `getline()`

```
1:    // Листинг 16.7. Использование getline()
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       char stringOne[256];
7:       char stringTwo[256];
8:       char stringThree[256];
9:
```

```
10:       cout << "Enter string one: ";
11:       cin.getline(stringOne,256);
12:       cout << "stringOne: " << stringOne << endl;
13:
14:       cout << "Enter string two: ";
15:       cin >> stringTwo;
16:       cout << "stringTwo: " << stringTwo << endl;
17:
18:       cout << "Enter string three: ";
19:       cin.getline(stringThree,256);
20:       cout << "stringThree: " << stringThree << endl;
21:     return 0;
22:    }
```

РЕЗУЛЬТАТ

```
Enter string one: one two three

stringOne: one two three
Enter string two: four five six
stringTwo: four
Enter string three: stringThree: five six
```

АНАЛИЗ

Этот пример требует детального исследования, поскольку возможны некоторые сюрпризы.

В строках 6–8 объявляются массивы символов. В строке 10 пользователю предлагается ввести строку текста, которая считывается функцией `getline()`. Аналогично функции `get()`, параметры `getline()` устанавливают буфер ввода и максимальное число символов. Однако, в отличие от `get()`, функция `getline()` считывает и удаляет из буфера символ разрыва строки. Как вы помните, функция `get()` воспринимает символ разрыва строк как разделитель и оставляет его в буфере ввода.

В строке 14 пользователю вновь предлагается ввести строку, которая теперь уже считывается оператором ввода. В нашем примере вводится строка `four five six`, после чего первое слово `four` присваивается переменной `stringTwo`. После отображения предложения `Enter string three:` снова вызывается функция `getline()`. Так как часть строки `five six` все еще находится в буфере ввода, она сразу считывается до символа новой строки. Функция `getline()` завершает свою работу, и строкой 20 выводится значение переменной `stringThree`.

В результате третья строка не вводится в программу, поскольку функция `getline()` возвращает часть строки, оставшуюся в буфере после операции ввода в строке 15, так как оператор >> считывает строку только до первого пробела и вставляет найденное слово в массив символов.

Как вы помните, можно использовать несколько вариантов перегруженной функции-члена `get()`. В первом варианте она не принимает никаких параметров и возвращает значение полученного символа. Во втором принимается ссылка на односимвольную переменную и возвращается объект `istream`. В третьей, последней версии в функцию `get()` устанавливаются массив символов, количество считываемых символов и символ разделения (которым по умолчанию является разрыв строки). Эта версия функции `get()` возвращает символы в массив либо до тех пор, пока не будет введено максимально возможное количество символов, либо до первого символа разрыва строки. Если функция `get()` встречает символ разрыва строки, ввод прерывается, а символ разрыва строки остается в буфере ввода.

Функция-член getline() также принимает три параметра: буфер ввода, число символов в строке с учетом концевого нулевого символа и символ разделения. Функция getline() действует аналогично описанной выше функции get(), но отличается от последней только тем, что не оставляет в буфере символ разрыва строки.

Использование функции cin.ignore()

В некоторых случаях возникает необходимость пропустить часть символов строки от начала до достижения конца строки (EOL) или конца файла (EOF). Именно этому и отвечает функция ignore(). Она принимает два параметра: число пропускаемых символов и символ разделения. Например, вызов функции ignore(80,'\n') приведет к пропуску 80 символов, если ранее не будет найден символ начала новой строки. Последний затем будет удален из буфера, после чего функция ignore() завершит свою работу. Использование функции ignore() показано в листинге 16.8.

Листинг 16.8. Использование функции ignore()

```
1:    // Листинг 16.8. Использование ignore()
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       char stringOne[255];
7:       char stringTwo[255];
8:
9:       cout << "Enter string one:";
10:      cin.get(stringOne,255);
11:      cout << "String one: " << stringOne << endl;
12:
13:      cout << "Enter string two: ";
14:      cin.getline(stringTwo,255);
15:      cout << "String two: " << stringTwo << endl;
16:
17:      cout << "\ n\ nNow try again...\ n";
18:
19:      cout << "Enter string one: ";
20:      cin.get(stringOne,255);
21:      cout << "String one: " << stringOne<< endl;
22:
23:      cin.ignore(255,'\ n');
24:
25:      cout << "Enter string two: ";
26:      cin.getline(stringTwo,255);
27:      cout << "String Two: " << stringTwo<< endl;
28:    return 0;
29:    }
```

РЕЗУЛЬТАТ

```
Enter string one: once upon a time
String one: once upon a time
Enter string two: String two:
```

РЕЗУЛЬТАТ

```
Now try again...
Enter string one: once upon a time
String one: once upon a time
Enter string two: there was a
String Two: there was a
```

АНАЛИЗ

В строках 6 и 7 создаются два массива символов. В строке 9 пользователю предлагается ввести строку. В нашем примере вводится строка `once upon a time`. Ввод завершается нажатием <Enter>. В строке 10 для считывания этой строки используется функция `get()`, которая присваивает эту строку переменной `stringOne` и останавливается на символе начала новой строки, оставляя его в буфере ввода.

В строке 13 пользователю еще раз предлагается ввести вторую строку, однако в этот раз функция `getline()` в строке 14 считывает символ разрыва строки, оставшийся в буфере, и сразу же завершает свою работу.

В строке 19 пользователю предлагается ввести первую строку. Однако в этом случае для пропуска символа разрыва строки используется функция `ignore()` (см. в листинге 16.8 строку 23). Таким образом, при вызове `getline()` строкой 26 буфер ввода пуст, и пользователь получает возможность ввести следующую строку.

Функции-члены peek() и putback()

Объект `cin` обладает двумя дополнительными методами, которые могут оказаться весьма полезными. Метод `peek()`просматривает, но не считывает очередной символ. Метод `putback()` вставляет символ в поток ввода. Использование этих методов показано в листинге 16.9.

Листинг 16.9. Использование функций `peek()` и `putback()`

```
1:    // Листинг 16.9. Использование peek() и putback()
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       char ch;
7:       cout << "enter a phrase: ";
8:       while ( cin.get(ch) )
9:       {
10:         if (ch == '!')
11:            cin.putback('$');
12:         else
13:            cout << ch;
14:         while (cin.peek() == '#')
15:            cin.ignore(1,'#');
16:      }
17:    return 0;
18:   }
```

РЕЗУЛЬТАТ

```
enter a phrase: Now!is#the!time#for!fun#!

Now$isthe$timefor$fun$
```

АНАЛИЗ

В строке 6 объявляется символьная переменная ch, а в строке 7 пользователю предлагается ввести строку текста. Назначение этой программы состоит в том, чтобы заменить все встречающиеся во введенной строке восклицательные знаки (!) знаком доллара ($) и удалить все символы (#).

Цикл while в теле функции main() программы прокручивается до тех пор, пока не будет возвращен символ конца файла (вводится комбинацией клавиш <Ctrl+C> в Windows или <Ctrl+Z> и <Ctrl+D> в MS DOS и UNIX соответственно). (Не забывайте, что функция cin.get() возвращает 0 в конце файла.) Если текущий символ оказывается восклицательным знаком, он отбрасывается, а в поток ввода функцией putback() возвращается символ $. Если же текущий символ не является восклицательным знаком, он выводится на экран. Если текущий символ оказывается #, то он пропускается функцией ignore().

Указанный подход далеко не самый эффективный способ решения подобных задач (более того, если символ # будет расположен в начале строки, то программа его пропустит). Но наша основная цель состояла в том, чтобы продемонстрировать работу функций putback() и ignore(). Впрочем, их использование достаточно просто и понятно.

ПРИМЕЧАНИЕ

Методы peek() и putback() обычно используются для синтаксического анализа строк. Необходимость в нем возникает, например, при создании компилятора.

Вывод данных с помощью cout

Ранее вы уже использовали объект cout вместе с перегруженным оператором вывода (<<) для выведения на экран строк, чисел и других данных. Этот объект позволяет также форматировать данные, выравнивать столбцы и выводить числовые значения в десятичном и шестнадцатеричном формате. Как это сделать, вы узнаете далее.

Очистка буфера вывода

Вы, вероятно, уже заметили, что использование endl приводит к очистке буфера вывода. Этот оператор вызывает функцию-член flush() объекта cout, которая и осуществляет очистку буфера. Вы можете напрямую вызывать метод flush(), либо вызвав функцию-член flush(), либо написав следующее выражение:

```
cout << flush
```

Указанный метод позволяет явно очистить буфер вывода на тот случай, если не вся информация из него была выведена на экран.

Функции-члены объекта cout

Аналогично тому, как мы обращались к методам объекта cin: get() и getline(), с объектом cout можно использовать функции put() и write().

Функция put() выводит один символ на стандартное устройство вывода. Так как эта функция возвращает ссылку на ostream, а cout является объектом ostream, есть возможность последовательного обращения к функции put() для вывода ряда значений, как и при вводе данных. Реализация этой возможности показана в листинге 16.10.

Листинг 16.10. Использование функции `put()`

```
1:    // Листинг 16.10. Использование put()
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       cout.put('H').put('e').put('l').put('l').put('o').put('\ n');
7:    return 0;
8:    }
```

РЕЗУЛЬТАТ Hello

ПРИМЕЧАНИЕ

> При запуске этой программы некоторые компиляторы не выведут заданное слово `Hello`. Если эта проблема коснется и вас, просто пропустите этот листинг и идите дальше.

АНАЛИЗ Строку 6 можно представить следующим образом: функция `cout.put('H')` выводит букву `H` на экран и возвращает объект `cout`. Оставшуюся часть выражения можно представить следующим образом:

```
cout.put('e').put('l').put('l').put('o').put('\ n');
```

Выводится буква `e`, после чего остается `cout.put('l')`. Таким образом, повторяется цикл, на каждом этапе которого выводится следующая бука и возвращается объект `cout`. После вывода последнего символа (`'\n'`) выполнение функции завершается.

Функция `write()` работает так же, как и оператор ввода (<<), но она принимает параметр, указывающий максимальное количество выводимых символов. Использование этой функции показано в листинге 16.11.

Листинг 16.11. Использование функции `write()`

```
1:    // Листинг 16.11. Использование write()
2:    #include <iostream.h>
3:    #include <string.h>
4:
5:    int main()
6:    {
7:       char One[] = "One if by land";
8:
9:
10:
11:      int fullLength = strlen(One);
12:      int tooShort = fullLength -4;
13:      int tooLong = fullLength + 6;
14:
15:      cout.write(One,fullLength) << "\ n";
```

```
16:       cout.write(One,tooShort) << "\ n";
17:       cout.write(One,tooLong) << "\ n";
18:     return 0;
19:   }
```

РЕЗУЛЬТАТ

```
One if by land
One if by
One if by land i?!
```

ПРИМЕЧАНИЕ

На вашем компьютере последняя строка вывода может выглядеть иначе.

АНАЛИЗ

В строке 7 создается массив символов для заданной строки текста. Длина введенного текста присваивается в строке 11 целочисленной переменной `fullLength`. Установленное значение переменной `tooShort` меньше этой длины на четыре единицы, а значение переменной `tooLong` больше на шесть.

В строке 15 с помощью функции `write()` выводится вся строка, поскольку в качестве первого параметра функции задается полная длина текстовой строки.

Строкой 16 вновь выводится строка, однако длина ее на четыре символа меньше, что и отражается в выводе.

Еще один вывод данных выполняется в строке 17, однако в этом случае функция `write()` выводит на шесть символов больше. После заданной строки на экране появятся символы, расположенные в ячейках памяти, следующих за ячейками массива символов.

Манипуляторы, флаги и команды форматирования

Поток вывода поддерживает установку большого количества флагов состояния, определяющих основание чисел (*десятичное или шестнадцатеричное*), ширину полей вывода и символы, используемые для заполнения полей. Флаг состояния представляет собой байт информации, каждый бит которого имеет специальное предназначение. Установка двоичных флагов более детально рассматривается на занятии 21. Для установки флагов потока `ostream` можно использовать функции-члены и манипуляторы.

Использование функции cout.width()

По умолчанию ширина поля вывода автоматически устанавливается такой, чтобы точно вместить все символы строки из буфера вывода. Но с помощью функции `width()` можно установить точное значение ширины поля вывода. Эта функция вызывается как метод объекта `cout`, поскольку является его функцией-членом. Функция `width()` изменяет ширину только следующего поля вывода. Использование этой функции проиллюстрировано в листинге 16.12.

Листинг 16.12. Настройка ширины поля вывода

```
1:    // Листинг 16.12. Настройка ширины поля вывода
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       cout << "Start >";
7:       cout.width(25);
8:       cout << 123 << "< End\ n";
9:
10:      cout << "Start >";
11:      cout.width(25);
12:      cout << 123<< "< Next >";
13:      cout << 456 << "< End\ n";
14:
15:      cout << "Start >";
16:      cout.width(4);
17:      cout << 123456 << "< End\ n";
18:
19:    return 0;
20:   }
```

РЕЗУЛЬТАТ

```
Start >                  123< End
Start >                  123< Next >456< End
Start >123456< End
```

АНАЛИЗ Сначала (строки 6–8) число 123 выводится в поле шириной в 25 символов. Ширина поля задается в строке 7. Результат этого форматирования показан в первой строке вывода.

Во второй строке вывода значение 123 распечатывается опять же в поле шириной 25, а затем сразу же выводится значение 456. Как видите, установка ширины поля применяется только первый раз, а для второго выражения с объектом cout уже не действует. Таким образом, установки функции width() применяются только к следующему выражению вывода данных.

В последней строке вывода видно, что установка ширины поля меньшей размера заносимого в него значения игнорируется программой. В этом случае ширина поля устанавливается равной размерам выводимых данных.

Установка символов заполнения

Обычно объект cout заполняет пробелами пустые позиции поля, заданные функцией width(), как было показано в приведенном выше примере. Однако иногда возникает необходимость заполнить пустые позиции другими символами, например звездочками (*). Для этого нужно использовать функцию fill(), в параметре которой указать символ заполнения. Использование функции fill() показано в листинге 16.13.

Листинг 16.13. Использование функции `fill()`

```
1:    // Листинг 16.13. Функция fill()
2:
3:    #include <iostream.h>
4:
5:    int main()
6:    {
7:       cout << "Start >";
8:       cout.width(25);
9:       cout << 123 << "< End\ n";
10:
11:
12:       cout << "Start >";
13:      cout.width(25);
14:      cout.fill('*');
15:      cout << 123 << "< End\ n";
16:   return 0;
17:   }
```

Результат

```
Start >                    123< End
Start >*******************123< End
```

Анализ Строки 7–9 переписаны из предыдущего листинга. То же можно сказать и о строках 12–15, однако в строке 14 этого листинга используется функция `fill('*')` для установки символа звездочки (*) в качестве символа заполнения, что наглядно отражается в выводе программы.

Установка флагов

Для отслеживания состояния объектов `iostream` используются флаги. Установку флагов осуществляют с помощью функции `setf()`, в качестве параметра которой используется одна из стандартных заранее установленных констант.

О состоянии объекта говорят в том случае, если режим использования некоторых или всех его данных может изменяться в ходе работы программы.

Например, можно изменить режим отображения чисел и запретить вывод на экран нулевых десятичных значений (чтобы число `20,00` выглядело как `20`). Для этого вызывается функция `setf(ios::showpoint)`.

Область видимости перечисления констант ограничена классом `iostream (ios)`, поэтому необходимо использовать явное указание имени константы с именем класса `ios::имяфлага`, например `ios::showpoint`.

Для добавления знака "плюс" (+) перед положительными значениями устанавливается флаг `ios::showpos`. Чтобы изменить выравнивание выводимых данных на экране влево, вправо и по центру поля вывода, используются флаги `ios::left`, `ios::right` и `ios::interval` соответственно.

Наконец, установка основания отображаемых числовых значений выполняется с помощью флагов `ios::dec` (десятичные числа), `ios::oct` (восьмеричные числа) или `ios::hex` (шестнадцатеричные числа). Эти флаги можно использовать в паре с оператором ввода (<<). Пример установки флагов показан в листинге 16.4.

Листинг 16.14. Установка флагов с помощью `setf`

```
1:    // Листинг 16.14. Использование функции setf
2:    #include <iostream.h>
3:    #include <iomanip.h>
4:
5:    int main()
6:    {
7:       const int number = 185;
8:       cout << "The number is " << number << endl;
9:
10:      cout << "The number is " << hex <<  number << endl;
11:
12:      cout.setf(ios::showbase);
13:      cout << "The number is " << hex <<  number << endl;
14:
15:      cout << "The number is " ;
16:      cout.width(10);
17:      cout << hex << number << endl;
18:
19:      cout << "The number is " ;
20:      cout.width(10);
21:      cout.setf(ios::left);
22:      cout << hex << number << endl;
23:
24:      cout << "The number is " ;
25:      cout.width(10);
26:      cout.setf(ios::internal);
27:      cout << hex << number << endl;
28:
29:      cout << "The number is:" << setw(10) << hex << number << endl;
30:    return 0;
31:   }
```

РЕЗУЛЬТАТ

```
The number is 185
The number is b9
The number is 0xb9
The number is         0xb9
The number is 0xb9
The number is 0x        b9
The number is:0x        b9
```

АНАЛИЗ В строке 7 целочисленная константа `number` инициируется значением `185`, которое выводится на экран в строке 8.

Это же значение выводится строкой 10, однако, поскольку здесь задействован манипулятор `hex`, оно отображается в шестнадцатеричном формате как `b9`. (Числу `b` в шестнадцатеричном коде соответствует `11` в десятичном. Умножение 11 на 16 дает 176. Добавив 9, получаем десятичное значение 185.)

В строке 12 установлен флаг showbase, что приводит к добавлению префикса 0x ко всем шестнадцатеричным значениям.

В строке 16 ширина поля устанавливается равной 10. Поэтому выводимое значение сдвинуто вправо. В строке 20 ширина также устанавливается равной 10, однако применяется выравнивание влево. Этот момент хорошо виден в выводе программы.

В строке 25 ширина остается равной 10, однако применяется выравнивание по ширине поля. Поэтому 0x вводится по левому краю поля, а b9 — по правому.

Наконец, в строке 29 повторяются те же установки, но в этот раз функция setw()используется не в отдельной строке, а в паре с оператором вывода (<<). Результат получается тот же.

Сравнение потоков и функции printf()

Большинство версий компиляторов C++ включают также стандартные библиотеки ввода-вывода языка C, позволяющие использовать для этого функцию printf(). Хотя использовать printf() немного проще, чем cout, применять ее не желательно.

Функция printf() не обеспечивает должного контроля за типами данных, поэтому можно легко ошибиться и отобразить число как символ или символ как число. Кроме того, функция printf() не поддерживает классы, поэтому ее трудно использовать для вывода данных объектов классов. Приходится задавать каждый член класса для printf() в отдельности.

С другой стороны, эта функция значительно упрощает форматирование выводимых данных, так как позволяет вставлять спецификаторы форматирования в качестве параметров функции. Поскольку функция printf() все еще эффективно применяется в некоторых программах и пользуется популярностью у многих программистов, этот раздел посвятим краткому обзору ее использования.

Для использования функции printf() необходимо включить в программу файл заголовка stdio.h. В самой простой форме функция printf() принимает в качестве параметра строку для форматированного вывода в виде текста, взятого в кавычки. Перед строкой могут быть установлены самые различные наборы спецификаторов форматирования. В табл. 16.1 показаны наиболее часто используемые спецификаторы преобразований типов, начинающиеся всегда с символа %.

Таблица 16.1. Спецификаторы преобразований типов

Спецификатор	*Используется для преобразования*
%s	В строку
%d	В число типа integer
%l	В число типа long integer
%ld	В число типа double
%f	В число типа float

Каждый спецификатор преобразования может также дополняться установкой общего числа знаков в выводимом значении и числа знаков после десятичной запятой. Эта установка имеет вид десятичного значения с плавающей точкой, где символы сле-

ва от точки устанавливают общее число знаков в выводимых значениях, а символы справа — число знаков после запятой. Например, спецификатор `%5d` задает вывод целочисленного значения длиной 5 знаков, а `%15.5f` — вывод числа с плавающей запятой общей длиной в 15 знаков, пять из которых составляют дробную часть. Различные способы использования `printf()` показаны в листинге 16.15.

Листинг 16.15. Вывод данных с помощью функции `printf()`

```
1:    #include <stdio.h>
2:    int main()
3:    {
4:       printf("%s","hello world\ n");
5:
6:       char *phrase = "Hello again!\ n";
7:       printf("%s",phrase);
8:
9:       int x = 5;
10:      printf("%d\ n",x);
11:
12:      char *phraseTwo = "Here's some values: ";
13:      char *phraseThree = " and also these: ";
14:      int y = 7, z = 35;
15:      long longVar = 98456;
16:      float floatVar =  8.8f;
17:
18:      printf("%s %d %d %s %ld %f\ n",phraseTwo,y,z, phraseThree,longVar,floatVar);
19:
20:      char *phraseFour = "Formatted: ";
21:      printf("%s %5d %10d  %10.5f\ n",phraseFour,y,z,floatVar);
22:    return 0;
23:   }
```

Результат

```
hello world
Hello again!
5
Here's some values: 7 35  and also these: 98456 8.800000
Formatted:      7        35     8.800000
```

Анализ Первый раз функция `printf()` вызывается в строке 4 и имеет стандартную форму: за именем функции `printf` следует спецификатор преобразования (в данном случае `%s`) и константная строка в кавычках, выводимая на экран.

Спецификатор `%s` указывает, что в данный момент выводится текстовая строка, указанная далее, — `"hello world"`.

Второй вызов функции `printf` в строке 7 аналогичен первому, но в данном случае вместо константной строки, заключенной в кавычки, используется указатель типа `char`.

В третьем вызове `printf()` в строке 10 используется спецификатор вывода целочисленного значения, хранимого в переменной `x`. Еще более сложным оказывается четвертый вариант вызова функции `printf()`, показанный в строке 18. Здесь выводится сразу шесть значений. Каждому приведенному спецификатору отвечает свое значение, отделенное от остальных с помощью запятых.

Наконец, в строке 21 в уже хорошо известной вам функции `printf()` используются спецификаторы форматирования, определяющие длину и точность выводимых значений. Многие считают, что форматирование вывода данных с помощью спецификаторов функции `printf()` намного проще, чем с помощью манипуляторов объекта `cout`.

Ранее уже отмечались основные недостатки функции `printf()` — отсутствие строгого контроля за типами данных и невозможность объявления этой функции как друга или метода класса. Поэтому при необходимости распечатать данные различных членов класса нужно использовать явно заданные методы доступа к членам класса.

Обобщение методов управления выводом данных в программах на C++

Для форматирования вывода данных в C++ можно использовать комбинации специальных символов, манипуляторов и флагов.

В выражениях с объектом `cout` используются следующие специальные символы:

- `\n` — новая строка;
- `\r` — возврат каретки;
- `\t` — табуляция;
- `\\` — обратный слеш;
- `\ddd` (число в восьмеричном коде) — символ *ASCII*;
- `\a` — звуковой сигнал (звонок).

Пример выражения вывода строки:

```
cout << "\aAn error occured\t"
```

Указанное выражение не только выводит сообщение об ошибке на экран компьютера, но подает предупреждающий звуковой сигнал и выполняет переход к следующей позиции табуляции. С оператором `cout` используются также манипуляторы. Однако для использования большинства манипуляторов нужно включить в программу файл `iomanip.h`.

Далее вашему вниманию представлен список манипуляторов, *не* требующих включения `iomanip.h`:

- `flush` — очищает буфер вывода;
- `endl` — вставляет символ разрыва строки и очищает буфер вывода;
- `oct` — устанавливает восьмеричное основание для выводимых чисел;
- `dec` — устанавливает десятичное основание для выводимых чисел;
- `hex` — устанавливает шестнадцатеричное основание для выводимых чисел.

А теперь приведем набор манипуляторов, для которых *необходимо* включение `iomanip.h`:

- `setbase (основание)` — устанавливает основание для выводимых чисел (0 = десятичная, 8 = восьмеричная, 10 = десятичная, 16 = шестнадцатеричная);
- `setw (ширина)` — устанавливает минимальную ширину поля вывода;
- `setfill (символ)` — устанавливает символ заполнения незанятых позиций поля вывода;
- `setprecision (точность)` — устанавливает число знаков после плавающей запятой;
- `setiosflags (флаг)` — устанавливает один или несколько флагов;

`resetiosflags (флаг)` — сбрасывает один или несколько флагов.

Например, в строке

```
cout << setw(12) << setfill ('#') << hex << x <<endl;
```

устанавливается ширина поля в 12 знаков, символ заполнения `#`, восьмеричное основание выводимых чисел, после чего выводится значение переменной `x`, добавляется символ разрыва строки и очищается буфер. Все манипуляторы, за исключением `flush`, `endl` и `setw`, остаются включенными на протяжении всей работы программы, если, конечно, не будут сделаны другие установки. Установка манипулятора `setw` отменяется сразу же после текущего вывода с объектом `cout`.

С манипуляторами `setiosflags` и `resetiosflags` могут использоваться следующие ios-флаги:

`ios::left` — выравнивает данные по левому краю поля вывода;

`ios::right` — выравнивает данные по правому краю поля вывода;

`ios::interval` — выравнивает данные по ширине поля вывода;

`ios::dec` —выводит данные в десятичном формате;

`ios::oct` — выводит данные в восьмеричном формате;

`ios::hex` — выводит данные в шестнадцатеричном формате;

`ios::showbase` — добавляет префикс `0x` к шестнадцатеричным значениям и `0` к восьмеричным значениям;

`ios::showpoint` — заполняет нулями недостающие знаки в значениях заданной длины;

`ios::uppercase` — отображает в верхнем регистре шестнадцатеричные и экспоненциальные значения;

`ios::showpos` — добавляет знак '+' перед положительными числами;

`ios::scientific` — отображает числа с плавающей запятой в экспоненциальном представлении;

`ios::fixed` — отображает числа с плавающей запятой в шестнадцатеричном представлении.

Дополнительную информацию можно получить из файла `ios.h` или из справочной системы компилятора.

Использование файлов для ввода и вывода данных

Потоки C++ обеспечивают универсальные методы обработки данных, поступающих с клавиатуры или диска, а также выводимых на экран и диск. В любом случае можно использовать либо операторы ввода и вывода, либо другие стандартные функции и манипуляторы. Дальнейшие разделы главы посвящены операциям открытия и закрытия файлов, которые сопровождаются созданием объектов `ifstream` и `ofstream`.

Объекты ofstream

Объекты, создаваемые для считывания или записи данных в файл, называются `ofstream`. Они являются производными от уже знакомого вам класса `iostream`.

Чтобы приступить к записи в файл, нужно сначала создать объект `ofstream`, а затем связать его с определенным файлом на диске. Использование объектов `ofstream` требует включения в программу файла заголовка `fstream.h`.

Поскольку fstream содержит в себе iostream.h, нет необходимости в отдельном включении файла iostream.h.

Состояния условий

Объектами iostream поддерживаются флаги, отражающие состояние ввода и вывода. Значение каждого из этих флагов можно проверить с помощью функций, возвращающих TRUE или FALSE: eof(), bad(), fail() и good(). Функция eof() возвращает значение TRUE, если в объекте iostream встретился символ EOF (end of file — конец файла). Функция bad() возвращает значение TRUE при попытке выполнить ошибочную операцию. Функция fail() возвращает значение TRUE каждый раз, когда это же значение возвращает функция bad(), а также в тех случаях, когда операция невыполнима в данных условиях. Наконец, функция good() возвращает значение TRUE, когда все идет хорошо, т.е. все остальные функции возвращают значение FALSE.

Открытие файлов для ввода-вывода

Для открытия файла myfile.cpp с помощью объекта ofstream нужно объявить экземпляр этого объекта, передав ему в качестве параметра имя файла:

```
ofstream fout("myfile.cpp");
```

Открытие файла для ввода выполняется аналогичным образом, за тем исключением, что для этого используется объект ifstream:

```
ifstream fin("myfile.cpp");
```

Обратите внимание, что в выражениях задаются имена объектов fout и fin, которые можно использовать так же, как объекты cout и cin соответственно.

Очень важным методом, используемым в файловых потоках, является функция-член close(). Каждый создаваемый вами объект файлового потока открывает файл для чтения или записи (или и для того и другого сразу). По завершении работы файл необходимо закрыть с помощью функции close(), чтобы впоследствии не повредить его и записанные в нем данные.

После связывания объектов потока с соответствующими файлами их можно использовать так же, как остальные объекты ввода-вывода. Пример использования объектов для обмена данными с файлами показан в листинге 16.16.

Листинг 16.16. Открытие файла для чтения и записи

```
1:    #include <fstream.h>
2:    int main()
3:    {
4:       char fileName[80];
5:       char buffer[255];    // для ввода данных пользователем
6:       cout << "File name: ";
7:       cin >> fileName;
8:
9:       ofstream fout(fileName);  // открытие файла для записи
10:      fout << "This line written directly to the file...\ n";
```

```
11:      cout << "Enter text for the file: ";
12:      cin.ignore(1,'\ n');       // пропускает символ разрыва строки после имени файла
13:      cin.getline(buffer,255);   // принимает данные, введенные пользователем,
14:      fout << buffer << "\ n";   // и записывает их в файл
15:      fout.close();              // закрывает файл, после чего его вновь можно открыть
16:
17:      ifstream fin(fileName);    // открывается тот же файл для чтения
18:      cout << "Here's the contents of the file:\ n";
19:      char ch;
20:      while (fin.get(ch))
21:         cout << ch;
22:
23:      cout << "\ n***End of file contents.***\ n";
24:
25:      fin.close();               // не забудь закрыть файл в конце программы
26:    return 0;
27:  }
```

РЕЗУЛЬТАТ

```
File name: test1
Enter text for the file: This text is written to the file!
Here's the contents of the file:
This line written directly to the file...
This text is written to the file!

***End of file contents.***
```

АНАЛИЗ В строке 4 создается массив для записи имени файла, а в строке 5 — еще один массив для временного хранения информации, вводимой пользователем. В строке 6 пользователю предлагается ввести имя файла, которое записывается в массив `fileName`. В строке 9 создается объект `ofstream` с именем `fout`, который связывается с введенным ранее именем файла. В результате происходит открытие файла. Если файл с таким именем уже существует, содержащаяся в нем информация будет замещена.

Строкой 10 введенный текст записывается прямо в файл, а в строке 11 пользователю вновь предлагается ввести новый текст. Символ разрыва строки, оставшийся в буфере после ввода имени файла, удаляется строкой 12, после чего все введенные пользователем данные записываются в массив в строке 13. Введенный текст записывается в файл вместе с символом разрыва строки, а затем в строке 15 этот файл закрывается.

В строке 17 файл открывается заново, но в этот раз для чтения, и его содержимое посимвольно вводится в программу в строках 20–21.

Настройка открытия файла объектом ofstream

По умолчанию при связывании объекта `ofstream` с именем файла создается новый файл с указанным именем, если таковой не существует, или удаляется содержимое уже существующего файла с таким же именем. Чтобы изменить установки по умолчанию, используется второй аргумент конструктора объекта `ofstream`.

Для второго аргумента можно устанавливать следующие константные значения:

- `ios::app` — добавляет данные в конец файла вместо удаления всего содержимого файла;
- `ios::ate` — переводит точку ввода в конец файла, но у вас есть возможность вводить новые данные в любом месте файла;

- ios::trunc — устанавливается по умолчанию; полностью удаляет (отбрасывает) текущее содержимое файла;
- ios::nocreate — если файл не существует, операция открытия не выполняется;
- ios::noreplace — если файл уже существует, операция открытия не выполняется.

Имена констант являются аббревиатурами выполняемых действий: app — *apend* (добавить), ate — *at end* (в конец), trunc — *truncate* (отбросить) и т.п.

Листинг 16.17 является модификацией листинга 16.16 с установкой опции добавления данных в файл при его повторном открытии.

Листинг 16.17. Добавление данных в конец файла

```
1:    #include <fstream.h>
2:    int main()   // возвращает 1 в случае ошибки
3:    {
4:       char fileName[80];
5:       char buffer[255];
6:       cout << "Please re-enter the file name: ";
7:       cin >> fileName;
8:
9:       ifstream fin(fileName);
10:      if (fin)                // файл уже существует?
11:      {
12:         cout << "Current file contents:\ n";
13:         char ch;
14:         while (fin.get(ch))
15:            cout << ch;
16:         cout << "\ n***End of file contents.***\ n";
17:      }
18:      fin.close();
19:
20:      cout << "\ nOpening " << fileName << " in append mode...\ n";
21:
22:      ofstream fout(fileName,ios::app);
23:      if (!fout)
24:      {
25:         cout << "Unable to open " << fileName << " for appending.\ n";
26:         return(1);
27:      }
28:
29:      cout << "\ nEnter text for the file: ";
30:      cin.ignore(1,'\ n');
31:      cin.getline(buffer,255);
32:      fout << buffer << "\ n";
33:      fout.close();
34:
35:      fin.open(fileName);  // переопределение существующего объекта fin!
36:      if (!fin)
37:      {
38:         cout << "Unable to open " << fileName << " for reading.\ n";
39:         return(1);
```

```
40:       }
41:       cout << "\ nHere's the contents of the file:\ n";
42:       char ch;
43:       while (fin.get(ch))
44:          cout << ch;
45:       cout << "\ n***End of file contents.***\ n";
46:       fin.close();
47:       return 0;
48:    }
```

РЕЗУЛЬТАТ

```
Please re-enter the file name: test1
Current file contents:
This line written directly to the file...
This text is written to the file!

***End of file contents.***
Opening test1 in append mode...

Enter text for the file: More text for the file!

Here's the contents of the file:
This line written directly to the file...
This text is written to the file!
More text for the file!

***End of file contents.***
```

АНАЛИЗ Пользователю вновь предлагается ввести имя файла, после чего в строке 9 создается объект файлового потока ввода. В строке 10 проверяется наличие на диске указанного файла и, если он уже существует, его содержимое выводится на экран строками 12–16. Обратите внимание на то, что выражение `if(fin)` аналогично `if(fin.good())`.

Файл ввода закрывается и снова открывается, однако теперь в режиме добавления (строка 22). После этого открытия (как, впрочем, после каждого открытия) выполняется проверка правильности открытия файла. В этом случае условие `if(!fout)` подобно условию `if (fout.fail())`. Пользователю предлагается ввести текст, после чего в строке 33 файл закрывается.

Наконец, как и в листинге 16.16, файл открывается в режиме чтения, но в этом случае не нужно повторно объявлять объект `fin`. Он просто связывается с тем же именем файла. После проверки правильности открытия файла в строке 36 содержимое файла выводится на экран и он окончательно закрывается.

Рекомендуется

Постоянно проверяйте правильность открытия файла.

Повторно используйте уже существующие объекты `ifstream` и `ofstream`.

Закрывайте все объекты `fstream` по завершении работы с ними.

Не рекомендуется

Не пытайтесь закрыть или переопределить объекты `cin` и `cout`.

Двоичные и текстовые файлы

Некоторые операционные системы, например DOS, различают текстовые и двоичные файлы. В первых все данные хранятся в виде текста (в кодах ASCII). Числовые значения, например 54321, хранятся в виде строки ('5','4','3','2','1'). Возможно это не совсем удобно, однако упрощает считывание информации многими простыми программами для DOS.

Чтобы помочь файловой системе отличить текстовый формат файла от двоичного, язык программирования C++ предоставляет флаг `ios::binary`. Во многих системах этот флаг игнорируется, поскольку все данные хранятся в двоичном формате. А в некоторых закрытых системах этот флаг вообще запрещен и не поддается компиляции!

В двоичных файлах могут храниться не только числа и строки, но и целые информационные структуры. Весь блок данных можно вывести сразу, используя метод `write()` объекта `fstream`.

Записав данные с помощью `write()`, можно возвратить эти данные обратно с помощью метода `read()`. В качестве параметра эти функции-члены ожидают получить указатель на символ, поэтому перед использованием функции необходимо привести адрес класса к указателю на строку символов.

Второй аргумент этих функций задает количество записываемых символов. Это значение можно определить с помощью функции `sizeof()`. Запомните, что записываются данные, а не методы. Соответственно и считываются только данные. В листинге 16.18 показано, как записать содержимое класса в файл,

Листинг 16.18. Запись класса в файл

```
1:    #include <fstream.h>
2:
3:    class Animal
4:    {
5:    public:
6:       Animal(int weight, long days):itsWeight(weight), itsNumberDaysAlive(days){ }
7:       ~Animal(){ }
8:
9:       int GetWeight()const { return itsWeight; }
10:      void SetWeight(int weight) { itsWeight = weight; }
11:
12:      long GetDaysAlive()const { return itsNumberDaysAlive; }
13:      void SetDaysAlive(long days) { itsNumberDaysAlive = days; }
14:
15:   private:
16:      int itsWeight;
17:      long itsNumberDaysAlive;
18:   };
19:
20:   int main()   // returns 1 on error
21:   {
22:      char fileName[80];
23:
24:
```

```
25:       cout << "Please enter the file name: ";
26:       cin >> fileName;
27:       ofstream fout(fileName,ios::binary);
28:       if (!fout)
29:       {
30:          cout << "Unable to open " << fileName << " for writing.\ n";
31:          return(1);
32:       }
33:
34:       Animal Bear(50,100);
35:       fout.write((char*) &Bear,sizeof Bear);
36:
37:       fout.close();
38:
39:       ifstream fin(fileName,ios::binary);
40:       if (!fin)
41:       {
42:          cout << "Unable to open " << fileName << " for reading.\ n";
43:          return(1);
44:       }
45:
46:       Animal BearTwo(1,1);
47:
48:       cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
49:       cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
50:
51:       fin.read((char*) &BearTwo, sizeof BearTwo);
52:
53:       cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
54:       cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
55:       fin.close();
56:       return 0;
57:    }
```

РЕЗУЛЬТАТ

```
Please enter the file name: Animals
BearTwo weight: 1
BearTwo days: 1
BearTwo weight: 50
BearTwo days: 100
```

АНАЛИЗ В строках 3–18 объявляется класс `Animal`. В строках 22–32 создается файл, который открывается для вывода в двоичном режиме. В строке 34 создается объект `Animal` со значениями переменных-членов `itsWeight` = 50 и `itsNumberDaysAlive` = 100. В следующей строке данные объекта заносятся в файл.

В строке 37 файл закрывается, после чего повторно открывается для чтения в двоичном режиме в строке 39. Создается второй объект `Animal`, значения обоих переменных-членов которого равны 1. В строке 51 данные из файла считываются в новый объект `Animal`, замещая собой текущие значения объекта.

Установка параметров ввода-вывода с помощью командной строки

Многие операционные системы, такие как DOS и UNIX, позволяют пользователю выполнять установки некоторых параметров при запуске программы. Эти установки называются опциями командной строки и, как правило, отделяются друг от друга пробелами, например:

```
SomeProgram Param1 Param2 Param3
```

Эти параметры не передаются напрямую в функцию `main()`. Вместо этого функция `main()`программы может принимать два других параметра. Первый — это целочисленное значение, указывающее число аргументов командной строки с учетом имени программы. Поэтому минимальное значение этого параметра равно единице (задается по умолчанию). Для показанной выше командной строки значение параметра будет равно четырем. (Имя `SomeProgram` плюс три параметра в сумме дают четыре аргумента командной строки.)

Второй параметр, передаваемый функции `main()`, — это массив указателей на строки символов. Так как имя массива является постоянным указателем на первый элемент массива, можно объявить этот аргумент как указатель на указатель типа `char`, указатель на массив символов или массив массивов символов.

Обычно первый аргумент называется `argc` (argument count — количество аргументов), однако вы можете присвоить ему любое имя, которое вам нравится. Второй аргумент зачастую называется `argv` (argument vector — вектор аргументов), однако это имя также не является обязательным.

Как правило, с помощью `argc` проверяется количество установленных аргументов коммандной строки, после чего для доступа к ним используется `argv`. Обратите внимание: `argv[0]` — это имя программы, а `argv[1]` — первый аргумент коммандной строки. Если программа принимает в качестве аргументов два числовых значения, нужно будет преобразовать их в строки. На занятии 21 вы узнаете, как выполнить это преобразование с помощью средств, предоставляемых стандартными библиотеками функций. В листинге 16.19 показан пример использования аргументов командной строки.

Листинг 16.19. Использование аргументов командной строки

```
1:    #include <iostream.h>
2:    int main(int argc, char **argv)
3:    {
4:       cout << "Received " << argc << " arguments...\ n";
5:       for (int i=0; i<argc; i++)
6:          cout << "argument " << i << ": " << argv[i] << endl;
7:    return 0;
8:    }
```

РЕЗУЛЬТАТ

```
TestProgram  Teach Yourself C++ In 21 Days
Received 7 arguments...
argument 0: TestProgram.exe
argument 1: Teach
```

РЕЗУЛЬТАТ

```
argument 2: Yourself
argument 3: C++
argument 4: In
argument 5: 21
argument 6: Days
```

Вам придется либо запустить этот код из командной строки DOS, либо установить параметры командной строки с помощью компилятора (см. документацию компилятора).

АНАЛИЗ

В функции `main()` объявляются два аргумента: `argc` — целочисленное значение, указывающее число аргументов командной строки, и `argv` — указатель на массив строк. Каждый элемент этого массива представляет аргумент командной строки. Обратите внимание, `argv` можно также объявить как `char *argv[]` или `char[][]`. Программист может выбрать вариант, который ему более по душе. Даже если в программе этот аргумент будет объявлен как указатель на указатель, для доступа к определенным элементам можно воспользоваться индексом смещения элемента от начала массива.

В строке 4 массив `argv` используется для вывода числа установленных аргументов командной строки. Всего их оказалось семь, включая имя программы.

В строках 5 и 6 задается цикл `for`, который выводит значения всех аргументов командной строки по отдельности, обращаясь к ним по имени массива `argv` с указанием смещения `[i]`. Для вывода значений аргументов используется объект `cout`.

Листинг 16.20 является переписанной версией листинга 16.18, в которой имя файла задается как аргумент командной строки.

Листинг 16.20. Использование аргументов командной строки

```
1:    #include <fstream.h>
2:
3:    class Animal
4:    {
5:    public:
6:       Animal(int weight, long days):itsWeight(weight), itsNumberDaysAlive(days){ }
7:       ~Animal(){ }
8:
9:       int GetWeight()const {  return itsWeight; }
10:      void SetWeight(int weight) {  itsWeight = weight; }
11:
12:      long GetDaysAlive()const {  return  itsNumberDaysAlive; }
13:      void SetDaysAlive(long days) {  itsNumberDaysAlive = days; }
14:
15:   private:
16:      int itsWeight;
17:      long itsNumberDaysAlive;
18:   } ;
19:
20:    int main(int argc, char *argv[])   // возвращает 1 в случае ошибки
21:    {
```

```
22:       if (argc != 2)
23:       {
24:          cout << "Usage: " << argv[0] << " <filename>" << endl;
25:          return(1);
26:       }
27:
28:       ofstream fout(argv[1],ios::binary);
29:       if (!fout)
30:       {
31:          cout << "Unable to open " << argv[1] << " for writing.\ n";
32:          return(1);
33:       }
34:
35:       Animal Bear(50,100);
36:       fout.write((char*) &Bear,sizeof Bear);
37:
38:       fout.close();
39:
40:       ifstream fin(argv[1],ios::binary);
41:       if (!fin)
42:       {
43:          cout << "Unable to open " << argv[1] << " for reading.\ n";
44:          return(1);
45:       }
46:
47:       Animal BearTwo(1,1);
48:
49:       cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
50:       cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
51:
52:       fin.read((char*) &BearTwo, sizeof BearTwo);
53:
54:       cout << "BearTwo weight: " << BearTwo.GetWeight() << endl;
55:       cout << "BearTwo days: " << BearTwo.GetDaysAlive() << endl;
56:       fin.close();
57:       return 0;
58:    }
```

РЕЗУЛЬТАТ

```
BearTwo weight: 1
BearTwo days: 1
BearTwo weight: 50
BearTwo days: 100
```

АНАЛИЗ Объявление класса `Animal` аналогично представленному в листинге 16.18. Однако в этом случае пользователю не предлагается ввести имя файла, а используется аргумент командной строки. В строке 2 объявляется функция `main()`, принимающая два параметра: количество аргументов командной строки и указатель на массив символов, в котором сохраняются аргументы командной строки.

В строках 22–26 проверяется, соответствует ли установленное число аргументов ожидаемому. Если пользователь забыл ввести имя файла, то выводится сообщение об ошибке:

```
Usage TestProgram <имяфайла>
```

После этого программа завершает свою работу. Обратите внимание, что при выводе имени программы используется не константная строка, а значение `argv[0]` . Данное выражение будет правильно выводить имя программы, даже если оно будет изменено после компиляции.

В строке 28 программа пытается открыть двоичный файл с указанным именем. Однако, вместо того чтобы копировать и хранить имя файла во временном массиве, как это было в листинге 16.18, его можно задать в командной строке и затем возвратить из `argv[1]`.

Точно так же имя файла возвращается в строке 40, где этот файл открывается для ввода данных, и в строках 25 и 31 при формировании сообщений об ошибках открытия файлов.

Резюме

Сегодня вы познакомились с потоками и глобальными объектами `cout` и `cin`. Основное предназначение объектов `istream` и `ostream` состоит в инкапсулировании буферизированого ввода и вывода данных на стандартные устройства ввода-вывода.

В каждой программе создается четыре стандартных потоковых объекта: `cout`, `cin`, `cerr` и `clog`. Однако в большинстве операционных систем эти объекты можно переадресовывать.

Объект `cin` класса `istream` используется для ввода данных обычно вместе с перегружаемым оператором ввода (>>). Объект `cout` класса `ostream` используется для вывода данных в комбинации с оператором вывода (<<).

Стандартные объекты ввода-вывода включают много других функций-членов, например `get()` и `put()`. Поскольку эти методы возвращают ссылки на объект потока, несколько вызовов функций можно объединять в одном выражении.

Для настройки работы объектов потока используются манипуляторы. С их помощью можно устанавливать не только опции форматирования и отображения, но и многие другие атрибуты объектов потока.

Обмен данными с файлами осуществляется с помощью классов `fstream`, производных от класса `iostream`. Кроме обычных операторов ввода и вывода, эти классы поддерживают использование функций `read()` и `write()`, позволяющих считывать и записывать целые объекты в двоичные файлы.

Вопросы и ответы

Как определить, когда использовать операторы ввода и вывода, а когда другие функции-члены классов потока?

В целом операторы ввода и вывода проще в использовании, поэтому в большинстве случаев лучше обращаться именно к ним. В некоторых других случаях, когда эти операторы не справляются со своей работой (например, при вводе строки из слов, разделенных пробелами), можно прибегнуть к использованию других функций.

Какое отличие между `cerr` и `clog`?

Объект `cerr` не буферизируется? Другими словами, все данные, поступающие в `cerr`, немедленно выводятся на экран. Это отлично подходит для вывода ошибок на

экран, однако дорого обойдется при записи регистрационной информации на диск. Объект clog буферизирует свой вывод, поэтому в последнем случае может быть более эффективным.

Зачем создавать потоки, если отлично работает функция `printf()`**?**

Функция `printf()` не контролирует строго типы выводимых данных, чего требуют стандарты C++. Кроме того, эта функция не поддреживает работу с классами.

Когда следует применять метод `putback()`**?**

Этот метод весьма эффективен в тех случаях, когда для определения соответствия введенного символа установленным ограничениям используется одна операция считывания, а для записи символа в буфер используются некоторые другие операций. Наиболее часто это находит применение при анализе синтаксических конструкций файла, например при создании компиляторов.

Когда следует использовать функцию `ignore()`**?**

Наиболее часто она используется после функции `get()`. Поскольку последняя оставляет в буфере символ разрыва строки, иногда за вызовом функции `get()` следует вызов `ignore(1, '\n');`. Эта функция, как и `putback()`, используется, как правило, при синтаксическом разборе файлов.

Мои друзья используют в своих программах на C++ функцию `printf()`**. Можно ли и мне ее использовать?**

Конечно же, можно. Однако, хотя эта функция более проста в использовании, вы утратите строгий контроль за типами файлов и затрудните работу с объектами классов.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний и приводится несколько упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Что такое оператор ввода и как он работает?
2. Что такое оператор вывода и как он работает?
3. Перечислите три варианта перегруженной функции `cin.get()` и укажите основные их отличия.
4. Чем `cin.read()` отличается от `cin.getline()`?
5. Какая ширина устанавливается по умолчанию для вывода длинных целых чисел с помощью оператора вывода?
6. Какое значение возвращает оператор вывода?
7. Какой параметр принимается конструктором объекта `ofstream`?
8. Что устанавливает аргумент `ios::ate`?

Упражнения

1. Напишите программу, использующую четыре стандартных объекта класса `iostream` — `cin`, `cout`, `cerr` и `clog`.
2. Напишите программу, предлагающую пользователю ввести свое полное имя с последующим выводом этого имени на экран.
3. Перепишите листинг 16.9, отказавшись от использования методов `putback()` и `ignore()`.
4. Напишите программу, считывающую имя файла в качестве аргумента командной строки и открывающую файл для чтения. Разработайте алгоритм анализа всех символов, хранящихся в файле, и выведите на экран только текстовые символы и знаки препинания (пропускайте все непечатаемые символы). Закройте файл перед завершением работы программы.
5. Напишите программу, которая выведет заданные аргументы командной строки в обратном порядке, отбросив имя программы.

Пространства имен

Одним из дополнений стандарта ANSI C++ является возможность использования программистами пространств имен, позволяющих избежать конфликтов имен при работе с большим количеством библиотек. Сегодня вы узнаете:

- Какие функции и классы вызываются по имени
- Как создаются пространства имен
- Как используются пространства имен
- Как используется стандартное пространство имен `std`

Введение

Конфликты имен возникают из-за недомолвок между разработчиками C и C++. Стандарты ANSI предлагают способ решения этой проблемы с помощью *пространств имен* (namespaces). Однако следует проявлять осторожность, так как не все компиляторы поддерживают это средство.

Конфликт имен возникает в тех случаях, когда в двух частях программы находятся подобные имена с совпадающими областями видимости. Наиболее часто это случается при использовании различных пакетов библиотек. Например, в разных библиотеках классов контейнеров часто объявляется и используется класс `List`. (Более подробно классы контейнеров рассматриваются на занятии 19.)

Тот же класс `List` используется и в библиотеках окон. Предположим, необходимо реализовать набор окон для приложения и применить класс `List` из библиотеки классов контейнеров. Для этого объявляется экземпляр класса `List` из библиотеки окон, чтобы поддержать работу окон приложения. Однако в результате может оказаться, что его функции-члены недоступны, поскольку компилятор автоматически связал объявленный класс с аналогичным классом `List` из стандартной библиотеки классов контейнеров, который вам вовсе не был нужен.

Пространство имени используется для разделения глобальных пространств имен, чтобы исключить или, по крайней мере, уменьшить количество конфликтов имен. Пространства имен весьма похожи на классы, в том числе и синтаксисом.

Объявленные внутри пространства имени элементы принадлежат к этому пространству, но являются открытыми. Пространства имен могут взаимно перекрываться.

Соответственно и функции могут объявляться как внутри, так и за пределами пространств имен. В последнем случае при вызове такой функции следует явно указывать соответствующее пространство имен.

Вызов по имени функций и классов

В процессе анализа кода программы и создания списка имен функций и переменных компилятор проверяет программу на наличие конфликтов имен. Конфликты, которые сам компилятор решить не в состоянии, могут устраняться компоновщиком.

Компилятор не в состоянии проверить конфликты имен в единицах трансляции (например, файлах объектов). Эта задача решается компоновщиком приложений. Поэтому компилятор не покажет даже предупреждение.

Довольно часто компоновщик выдает сообщение об ошибке `Identifier multiply defined` (множественное объявление идентификатора). Это сообщение появится в том случае, если вы попытаетесь описать идентификаторы с одинаковыми именами и перекрывающимися областями видимости. Если два идентификатора будут объявлены с общими областями видимости в одном файле источника, то об ошибке сообщит компилятор. Сообщение об ошибке поступит от компоновщика при попытке скомпилировать и связать следующий код программы:

```
// файл first.cpp
int integerValue = 0 ;
int main( ) {
    int integerValue = 0 ;
    // . . .
    return 0 ;
} ;

// файл second.cpp
int integerValue = 0 ;
// конец second.cpp
```

Компоновщик выдает сообщение `in second.obj: integerValue already defined in first.obj` (`integerValue` из `second.obj` уже объявлен в `first.obj`). Если бы эти имена располагались в разных областях видимости, то компилятор и компоновщик не имели бы ничего против.

Может поступить и такое предупреждение от компилятора: `identifier hiding` (идентификатор скрыт). Суть его состоит в том, что в файле `first.cpp` объявление переменной `integerValue` в функции `main()` скрывает глобальную переменную с таким же именем.

Чтобы использовать в функции `main()` глобальную переменную `integerVaalue`, объявленную за пределами `main()`, необходимо явно указать глобальность этой переменной с помощью оператора видимости (`::`). Так, в следующем примере значение `10` будет присвоено глобальной переменной `integerValue`, а не переменной с таким же именем, объявленной внутри `main()`:

```
// файл first.cpp
int integerValue = 0 ;
int main( )
{
    int integerValue = 0 ;
    ::integerValue = 10 ; //присваиваем глобальной переменной integerValue
```

```
    // . . .
    return 0 ;
}  ;

// файл second.cpp
int integerValue = 0 ;
// конец second.cpp
```

Обратите внимание на использование оператора видимости (::), который необходим для указания глобальности переменной integerValue в том случае, если в функции была объявлена переменная с таким же именем.

Проблема с двумя глобальными переменными, объявленными за пределами какой-либо функции, заключается в том, что они имеют одинаковые имена и перекрывающиеся области видимости и это вызывает ошибку в работе компоновщика.

Под *видимостью* объекта, который может быть переменной, классом или функцией, понимают ту часть программы, в которой данный объект может использоваться. Например, переменная, объявленная и определенная за пределами всякой функции, имеет *файловую*, или *глобальную* область видимости. Ее видимость распространяется от точки объявления до конца файла. Переменная, имеющая *модульную*, или *локальную* область видимости, объявляется внутри программного модуля. Чаще всего локальные переменные объявляются в теле функции. Ниже показаны примеры объектов с различными областями видимости:

```
int globalScopeInt  = 5 ;
void f( )
{
    int localScopeInt = 10 ;
}
int main( )
{
    int localScopeInt = 15 ;
    {
        int anotherLocal = 20 ;
        int localScopeInt = 30 ;
    }
    return 0 ;
}
```

Первая целочисленная переменная GlobalScopeInt будет видна как внутри функции f(), так и main(). В теле функции f() содержится объявление переменной localScopeInt. Ее область видимости локальна, т.е. ограничивается пределами модуля, содержащего объявление функции.

Функция main() не может получить доступ к переменной localScopeInt функции f(). Как только завершается выполнение функции f(), переменная localScopeInt удаляется из памяти компьютера. Объявление третьей переменной, также названной localScopeInt, располагается в теле функции main(). Область ее видимости также локальна.

Обратите внимание: переменная localScopeInt функции main() не конфликтует с одноименной переменной функции f(). Видимость следующих двух переменных — anotherLocal и localScopeInt — также ограничена областью модуля. Другими словами, эти переменные видны от места объявления до закрывающей фигурной скобки, ограничивающей тело модуля, в котором эта функция была объявлена.

Вы, наверное, обратили внимание, что в программе объявляются две одноименные локальные переменные localScopeInt, причем одна из них объявляется во внешнем модуле, а вторая — во вложенном. Таким образом, их области видимости перекрываются. Переменная, объявленная во внутреннем модуле, будет скрывать в нем переменную внешнего модуля. После закрытия фигурной скобки внутреннего модуля вторая переменная localScopeInt из внешнего модуля вновь становится видимой. Все изменения, внесенные в localScopeInt внутри фигурных скобок, никоим образом не повлияют на значение внешней переменной localScopeInt.

Имена могут иметь *внутреннюю* или *внешнюю связь*. Оба эти термина относятся к использованию или доступности имени в нескольких или одной программной единице. На всякое имеющее внешнюю связь имя можно ссылать только в пределах определяющей его единицы. Например, переменная, имеющая внутреннюю связь, может использоваться функциями только внутри блока программы, где эта переменная была объявлена. Имена с внешними связями доступны функциям *из других блоков. Примеры внутренних и внешних связей* иллюстрирует приведенный ниже код.

```
// файл: first.cpp
int externalInt = 5 ;
const int j = 10 ;
int main()
{
    return 0 ;
}

// файл: second.cpp
extern int externalInt ;
int anExternalInt = 10 ;
const int j = 10 ;
```

Переменная externalInt, объявленная в файле first.cpp, имеет внешнюю связь. Несмотря на то что она объявлена в файле first.cpp, доступ к этой переменной можно получить и из файла second.cpp. В обоих файлах также есть константы j, которые по умолчанию имеют внутренние связи. Чтобы изменить заданную по умолчанию внутреннюю связь констант, необходимо явно указать их глобальность, как это сделано в следующем примере:

```
// файл: first.cpp
extern const int j = 10 ;

// файл: second.cpp
extern const int j ;
#include <iostream>
int main()
```

```
{
    std::cout << "j = " << j << std::endl ;
    return 0 ;
}
```

Обратите внимание на использование обозначения пространства имени `std` перед `cout`, что позволяет ссылаться на все объекты стандартной библиотеки ANSI. После выполнения этого кода на экране появится строка:

```
j = 10
```

Комитет по стандартизации не рекомендует использовать `static` для ограничения области видимости внешней переменной, как в следующем примере:

```
static int staticInt = 10 ;
int main()
{
    //...
}
```

Если сейчас такое использование `static` просто не рекомендуется, то в будущем подобное выражение вообще может рассматриваться как ошибочное. Поэтому уже сейчас вместо `static` лучше использовать пространства имен.

Рекомендуется	Не рекомендуется
Используйте пространства имен.	**Не применяйте** ключевое слово `static` для ограничения области видимости переменной пределами файла.

Создание пространства имени

Синтаксис объявления пространства имен аналогичен синтаксису объявления структур и классов. После ключевого слова `namespace` стоит имя пространства имен, которое может и отсутствовать, а затем следует открывающая фигурная скобка. Пространство имен завершается закрывающей фигурной скобкой без точки с запятой в конце выражения. Например:

```
namespace Window
{
    void move( int x, int y) ;
}
```

Имя `Window` идентифицирует пространство имен. Можно создавать множество экземпляров именованных пространств имен, расположенных внутри одного файла или в разных единицах трансляции. Примером тому может быть пространство имен `std` стандартной библиотеки C++. Его использование обосновано в данном случае тем, что стандартная библиотека представляет собой логически единую группу функций.

Основное назначение пространств имен состоит в группировании связанных элементов в именованной области программы. Ниже показан пример пространства имен, объединяющего несколько файлов заголовков:

```
// header1.h
namespace Window
{
    void move( int x, int y) ;
}

// header2.h
namespace Window
{
    void resize( int x, int y ) ;
}
```

Объявление и определение типов

Внутри пространства имен можно объявлять и определять типы и функции. Тут не обойтись без обсуждения стратегических подходов программирования в C++. Правильность структуры программы определяется тем, насколько четко отделен интерфейс программы от ее процедурной части. Этому принципу необходимо следовать не только при работе с классами, но и при создании пространств имен. Ниже показан пример плохо структурированного пространства имен:

```
namespace Window {
    // . . . другие обьявления и определения переменных.
    void move( int x, int y) ; // объявления
    void resize( int x, int y ) ;
    // . . . другие обьявления и определения переменных.

    void move( int x, int y )
    {
        if( x < MAX_SCREEN_X  && x > 0 )
            if( y < MAX_SCREEN_Y  && y > 0 )
                platform.move( x, y ) ; // специальная программа
    }

    void resize( int x, int y )
    {
        if( x < MAX_SIZE_X  && x > 0 )
            if( y < MAX_SIZE_Y  && y > 0 )
                platform.resize( x, y ) ; // специальная программа
    }
    // . . . продолжение определений
}
```

Наглядно видно, как быстро пространство имен становится хаотичным и беспорядочным! Причем в этом примере объявление пространства имен составляло всего около 20 строк, а во что превратилась бы программа, будь объявление более длинным?

Объявление функций за пределами пространства имени

Функции пространства имен следует объявлять за пределами тела пространства. Это позволит явно отделить объявления функций от определения их выполнения, не захламляя тело пространства имен. Кроме того, вынос объявлений функции даст возможность разместить пространство имен и его внедренные объявления в файле заголовка, а определения выполнения поместить в исполняемый файл программы. Например:

```
// файл header.h
namespace Window {
    void move( int x, int y) ;
    // другие объявления
}

// file impl.cpp
void Window::move( int x, int y )
{
    // код перемещения окна
}
```

Добавление новых членов

Добавление новых членов в пространство имен осуществляет только в теле пространства. Невозможно создавать новые члены пространства имен вне тела пространства, указывая его имя, как это делалось с объектами классов. Компилятор ответит на это сообщением об ошибке. Пример такой ошибки показан ниже.

```
namespace Window {
    // ряд объявлений
}
// код программы
int Window::newIntegerInNamespace ; // ошибка
```

Последняя строка неправильна, и компилятор сообщит об этом. Чтобы исправить ошибку, перенесите объявление переменной-члена `newIntegerInNamespace` в тело пространства имен.

Все заключенные в пространстве имени члены являются открытыми. Поэтому неправильным будет и следующий код:

```
namespace Window {
    private:
        void move( int x, int y ) ;
}
```

Вложения пространств имен

Одно пространство имен можно вложить в другое пространство имен. К подобному вложению прибегают в том случае, когда определение выполнения одного пространства имен должно содержать объявление нового пространства. Чтобы обратиться

к члену внутреннего пространства имен, необходимо явно указать имена обоих пространств. Так, в следующем примере одно именованное пространство объявляется внутри другого именованного пространства:

```
namespace Window {
    namespace Pane {
        void size( int x, int y ) ;
    }
}
```

Для доступа к функции `size()` за пределами пространства имен `Window` нужно дополнить имя вызываемой функции именами пространств имен, внутри которых она была объявлена, например:

```
int main( )
{
    Window::Pane::size( 10, 20 ) ;
    return 0 ;
}
```

Использование пространств имен

Теперь рассмотрим пример использования пространства имен и связанного с ним оператора видимости. Сначала внутри пространства имен `Window` объявляются все типы и функции, после чего за его пределами следуют определения функций-членов. Чтобы определить функцию, объявленную в пространстве имен, следует перед именем функции установить имя пространства имен и оператор видимости, как это делается в листинге 17.1.

Листинг 17.1. Использование пространства имен

```
1: #include <iostream>
 2:
 3: namespace Window
 4: {
 5:     const int MAX_X = 30 ;
 6:     const int MAX_Y = 40 ;
 7:     class Pane
 8:     {
 9:         public:
10:             Pane() ;
11:             ~Pane() ;
12:             void size( int x, int y ) ;
13:             void move( int x, int y ) ;
14:             void show( ) ;
15:         private:
16:             static int cnt ;
17:             int x ;
18:             int y ;
19:     } ;
20: }
```

```
21:
22: int Window::Pane::cnt = 0 ;
23: Window::Pane::Pane() : x(0), y(0) {  }
24: Window::Pane::~Pane() {  }
25:
26: void Window::Pane::size( int x, int y )
27: {
28:     if( x < Window::MAX_X  &&  x > 0 )
29:         Pane::x = x ;
30:     if( y < Window::MAX_Y  &&  y > 0 )
31:         Pane::y = y ;
32: }
33: void Window::Pane::move( int x, int y )
34: {
35:     if( x < Window::MAX_X  &&  x > 0 )
36:         Pane::x = x ;
37:     if( y < Window::MAX_Y  &&  y > 0 )
38:         Pane::y = y ;
39: }
40: void Window::Pane::show( )
41: {
42:     std::cout << "x " << Pane::x ;
43:     std::cout << " y " << Pane::y << std::endl ;
44: }
45:
46: int main( )
47: {
48:     Window::Pane pane ;
49:
50:     pane.move( 20, 20 ) ;
51:     pane.show( ) ;
52:
53:     return 0 ;
54: }
```

РЕЗУЛЬТАТ x 20 y 20

АНАЛИЗ Обратите внимание, что класс `Pane` вложен в пространство имен `Window`. Поэтому при обращении к объектам класса `Pane` их имена дополняются идентификатором `Window::`.

Статическая переменная-член `cnt`, объявленная в строке 16 внутри класса `Pane`, определяется как обычно. Но при определении функции-члена `Pane::size()` и обращениях к переменным-членам `MAX_X` и `MAX_Y` в строках 26–32 используется явное указание пространства имен. Дело в том, что статическая переменная-член определяется внутри класса `Pane`, а определения других функций-членов (это же справедливо для функции `Pane::move()`) происходят как за пределами класса, так и вне тела пространства имен. Без явного указания пространства имен компилятор покажет сообщение об ошибке.

Обратите внимание также на то, что внутри определений функций-членов обращение к объявленным переменным-членам класса происходит с явным указанием имени класса: `Pane::x` и `Pane::y`. Зачем это делается? Дело в том, что у вас возникли бы проблемы, если функция `Pane::move()` определялась бы следующим образом:

```
void Window::Pane::move( int x, int y )
{
    if( x < Window::MAX_X  &&  x > 0 )
        x = x ;
    if( y < Window::MAX_Y  &&  y > 0 )
        y = y ;
    Platform::move( x, y ) ;
}
```

Ну что, догадались, в чем проблема? Опасность состоит в том, что компилятор в этом выражении никаких ошибок не заметит.

Источник проблемы заключается в аргументах функции. Аргументы `x` и `y` скроют закрытые переменные-члены `x` и `y`, объявленные в классе `Pane`, поэтому вместо присвоения значений аргументов переменным-членам произойдет присвоение этих значений самим себе. Чтобы исправить эту ошибку, необходимо явно указать переменные-члены класса:

```
Pane::x = x;
Pane::y = y;
```

Ключевое слово using

Ключевое слово `using` может использоваться и как оператор, и в качестве спецификатора при объявлении членов пространства имен, но синтаксис использования `using` при этом меняется.

Использование using как оператора

С помощью ключевого слова `using` расширяются области видимости всех членов пространства имен. Впоследствии это позволяет ссылаться на члены пространства имен, не указывая соответствующее имя пространства. Использование `using` показано в следующем примере:

```
namespace Window {
    int value1 = 20 ;
    int value2 = 40 ;
}
. . .
Window::value1 = 10 ;

using namespace Window ;
value2 = 30 ;
```

Все члены пространства имен `Window` становятся видимыми, начиная от строки `using namespace Window;` и до конца соответствующего модуля программы. Обратите внимание, что если для обращения к переменной `value1` в верхней части фрагмента

программы необходимо указывать пространство имен, то в этом нет необходимости при обращении к переменной value2, поскольку оператор using сделал видимыми все члены пространства имен Window.

Оператор using может использовать в любом модуле программы с различной областью видимости. Когда выполнение программы выходит за область видимости данного модуля, автоматически становятся невидимыми все члены пространства имен, открытые в этом модуле. Проанализируйте это на следующем примере:

```
namespace Window {
    int value1 = 20 ;
    int value2 = 40 ;
}
//. . .
void f()
{
    {
        using namespace Window ;
        value2 = 30 ;
    }
    value2 = 20 ; //ошибка!
}
```

Последняя строка кода функции f() — value2 = 20 — вызовет ошибку во время компиляции, поскольку переменная value2 в этом месте невидима. Видимость этой переменной, заданная оператором using, закончилась сразу за закрывающими фигурными скобками в предыдущей строке программы.

В случае объявления внутри модуля локальных переменных все одноименные переменные пространства имен, открытые в этом модуле, будут скрыты. Это аналогично сокрытию глобальных переменных локальными в случае совпадения их областей видимости. Даже если переменная, объявленная в пространстве имен, будет открыта с помощью using после объявления локальной переменной, последняя все равно будет иметь приоритет. Это наглядно показано в следующем примере:

```
namespace Window {
    int value1 = 20 ;
    int value2 = 40 ;
}
//. . .
void f()
{
   int value2 = 10 ;
   using namespace Window ;
   std::cout << value2 << std::endl  ;
}
```

При выполнения этой функции на экране появится значение 10, а не 40, подтверждая тот факт, что переменная value2 пространства имен Window скрывается переменной value2 функции f(). Если все же требуется использовать переменную пространства имен, явно укажите имя пространства.

При использовании одноименных идентификаторов, один из которых объявлен как глобальный, а другой — внутри пространства имен, также может возникнуть двусмысленность. Чтобы избежать ее, всегда явно указывайте имя пространства при вызове объекта, как в следующем фрагменте программы:

```
namespace Window {
    int value1 = 20 ;
}
//. . .
using namespace Window ;
int value1 = 10 ;
void f( )
{
   value1 = 10 ;
}
```

В данном примере неопределенность возникает внутри функции `f()`. Оператор `using` сообщает переменной `Window::value1` глобальную область видимости. Однако в программе объявляется другая глобальная переменная с таким же именем. Какая из них используется в функции `f()`? Обратите внимание, что ошибка будет показана не во время объявления одноименной глобальной переменной, а при обращении к ней в теле функции `f()`.

Использование using в объявлениях

Назначение `using` в объявлениях идентификаторов аналогично использованию `using` как оператора с той лишь разницей, что обеспечивается более высокий уровень контроля. Этот способ используется для открытия видимости только для одного идентификатора, объявленного в пространстве имен, как показано в следующем примере:

```
namespace Window {
    int value1 = 20 ;
    int value2 = 40 ;
    int value3 = 60 ;
}
//. . .
using Window::value2 ; //открытие доступа к value2 в текущем модуле
Window::value1 = 10 ;  //для value1 необходимо указание пространства имен
value2 = 30 ;
Window::value3 = 10 ;  // для value3 необходимо указание пространства имен
```

Итак, с помощью `using` можно открыть доступ в текущую область видимости к отдельному идентификатору пространства имен, не повлияв на остальные идентификаторы, заданные в этом пространстве. В предыдущем примере переменная `value2` вызывается без явного указания пространства имен, что невозможно при обращении к `value1` и `value3`. Использование `using` при объявлении обеспечивает дополнительный контроль над видимостью каждого идентификатора пространства имен. В этом и заключается отличие от использования `using` как оператора, открывающего доступ сразу ко всем идентификаторам пространства имен.

Видимость имени распространяется до конца блока, что можно сказать и о любом другом объявлении. С помощью `using` идентификаторы можно объявлять как глобально, так и в локальной области.

Если в локальную область, где уже объявлен идентификатор из пространства имен, добавляется другой идентификатор с таким же именем, это приводит к ошибке компиляции. Ошибкой будет и объявление идентификатора из пространства имен в области, где уже существует другой идентификатор с таким же именем. Это показано в следующем примере:

```
namespace Window {
    int value1 = 20 ;
    int value2 = 40 ;
}
//. . .
void f()
{
   int value2 = 10 ;
   using Window::value2 ; // ряд объявлений
   std::cout << value2 << std::endl  ;
}
```

Компиляция второй строки функции `f()` приведет к ошибке, поскольку переменная с именем `value2` в этом блоке уже объявлена. Тот же результат получится, если объявление с `using` разместить перед объявлением локальной переменной `value2`.

Идентификатор пространства имен, введенный в локальную область с помощью `using`, скрывает аналогичный идентификатор, объявленный за пределами этой области. Проанализируйте следующий пример:

```
namespace Window {
    int value1 = 20 ;
    int value2 = 40 ;
}
int value2 = 10 ;
//. . .
void f()
{
    using Window::value2 ;
    std::cout << value2 << std::endl  ;
}
```

Объявление переменной с помощью `using` в функции `f()` скрывает глобальную переменную `value2`.

Как отмечалось ранее, этот способ использования `using` позволяет дополнительно контролировать области видимости отдельных идентификаторов пространства имен. Оператор `using` открывает доступ в локальной области ко всем идентификаторам, объявленным в пространстве имен. Поэтому предпочтительней использовать `using` в объявлениях, а не как оператор, чтобы в полной мере воспользоваться всеми преимуществами, предоставляемыми пространством имени. Явное расширение области видимости для отдельных идентификаторов позволяет снизить вероятность возникновения конфликтов имен. Использование оператора `using` оправдано только в том случае, если необходимо открыть доступ сразу ко всем идентификаторам пространства имен.

Псевдонимы пространства имен

Псевдонимы пространства имен используется для создания дополнительного имени именованного пространства. Как правило, псевдоним представляет собой информативный термин, используемый для ссылки на пространство имен. Это весьма эффективно, если имя пространства очень длинное. Создание псевдонимов поможет упростить дальнейшую работу с пространствами имен. Рассмотрим следующий пример:

```
namespace the_software_company {
    int value ;
    // . . .
}
the_software_company::value = 10 ;
. . .
namespace TSC = the_software_company ;
TSC::value = 20 ;
```

Недостаток этого метода состоит в том, что подобный псевдоним может уже существовать в указанной области видимости. В этом случае компилятор сообщит об ошибке, и вы сможете просто изменить псевдоним.

Неименованные пространства имен

Такие пространства имен отличаются от именованных тем, что не имеют имени. Наиболее часто они используются для защиты глобальных данных от потенциальных конфликтов имен. Каждая единица программы имеет собственное уникальное неименованное пространство. Все идентификаторы, объявленные внутри такого пространства имен, вызываются просто по имени без каких-либо префиксов. В следующем коде представлены примеры двух неименованных пространств, расположенных в двух разных файлах.

```
// файл: one.cpp
namespace {
    int value ;
    char p( char *p ) ;
    //. . .
}

// файл: two.cpp
namespace {
    int value ;
    char p( char *p ) ;
    //. . .
}
int main( )
{
    char c = p( ptr ) ;
}
```

В каждом файле объявляется переменная `value` и функция `p()`. Благодаря тому что для каждого файла задано свое неименованное пространство, обращения к одноименным идентификаторам внутри файлов одной программы не приводит к конфликтам

имен. Это хорошо видно при вызове функции p(). Функционирование неименованного пространства имен аналогично работе статического объекта с внешней связью, такого как

```
static int value = 10 ;
```

Не забывайте, что подобное использование ключевого слова static не рекомендуется комитетом по стандартизации. Для решения подобных задач теперь используются пространства имен. Можно провести еще одну аналогию с неименованными пространствами: они очень похожи на глобальные переменные с внутренней связью.

Стандартное пространство имен std

Наилучший пример пространств имен можно найти в стандартной библиотеке C++. Все функции, классы, объекты и шаблоны стандартной библиотеки объявлены внутри пространства имен std.

Вероятно, вам приходилось видеть подобные выражения:

```
#include <iostream>
using namespace std ;
```

Не забывайте, что использование директивы using открывает доступ ко всем идентификаторам именованного пространства имен. Поэтому лучше не обращаться к помощи данного оператора при работе со стандартными библиотеками. Почему? Да потому, что таким образом вы нарушите основное предназначение пространств имен. Глобальное пространство будет буквально заполнено именами различных идентификаторов из файлов заголовков стандартной библиотеки, большая часть которых не используется в данной программе. Помните, что во всех файлах заголовков используется средство пространства имен, поэтому, если вы включите в программу несколько файлов заголовков и используете оператор using, все идентификаторы, объявленные в этих файлах заголовков, получат глобальную видимость. Вы могли заметить, что в большинстве примеров данной книги это правило нарушается. Это сделано исключительно для краткости изложения примеров. Вам же следует использовать в своих программах объявления с ключевым словом using, как в следующем примере:

```
#include <iostream>
using std::cin ;
using std::cout ;
using std::endl ;
int main( )
{
    int value = 0 ;
cout << "So, how many eggs did you say you wanted?" << endl ;
    cin >> value ;
    cout << value << " eggs, sunny-side up!" << endl ;
    return( 0 ) ;
}
```

Выполнение этой программы приведет к следующему выводу:

```
So, how many eggs did you say you wanted?
4
4 eggs, sunny-side up!
```

В качестве альтернативы можно явно обращаться к идентификаторам, объявленным в пространстве имен:

```
#include <iostream>
int main( )
{
    int value = 0 ;
std::cout << "How many eggs did you want?" << std::endl ;
    std::cin >> value ;
    std::cout << value << " eggs, sunny-side up!" << std::endl ;
    return( 0 ) ;
}
```

Программа выведет следующие данные:

```
How many eggs did you want?
4
4 eggs, sunny-side up!
```

Такой подход вполне годится для небольшой программы, но в больших приложениях будет довольно сложно проследить за всеми явными обращениями к идентификаторам пространства имен. Только представьте себе: вам придется добавлять `std::` для каждого имени из стандартной библиотеки!

Резюме

Создать пространство имени так же просто, как описать класс. Есть несколько различий, но они весьма незначительны. Во-первых, после закрывающей фигурной скобки пространства имен не следует точка с запятой. Во-вторых, пространство имен всегда открыто, в то время как класс закрыт. Это означает, что вы можете продолжить объявление пространства имен в других файлах или в разных местах одного файла.

Вставлять в пространство имен можно все, что подлежит объявлению. Создавая классы для своей будущей библиотеки, вам следует взять на вооружение средство пространства имен. Объявленные внутри пространства имен функции должны определяться за его пределами. Благодаря этому интерфейс программы отделяется от ее выполнения.

Можно вкладывать одно пространство имен в другое. Однако не забывайте, что при обращении к членам внутреннего пространства имен необходимо явно указывать имена внешнего и внутреннего пространств.

Для открытия доступа ко всем членам пространства имен в текущей области видимости используется оператор `using`. Однако в результате этого слишком много идентификаторов могут получить глобальную видимость, что чревато конфликтами имен. Поэтому использование данного оператора не относится к правилам хорошего тона, особенно при работе со стандартными библиотеками. Вместо этого воспользуйтесь ключевым словом `using` при объявлении в программе идентификаторов из пространства имен.

Ключевое слово `using` в объявлении идентификатора используется для открытия доступа в текущей области видимости только к отдельному идентификатору из пространства имен, что существенно снижает вероятность возникновения конфликтов имен.

Псевдонимы пространств имен аналогичны оператору `typedef`. С их помощью можно создавать дополнительные имена для именованных пространств, что оказывается весьма полезным, если исходное имя длинное и неудобное.

Неименованное пространство может содержаться в каждом файле. Как следует из названия, это пространство без имени. Описав неименованное пространство имен с помощью ключевого слова namespace, можно использовать одноименные идентификаторы в разных файлах программы. Благодаря неименованному пространству имена переменных становятся локальными для текущего файла. Неименованные пространства имен рекомендуется использовать вместо ключевого слова static.

В стандартной библиотеке C++ используется пространство имен std. Однако избегайте использования оператора using, открывающего доступ ко всем идентификаторам стандартной библиотеки. Воспользуйтесь лучше объявлениями с ключевым словом using.

Вопросы и ответы

Обязательно ли использовать пространства имен?

Нет, вы можете писать простые программы и без помощи пространств имен. Просто убедитесь, что вы используете старые стандартные библиотеки (например, `#include <string.h>`), а не новые (например, `#include <cstring.h>`).

Каковы отличия между двумя способами использования ключевого слова `using`?

Ключевое слово using можно использовать как оператор и как спецификатор описания. В первом случае открывается доступ ко всем идентификаторам пространства имен. Во втором — доступ можно открыть только для отдельных идентификаторов.

Что такое неименованные пространства имен и зачем они нужны?

Неименованными называются пространства имен, для которых не задано собственное имя. Они используются для защиты наборов идентификаторов в разных файлах одной программы от возможных конфликтов имен. Идентификаторы неименованных пространств не могут использоваться за пределами области видимости их пространства имен.

Контрольные вопросы

1. Можно ли использовать идентификаторы, объявленные в пространстве имен, без применения ключевого слова using?
2. Назовите основные отличия между именованными и неименованными пространствами имен.
3. Что такое стандартное пространство имен std?

Упражнения

1. **Жучки**: найдите ошибку в следующем коде:

```
#include <iostream>

int main()
{
    cout << "Hello world!" << end;
    return 0;
}
```

2. Перечислите три способа устранения ошибки, найденной в коде упражнения 1.

Анализ и проектирование объектно-ориентированных программ

Углубившись в синтаксис C++, легко упустить из виду, как и зачем используются различные подходы и средства программирования. Сегодня вы узнаете:

- Как проводить анализ проблем и поиск решений, основываясь на подходах объектно-ориентированного программирования
- Как проектировать эффективные объектно-ориентированные программы для нахождения оптимальных решений поставленных задач
- Как использовать унифицированный язык моделирования (UML) для документирования анализа и проектирования

Является ли C++ объектно-ориентированным языком программирования

Язык C++ был создан как связующее звено между новыми принципами объектно-ориентированного программирования и одним из самых популярных в мире языком программирования C для разработки коммерческих программ. Для реализации назревших идей объектного программирования требовалась разработка надежной и эффективной среды программирования.

Язык C был разработан как нечто среднее между языками высокого уровня для бизнес-приложений, такими как COBOL, и работающим на уровне "железа", высокоэффективным, но трудным в использовании языком ассемблер. Язык C разрабатывался для реализации структурного программирования, при котором решение задачи разбивается на более мелкие рутинные единицы повторяющихся действий, называемых *процедурами*.

Программы, которые создавались в конце девяностых, принципиально отличаются от написанных в начале десятилетия. Программами, основанными на процедурных подходах, обычно трудно управлять, их тяжело поддерживать и модернизировать. Графические ин-

терфейсы пользователя, Internet, выход на телефонные линии по цифровым каналам и масса новых технологий резко увеличили сложность проектов, при этом ожидания потребителей относительно качества интерфейса пользователя также постоянно росли.

Под натиском такого стремительного усложнения программ разработчики вынуждены были заняться поиском новых подходов программирования. Старые процедурные подходы все более отставали от требований сегодняшнего дня. Программы быстро устаревали, а модернизация процедурной программы проходила не проще, чем разработка новой. По мере увеличения размеров программ значительно усложнялась их отладка. Проекты часто устаревали еще до того, как попадали на рынок. Расходы на поддержку и модернизацию этих проектов превышали доходы от их реализации.

Таким образом, внедрение в жизнь новых подходов объектно-ориентированного программирования было не прихотью программистов, а единственным спасением. Объектно-ориентированные языки программирования создают прочную связь между структурами данных и методами, обрабатывающими эти данные. Более важно то, что при таком программировании нет необходимости думать о том, как хранятся и обрабатываются данные в отдельных модулях. Программист просто узнает из интерфейса объекта, какие данные ему нужно передать и что он возвращает, после чего использует готовый модуль в своей программе.

Что же представляют собой виртуальные объекты? Сравним их с предметами и объектами окружающего мира: автомобилями, собаками, деревьями, облаками, цветами. Каждый из них имеет свои характеристики: быстрый, дружелюбный, коричневый, густой, красивый. Кроме того, множеству объектов свойственно определенное поведение: они движутся, лают, растут, проливаются дождем, увядают. Под словом "собака" большинство из нас понимают не совокупность пищеварительной, нервной и прочих систем, что может заинтересовать лишь узких специалистов, а мохнатого друга с четырьмя лапами, приветливо машущего хвостом и заливающегося звонким лаем. Для нас важны внешние признаки и поведение собаки, а не ее внутреннее "устройство".

Построение моделей

Чтобы отследить все признаки и связи объекта окружающего мира, нам пришлось бы создавать модель вселенной, настолько в этом мире все взаимосвязано. Целью модели является создание осмысленной абстракции реального мира. Такая абстракция должна быть проще самого мира, но при этом отображать его достаточно точно, чтобы модель можно было использовать для предсказания поведения предметов в реальном мире.

Классической моделью является детский глобус. Модель — это не сам предмет; мы никогда не спутаем детский глобус с планетой Земля, но первое настолько хорошо отображает второе, что мы можем познавать Землю, изучая глобус.

Конечно, здесь имеются существенные упрощения. На глобусе моей дочери никогда не бывает дождей, наводнений, "глобусотрясений" и т.п., но я могу его использовать, например, для того, чтобы рассчитать, сколько понадобится времени для полета от дома до Москвы. Это может потребоваться, скажем, при планировании времени и расходов на командировку.

Модель, которая не будет проще моделируемого предмета, бесполезна. Стивен Райт (Steven Wright) пошутил на эту тему: "У меня есть карта, где один дюйм равен дюйму. Я живу на E5".

Создание хорошей объектно-ориентированной программы по сути своей является моделированием реальных объектов средствами программирования. Для создания такой виртуальной модели важно хорошо знать, во-первых, средства программирования и, во-вторых, последовательность построения программы с помощью этих средств.

Проектирование программ: язык моделирования

Язык моделирования — это, по сути, фикция, набор соглашений по поводу принципов предварительного моделирования программы на бумаге. Тем не менее без этого этапа невозможно создать эффективный профессиональный программный продукт. Давайте договоримся изображать классы на бумаге в виде треугольников, а отношения наследования между ними — в виде пунктирных стрелок от базового класса к производному. Для примера смоделируем класс `Geranium` (Герань), произведенный от класса `Flower` (Цветок), как показано на рис. 18.1.

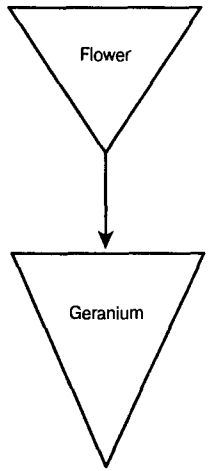


Рис. 18.1. Схематическое изображение наследования класса

На рисунке видно, что `Geranium` — особый вид `Flower`, и это вполне соответствует действительности. Если мы с вами договоримся графически изображать таким способом наследования классов, то будем прекрасно понимать друг друга. Со временем мы, вероятно, захотим моделировать многие сложные отношения и разработаем свой набор соглашений и правил по созданию диаграмм, отображающих взаимосвязи объектов программы.

Конечно, нам также придется довести эти соглашения до сведения других сотрудников, которые работают или будут работать вместе с нами над общим проектом. Возможно, мы будем взаимодействовать с другими фирмами, имеющими свои соглашения, и надо будет потратить время, чтобы выработать общие принципы, позволяющие избежать возможных недоразумений.

В таком случае было бы полезным существование единого языка моделирования, понятного для всех. (В действительности эту прекрасную идею реализовать ничуть не проще, чем заставить всех жителей Земли говорить на эсперанто.) Тем не менее такой язык был создан, и имя ему — UML (Unified Modeling Language — унифицированный язык моделирования). Его задача состоит в том, чтобы добиться единообразия в отображении взаимоотношений между объектами в диаграммах. В соответствии с соглашениями языка UML нашу схему, представленную на рис. 18.1, следовало бы изобразить иначе (рис. 18.2).

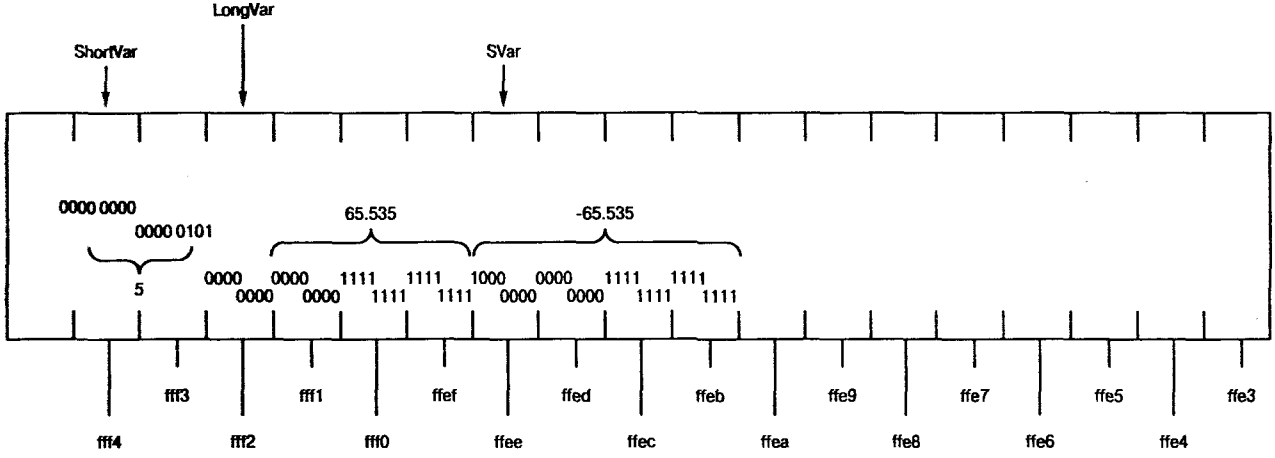


Рис. 18.2. Те же отношения наследования, но с учетом соглашений UML

В соответствии с соглашениями UML классы изображаются в виде прямоугольников, а наследование — в виде стрелки, направленной от производного класса к базовому. Направление стрелки противоречит тому, что подсказывает интуиция большинства из нас, но это не страшно: когда мы все договоримся, система заработает как надо.

Соглашения UML совсем несложные. Диаграммы нетрудно понимать и использовать. Мы рассмотрим эти соглашения на конкретных примерах в ходе освоения этой главы, что гораздо проще изучения UML вне контекста. Хотя этой теме можно посвятить целую книгу, по правде говоря, это будет пустая трата времени и бумаги, поскольку язык UML вполне соответствует принципу: лучше один раз увидеть на практике, чем десять раз прочитать.

Процесс проектирования программ

Правильное выполнение анализа и проектирования объектно-ориентированной программы намного важнее, чем соблюдение соглашений языка моделирования. Именно этой тематики посвящено большинство публикаций и конференций. И если по поводу языка моделирования удалось прийти к общим соглашениям и выработать UML, то споры по поводу основополагающих принципов анализа и проектирования программ продолжаются по сей день.

Появилась даже новая профессия — *методологи*: это программисты, которые изучают и разрабатывают методы программирования. Часто в литературе можно встретить статьи, посвященные описанию нового метода программирования. *Метод* — это совокупность языка моделирования и подходов анализа и проектирования. Три наиболее известных методолога в мире — это Грейди Буч (Grady Booch), создавший метод Буча, Айвер Якобсон (Ivar Jacobson), разработавший подходы объектно-ориентированного программирования, и Джеймс Рамбо (James Rumbaugh), создавший технологию объектного моделирования. Вместе они создали метод *Objectory* — коммерческий продукт от фирмы Rational Software, Inc. Это фирма, в которой они работают и где их любовно величают *"три амигос"*.

Материал, изложенный на этом занятии, приблизительно следует методам *Objectory*. Точного соответствия не будет, так как я не верю в рабское следование академической теории. Я считаю создание конкурентноспособной профессиональной программы более важным, чем точное соответствие этой программы каким бы то ни было абстрактным методам. В конце концов на *Objectory* свет клином не сошелся, и я рекомендую вам быть эклектиками и выбирать все лучшее из всех методов, которые вам известны.

Процесс проектирования программ *итеративен*. Это значит, что при разработке программы мы периодически повторяем весь процесс, по мере того как растет понимание требований. Проект нацелен на решение задачи, но нюансы, возникающие в ходе поиска оптимального решения, воздействуют на сам проект. Невозможно разработать серьезный крупный проект идя по прямой от начала до конца. Вместо этого на отдельных этапах приходится возвращаться к началу, постоянно совершенствуя интерфейсы и процедуры выполнения отдельных объектов.

Итеративную разработку следует отличать от каскадной, при которой выход из одной стадии становится входом в следующую и назад дороги нет (рис. 18.3). Этот процесс напоминает конвейер сборки автомобилей, где на каждом этапе собирается и тестируется один узел, постепенно формируя автомобиль. Но программа, в отличие от автомобиля, продукт штучный. Разработку программ редко удается поставить на конвейер.

При итеративном проектировании теоретик предлагает новую идею, а прикладник начинает творческую реализацию этой абстрактной идеи в программе. По мере того как начнут прорисовываться детали проекта, будут меняться наши представления о форме реализации исходной идеи. Работа над проектом начинается с формулирования требо-

ваний к проекту, которые в ходе разработки могут меняться, что потребует внесения изменений в уже созданные программные блоки. Большой проект разбивается на отдельные блоки, для которых сначала создаются прототипы, а затем процедуры их выполнения. Тестирование выполнения отдельных модулей может привести к необходимости внесения изменений в их прототипы, а изменения отдельных блоков заставляют время от времени пересматривать принципы их взаимодействия в целом проекте.

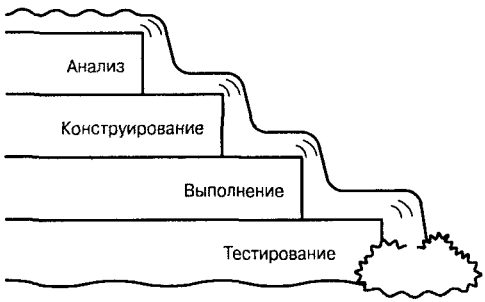


Рис. 18.3. Каскадный процесс проектирования

Хотя цикличность работы над проектом очевидна, описать эти процессы в виде какого-то стабильного цикла довольно сложно. Поэтому предлагаю вам лишь логическую последовательность действий: возникновение идеи, анализ и осмысление ее, проектирование, программирование, тестирование и возвращение к тому этапу, который можно модернизировать. Таким образом, итеративность разработки проекта не заставляет вас кружить по замкнутому циклу, а позволяет творчески подойти к решению задач и возвращаться всякий раз к тому этапу, где вы видите возможность повысить эффективность выполнения программы.

Еще раз повторим последовательность действий.

1. Разработка концепции.
2. Анализ.
3. Проектирование.
4. Реализация.
5. Тестирование.
6. Возвращение.

Разработка концепции — это вынашивание чистой идеи, к сожалению, далекой от реальной жизни. Анализ — это процесс осознания требований к проекту. Проектирование — процесс формирования модели классов, на основе которой будет создаваться код. Реализация — написание кода (например, на C++); тестирование — проверка того, все ли в порядке, и возвращение — это шлифовка вашего продукта до того состояния, когда его можно будет отдать заказчику. Осталось реализовать все это на практике.

Идея

Любая гениальная программа начинается с идеи. Некто думает о продукте, который, с его точки зрения, было бы хорошо создать. Реже сногшибательную идею выдают комитеты. На самой первой стадии анализа и проектирования объектно-ориентированного

программного продукта эта самая идея должна быть зафиксирована одним предложением (в крайнем случае, кратким абзацем). Идея становится ведущим принципом разработки, и команда, собравшаяся для ее реализации, по мере продвижения вперед должна на нее оглядываться, а в случае необходимости и корректировать.

Дискуссии

Много спорят о том, что происходит на каждом этапе процесса итеративного проектирования, и даже о том, как называется каждый этап. Откроем тайну: *это не имеет значения*. Основные этапы каждого процесса одни и те же: найдите, что надо построить, спроектируйте решение и реализуйте проект.

Хотя на дискуссиях расцвели пышным цветом группы новостей и списки электронных адресов специалистов по объектным технологиям, в сущности, объектно-ориентированный анализ и проектирование довольно просты. В этой главе описан практический подход к процессу создания архитектуры приложения.

Целью всей этой работы является создание кода, соответствующего установленным требованиям, а также отличающегося надежностью, расширяемостью и настраиваемостью. Не менее важным является создание высококачественного продукта в установленные сроки и в пределах бюджета.

Даже если новая идея исходит от группы товарищей из отдела маркетинга, кто-то все-таки должен стать "крестным отцом" этой идеи и блюсти ее чистоту. Из идеи проистекают требования к проекту. Детали исходной идеи могут преобразиться с учетом реалий сроков и требований рынка, но основная задача, которую планируется решить с помощью новой программы, должна оставаться неизменной, иначе зачем же браться за этот проект. Если в ходе проработки деталей вы забудете о том, ради чего был задуман проект, то такой проект обречен.

Анализ требований

Этап разработки концепции, когда формулируется идея, очень короткий. Это не более чем вспышка озарения с последующим изложением на бумаге идеи, рожденной в уме теоретика. Большинство программистов включаются в проект на более поздних этапах, когда основная идея уже сформулирована.

Иногда формулирование идеи путают с определением требований к проекту. Сильная идея необходима, но этого недостаточно. Чтобы перейти к анализу, требуется понять, каким образом, где и кем будет использоваться данный программный продукт. Цель этапа анализа состоит в том, чтобы сформулировать и зафиксировать эти требования. Результатом анализа должен быть документ с четкими требованиями к разработчикам проекта. Первым его разделом будет определение ситуаций использования проекта.

Ситуации использования

Определение ситуаций использования проекта лежит в основе анализа и проектирования программного продукта. *Ситуация использования* — это описание в общих чертах того, каким образом будет использоваться программный продукт. От этого зависит подбор методов и классов для реализации основной идеи.

Обсуждение всех возможных ситуаций использования может быть важнейшей задачей анализа. На этом этапе просто необходимо прибегнуть к помощи экспертов, которые помогут учесть многие моменты, далекие от обычного программирования, например особенности спроса и предложения на рынке программных продуктов и многое другое.

На этом этапе также следует уделить некоторое внимание проектированию интерфейса программного продукта, но внутренние методы реализации проекта нас еще не должны волновать. Пока наше внимание сконцентрировано на пользователе. *Пользователем* может быть не только отдельный человек, но и определенная группа людей, организация или другой программный продукт.

Таким образом, определение ситуаций использования включает:

- формулирование общих представлений о том, где и каким образом будет использоваться создаваемый программный продукт;
- работу с экспертами по выяснению особенностей предполагаемого места использования продукта, не связанных с проблемами обычного программирования;
- определение пользователя, для которого создается программный продукт.

Под ситуацией использования следует понимать больше, нежели просто тип компьютерной системы или конкретная организация-заказчик. Необходимо также учесть особенности взаимодействия будущих пользователей с разрабатываемым программным продуктом. На данном этапе программный продукт следует рассматривать как "черный ящик". Важно четко определить, какие вопросы будет ставить пользователь перед системой и какие ответы он ожидает получить.

Определение пользователей

Обратите внимание, что пользователи — это не обязательно люди. Системы, которые будут взаимодействовать с создаваемой нами системой, тоже пользователи. Таким образом, если создается программа для автоматизированного кассового аппарата (АТМ, известного как банкомат), то пользователем по отношению к нему будут клиенты и банковские клерки, а также другие банковские системы, например система по отслеживанию ипотек или по выдаче ссуд для студентов. Основные характеристики пользователей таковы:

- они являются внешними по отношению к системе;
- они взаимодействуют с системой.

При анализе ситуаций использования нередко самым трудным бывает начало. Лучше на этом этапе слишком много не думать, а сразу броситься в атаку: просто напишите список людей и систем, которые будут взаимодействовать с вашей системой. Помните, что важно не то, как зовут человека, а в какой роли он будет выступать по отношению к новой системе: клерком, менеджером, клиентом и т.д. Один человек может иметь несколько ролей.

В случае создания программного обеспечения для АТМ необходимо учесть следующих возможных пользователей:

- клиент;
- менеджер;
- компьютерная система банка;
- клерк, заправляющий кассовый аппарат деньгами и ответственный за его включение и выключение.

Поначалу нет необходимости чрезмерно расширять и детализировать исходный список пользователей. Для описания ситуаций использования достаточно определить трех или четырех пользователей. Каждый из них по-разному взаимодействует с системой. Каждое взаимодействие должно быть учтено при определении ситуаций использования.

Определение первой ситуации использования

Начнем с клиента. В общих чертах опишем, как клиент будет взаимодействовать с нашей системой.

- Клиент проверяет, что осталось на его счетах.
- Клиент кладет деньги на свой счет.
- Клиент снимает деньги со своего счета.
- Клиент переводит деньги со счета на счет.
- Клиент открывает счет.
- Клиент закрывает счет.

Надо ли различать ситуации, когда клиент кладет деньги на свой расчетный, а когда на депозитный счет, или можно скомбинировать эти действия в одну ситуацию: клиент кладет деньги на свой счет, как было сделано в списке? Ответ зависит от значимости такого различия для конкретного банка.

Чтобы определить, представляют ли эти действия одну ситуацию использования или две, надо выяснить, различны ли *механизмы обработки* (делает ли клиент нечто существенно различное с этими вкладами) и различны ли *выходы* (реагирует ли система по-разному). На оба вопроса в нашем случае ответ будет отрицательным: механизм внесения клиентом денег на разные счета в целом одинаков и система в обоих случаях прореагирует однотипно — увеличит сумму на соответствующем счете.

При условии, что пользователь и система ведут себя более-менее идентично в двух разных ситуациях, эти ситуации можно объединить в одну. Позднее можно конкретизировать сценарии использования системы и разделить эти ситуации, если возникнет необходимость.

Анализируя действия разных пользователей, можно обнаружить дополнительные ситуации использования, ответив на ряд вопросов.

- Почему пользователь использует систему?

  Чтобы получить наличные, сделать вклад или проверить остаток на счете.
- Какой результат ожидает пользователь от своего запроса к системе?

  Положить наличные на счет или снять их, чтобы сделать покупку.
- Что заставило пользователя прибегнуть к этой системе сейчас?

  Возможно, ему недавно выплатили зарплату или надо сделать покупку.
- Что следует выполнить пользователю, чтобы воспользоваться системой?

  Вставить карточку в гнездо кассового аппарата ATM.

  Ага! Нужно учесть ситуацию, когда клиент регистрируется в системе.
- Какую информацию клиент должен предоставить системе?

  Ввести личный идентификационный номер.

Ага! Нужно предоставить возможность клиенту получить или изменить личный идентификационный номер.

- Какую информацию пользователь хочет получить от системы?

  Остатки на счетах и т. д.

Часто можно обнаружить дополнительные ситуации использования, обратив внимание на структуру учета пользователей в доменах. У клиента есть имя, личный идентификационный номер и номер счета. Предусмотрена ли в системе возможность обработки и изменения этих данных? Счет имеет номер, остаток и записи трансакций. Как в системе будут возвращаться и обновляться эти данные?

После детального изучения всех ситуаций использования, связанных с клиентом, следующим шагом будет анализ ситуаций использования для всех оставшихся пользователей. В примере с ATM можно получить следующий список ситуаций использования для разрабатываемой нами системы:

- Клиент проверяет остатки на своих счетах.
- Клиент кладет деньги на свой счет.
- Клиент снимает деньги со своего счета.
- Клиент переводит деньги со счета на счет.
- Клиент открывает счет.
- Клиент закрывает счет.
- Клиент получает доступ к своему счету.
- Клиент проверяет недавние трансакции.
- Банковский служащий получает доступ к специальному управляющему счету.
- Банковский служащий регулирует выплаты по счетам клиентов.
- Банковская компьютерная система обновляет счет клиента на основе внешних поступлений.
- Изменения на счете клиента отображаются и возвращаются в банковскую компьютерную систему.
- ATM сигнализирует об отсутствии наличных денег для выдачи.
- Банковский клерк заправляет ATM наличными и включает его.

Создание модели домена

После того как сделан первый набросок ситуаций использования системы, можно приступать к описанию в документе требований модели домена. *Модель домена* — это документ, фиксирующий все, что известно о *домене* (области использования программного продукта). Модель домена состоит из *объектов домена*, каждый из которых соответствует определенному элементу, упоминавшемуся при описании ситуаций использования системы. В нашем примере с кассовым аппаратом необходимо учесть следующие объекты: клиент, персонал банка, банковская компьютерная система, расчетный счет, депозитный счет и т.д.

Для каждого из этих объектов домена требуется зафиксировать такие данные: имя (например, клиента, счета и т.д.), основные атрибуты объекта, является ли объект пользователем и прочее. Многие средства моделирования поддерживают фиксирование такого рода информации в описаниях классов. На рис. 18.4 показано, как эта информация фиксируется с помощью системы Rational Rose.

Важно понять, что мы имеем дело не с программными объектами, а с реальными фигурантами, которых следует учитывать при разработке проекта. Никто не заставляет нас для каждого объекта домена создавать объекты в программе.

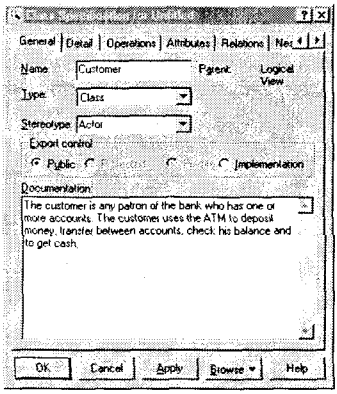


Рис. 18.4. Система Rational Rose

Используя соглашения UML, можно создать диаграмму для нашего кассового аппарата, в которой будут отражены отношения между объектами домена точно так же, как изображаются отношения между классами в программе. В этом одна из сильных сторон UML: на всех этапах проектирования можно использовать одни и те же средства.

Например, можно зафиксировать, что расчетный и депозитный счета являются уточнениями более общего понятия банковского счета. Как уже отмечалось, в UML обобщение производных классов в базовый отображается с помощью стрелок (рис. 18.5).

На диаграмме, показанной на этом рисунке, прямоугольники представляют различные объекты домена, а стрелки, направленные вверх, означают обобщение частных объектов в общий. Таким образом, в терминах языка C++ можно сказать, что объекты домена *Расчетный счет* и *Депозитный счет* являются производными от объекта *Банковский счет*.

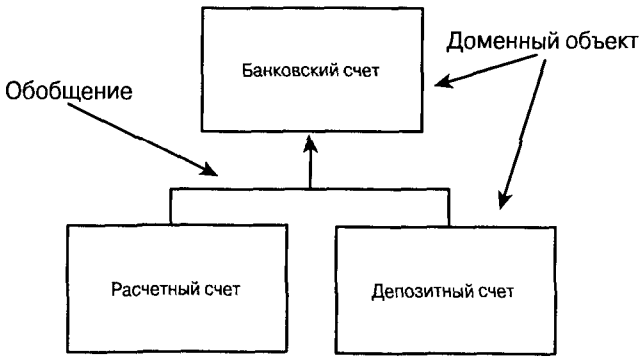


Рис. 18.5. Отношения между объектами домена, выраженные средствами UML

UML — богатый язык моделирования, с помощью которого можно фиксировать самые разные отношения. Однако для нас наиболее важными будут отношения обобщения, вложения и ассоциации.

ПРИМЕЧАНИЕ

Вновь обратите внимание, что в данном случае рассматриваются отношения между объектами домена. Позднее, при разработке проекта, возможно, вы захотите реализовать эти отношения между объектами классов `CheckingAccount` (Расчетный счет) и `BankAccount` (Банковский счет), используя наследование классов, но это будет лишь один из возможных вариантов разработки проекта. Пока что мы просто пытаемся разобраться, как взаимодействуют друг с другом реальные объекты домена.

Обобщение

Обобщение часто рассматривают как синоним наследования, но между ними есть существенное отличие. Обобщение описывает вид отношений, а наследование является реализацией обобщения средствами программирования.

Обобщение подразумевает, что производный объект *является* подтипом базового. Таким образом, расчетный счет *является* видом банковского счета. В свою очередь, банковский счет *обобщает* атрибуты и свойства расчетного и депозитного счетов.

Вложение

Часто один объект состоит из многих подобъектов. Например, автомобиль состоит из руля, шин, дверей, коробки передач и т.п. Расчетный счет состоит из сальдо, записи трансакций, кода клиента и т.д. Мы говорим, что расчетный счет содержит эти объекты в себе, другими словами, эти объекты вложены в расчетный счет. Вложенность, или содержание в себе средствами UML обозначается стрелкой с ромбом на конце, которая направлена от внешнего объекта к внутреннему (рис. 18.6).

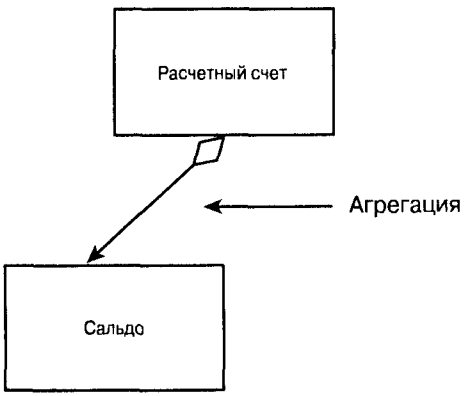


Рис. 18.6. Отношение вложения

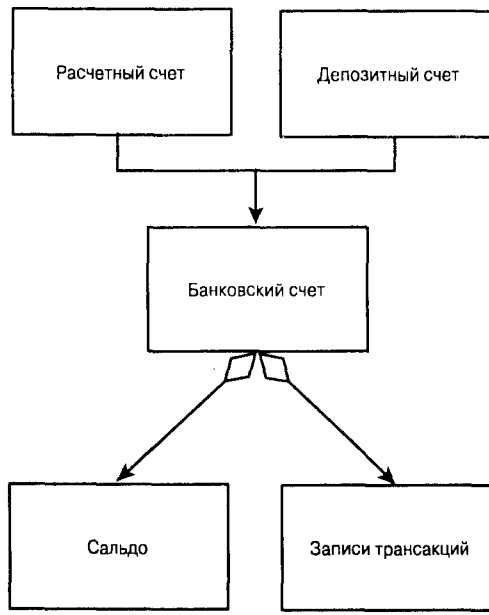


Рис. 18.7. Сложные отношения между объектами

Диаграмма на рис. 18.6 показывает, что объект *Расчетный счет* содержит в себе другой доменный объект — *Сальдо*. Чтобы показать достаточно сложный набор отношений, две предыдущие диаграммы можно скомбинировать (рис. 18.7).

Диаграмма на рис. 18.7 показывает, что объекты *Расчетный счет* и *Депозитный счет* обобщены в *Банковский счет*, а в объект *Банковский счет* вложены объекты *Сальдо* и *Записи трансакций*.

Ассоциация

Третье отношение — ассоциация — обычно фиксируется во время анализа домена. *Ассоциация* предполагает, что два объекта "знают" друг друга и некоторым образом взаимодействуют. Определение станет намного точнее на этапе проектирования, но для анализа лишь предполагается, что *Объект А* и *Объект Б* взаимодействуют, но ни один из них не содержит и не является частным видом другого. В UML эта ассоциация показана с помощью простой прямой линии между объектами (рис. 18.8).

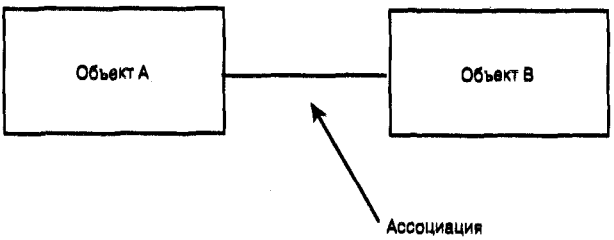


Рис. 18.8. Отношение ассоциации

Диаграмма на рис. 18.8 означает, что *Объект А* некоторым образом взаимодействует с *Объектом Б*.

Разработка сценариев

Теперь, когда мы разобрались со всеми ситуациями использования программы и средствами отображения отношений между объектами домена, можно углубиться в детализацию требований к программному продукту.

Каждый случай использования можно разбить на ряд сценариев. *Сценарий* — это описание определенного набора обстоятельств, конкретизирующих ситуации использования. Например, ситуация использования, при которой клиент снимает деньги со счета, может иметь несколько сценариев.

- Клиент делает запрос на снятие $300 с расчетного счета, кладет наличные в кошелек и ожидает квитанции.
- Клиент делает запрос на снятие $300 с расчетного счета, но остаток на счете составляет всего $200. Ему поступает информация, что для выполнения операции недостаточно денег на расчетном счете.
- Клиент делает запрос на снятие $300 с расчетного счета, но сегодня с этого счета уже сняли $100, а дневной лимит составляет $300. Поступает информация, что ему разрешается снять только $200.
- Клиент делает запрос на снятие $300 с расчетного счета, но в рулоне для печатания квитанций закончилась бумага. Ему поступает информация о возникшей технической неисправности и предложение подождать, пока персонал банка устранит эту проблему.

Список сценариев можно продолжить. Каждый сценарий определяет вариант первоначальной ситуации использования системы. Часто такие варианты являются исключительными ситуациями (недостаточно денег на счете, техническая неисправность и т.д.). Иногда сценарий содержит в себе вариант решения, предлагаемого пользователю. Например, предложение клиенту перевести деньги со счета до его закрытия.

Различных сценариев можно придумать бесчисленное множество, но отобрать среди них следует только те, на которые система готова ответить определенными действиями, сформулированными в требованиях к ней.

Разработка путеводителей

Определив список сценариев для каждой ситуации использования, необходимо разработать путеводители для всех сценариев, которые включаются в документ требований и содержат ряд определений.

- Предварительные условия, определяющие начало сценария.
- Переключатели, включающие выполнение именно этого сценария.
- Действия, выполняемые пользователем.
- Требуемые результаты выполнения программы.
- Информация, возвращаемая пользователю.
- Запускаемые циклы и условия выхода из них.
- Логическое описание сценария.
- Условие завершения сценария.
- Итоги выполнения сценария.

Кроме того, в документе требований нужно указать ситуацию использования и имя сценария, как в следующем примере.

Ситуация использования:	Клиент снимает наличные со счета
Сценарий:	Успешное снятие наличных с расчетного счета
Предварительные условия:	Клиент уже имеет доступ в систему
Переключатель:	Запрос от клиента на снятие денег со счета
Описание:	От клиента поступил запрос на снятие денег с расчетного счета. На счете имеется достаточная сумма. В кассовом аппарате достаточно денег и заправлена бумага для квитанций; сеть включена и работает. АТМ просит клиента указать сумму денег для снятия. Клиент указывает сумму, не превышающую $300. Машина выдает деньги и печатает квитанцию
Итоги:	Со счета клиента снята указанная сумма; сальдо счета уменьшено на эту сумму

Эту ситуацию использования можно изобразить с помощью простой диаграммы, представленной на рис. 18.9.

Диаграмма не может похвастаться обилием отображаемой информации. Отношения между пользователем и системой показаны довольно абстрактно. Эта диаграмма станет гораздо полезнее, когда будут показаны взаимоотношения между разными си-

туациями использования. Таких отношений может быть только два: *использование* и *расширение*. Отношение использования означает, что одна ситуация использует другую. Например, невозможно снять деньги со счета без регистрации в системе. Это отношение показано на рис. 18.10.

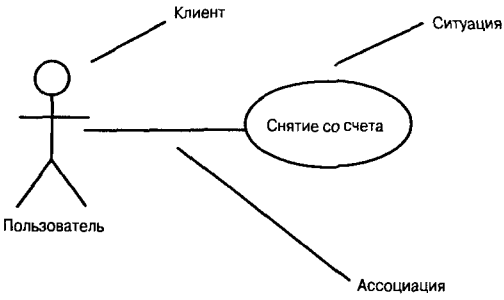


Рис. 18.9. Диаграмма ситуации использования

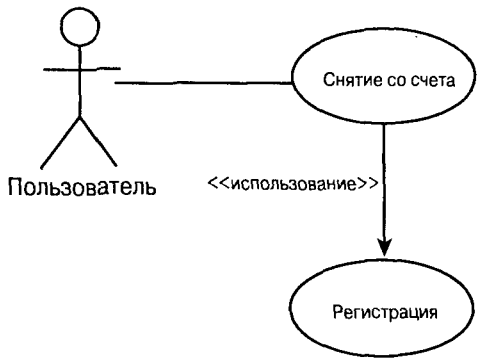


Рис. 18.10. Отношение подчинения между случаями использования

На рис. 18.10 показано, что для снятия денег со счета необходимо выполнить регистрацию в системе. Таким образом, ситуация *Снятие со счета* использует ситуацию *Регистрация в системе*, т.е. операция регистрации является частью операции снятия со счета.

Расширение ситуации использования подразумевает установление каких-то логических условных отношений между разными ситуациями, что также может реализовываться наследованием классов. Вообще, среди специалистов по объектному моделированию существует столько разногласий по поводу того, чем отличается использование от расширения, что многие из них просто не применяют второй термин, считая его слишком неопределенным. Лично я обращаюсь к термину *использование*, когда одна операция абсолютно необходима для выполнения другой. Если же выполнение операции ограничивается рядом условий, то я пользуюсь термином *расширение* ситуации использования.

Диаграммы взаимодействий

Хотя диаграмма ситуации использования вряд ли представляет собой большую ценность, такого рода диаграммы можно комбинировать, что значительно улучшает документацию и понимание взаимоотношений объектов системы. Например, известно, что сценарий ситуации *Снятие со счета* представляет взаимодействие между такими объектами домена, как клиент, расчетный счет и интерфейс пользователя системы. Это можно документировать диаграммой взаимодействий, показанной на рис. 18.11.

Эта диаграмма фиксирует детали сценария, которые могут не быть очевидными при чтении текста. Взаимодействующие объекты являются объектами *домена*. Весь пользовательский интерфейс кассового аппарата рассматривается как единый объект. В деталях рассматривается только определение системой расчетного счета в банке и снятие с него заказанной суммы денег.

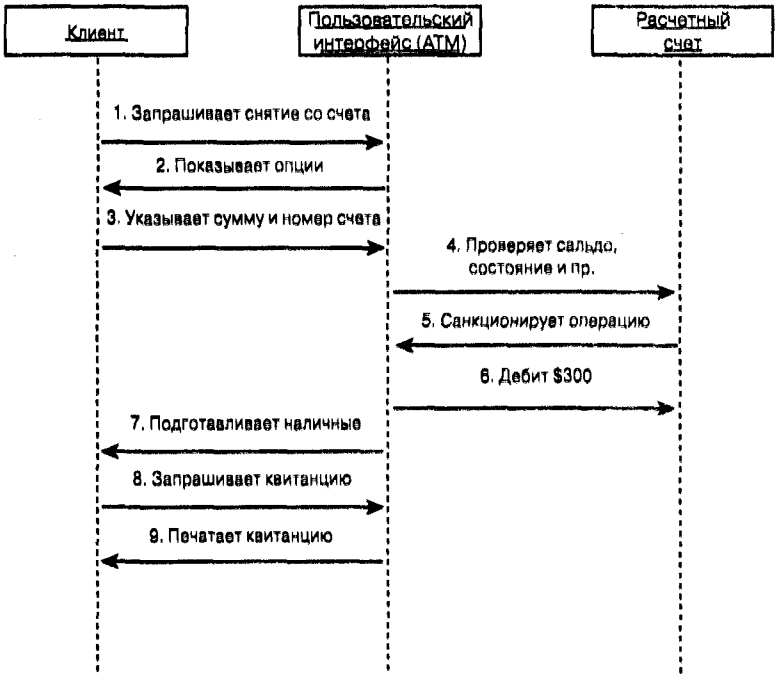


Рис. 18.11. Диаграмма взаимодействий системы кассового аппарата ATM с клиентом при выполнении операции снятия со счета

Отношения между объектами домена пока раскрыты лишь в общих чертах, но это уже большой шаг в выявлении ключевых моментов этих отношений, на базе которых будут формироваться требования к разрабатываемой системе.

Создание пакетов

Так как при анализе любой более-менее серьезной проблемы число возможных ситуаций использования разрастается как снежный ком, бывает сложно отобразить все эти ситуации в одной диаграмме. Язык моделирования UML предоставляет возможность группировать различные ситуации в пакеты.

Пакет напоминает папку в файловой системе компьютера. Он является набором объектов моделирования (классов, деятелей и т.п.). Чтобы упорядочить ситуации использования, их можно распределить по пакетам в соответствии с конкретными логическими концепциями. Так, можно объединить вместе ситуации использования определенных банковских счетов (расчетных или депозитных) либо разделить их по типам клиентов или любым другим важным характеристикам. Одна и та же ситуация использования может быть представлена в разных пакетах, в результате чего между пакетами образуются логические взаимосвязи.

Анализ совместимости приложения

В дополнение к определению ситуаций использования в документе требований следует четко описать предполагаемых клиентов системы, ограничения и требования к вычислительной аппаратуре и операционным системам. Документ требований к

приложению можно представить как первого абстрактного пользователя вашей системы, желания и предпочтения которого следует учесть при разработке приложения. От того, насколько точно документ требований будет отражать чаяния, умения и навыки реальных клиентов, зависит успех вашего проекта.

На требования к приложению часто накладывают отпечаток реалии существующих аппаратных и программных систем, под которые разрабатывается проект. Очень важно, чтобы новая система органично влилась в те системы и структуры, которые на данный момент уже существуют у заказчика.

В идеале программист разрабатывает проект решения поставленных задач, а затем определяет, какая платформа и операционная система максимально подходят для проекта. Этот сценарий сколь идеален, столь и уникален. Чаще заказчик уже давно потратил деньги на определенную операционную систему или аппаратное обеспечение, а теперь хочет с их помощью реализовать новый проект. Важно еще на ранней стадии проектирования зафиксировать реальное программное и аппаратное обеспечение заказчика, чтобы строить новый проект в соответствии с этими реалиями.

Анализ существующих систем

Некоторые программы пишутся, чтобы работать самостоятельно вне каких бы то ни было систем, напрямую взаимодействуя лишь с конечным пользователем. Однако часто приходится разрабатывать проекты, которые необходимо внедрить в уже существующую систему. В таком случае следует проанализировать все детали и механизмы работы систем, с которыми требуется наладить взаимодействие. Будет ли создаваемая система сервером, обслуживающим существующую систему, или ее клиентом? Сможете ли вы добиться однотипности интерфейсов двух систем и адаптировать свой проект к имеющимся стандартам? Будут ли взаимосвязи с существующей системой статическими или динамическими?

На эти и аналогичные вопросы следует отвечать на этапе анализа, прежде чем вы приступите к проектированию новой системы. Кроме того, необходимо зафиксировать те ограничения, которые могут возникнуть косвенно в результате взаимодействия двух систем. Не замедлит ли новая система работу существующей системы, не исчерпает ли она предоставляемые ресурсы и машинное время и т.д.

Прочая документация

Когда наконец-то придет понимание того, что система должна делать и как себя вести, необходимо уточнить бюджет и сроки проекта. Часто крайний срок диктуется заказчиком: "На эту работу у вас 18 месяцев". У программиста на этот счет может быть свое мнение, которое необходимо высказать. Идеально, если заказчик и исполнитель придут к компромиссу, но в любом случае время и бюджет всякого проекта всегда ограничены. Уложиться в сроки и не превысить бюджет часто бывает труднее, чем написать программу.

При определении бюджета и сроков следует учесть два момента.

- Если вы определили, во сколько в среднем обойдется проект, то попросите немного больше, тогда, может быть, вам дадут ту сумму, на которую вы рассчитывали.
- Закон Либерти утверждает, что на все требуется больше времени, чем ожидалось, даже если был учтен закон Либерти.

После того как время и бюджет будут установлены, определитесь с приоритетами. *Вы все равно не уложитесь в срок*, будьте к этому готовы. Важно, чтобы к тому моменту, когда нужно будет что-то показывать, у вас уже было что показать. Если вы строи-

ли мост, а время уже истекло, то позаботьтесь о том, чтобы была проложена хотя бы велосипедная дорожка. Это, конечно, не Бог весть что, но лучше чем ничего. По крайней мере, можно будет попросить денег на продолжение. Если же время истекло, а вы дошли только до середины реки, то это еще хуже.

Ко всем цифрам, зафиксированным в документации, следует относиться серьезно, но не "брать дурного в голову". В начале работ фактически невозможно точно оценить сроки выполнения проекта. Желательно приберечь для себя от 20 до 25% времени для маневра, если в ходе выполнения проекта возникнут неожиданности. В конце концов, для всех важен успех проекта и обоснованные колебания в сроках всегда допустимы.

Мы вовсе не призываем к бесшабашному отношению к срокам, зафиксированным в документе. Просто реалии таковы, что на ранних этапах планирования невозможно точно определить, сколько времени и денег потребуется на разработку этого проекта. Приоритетом при этом должна быть максимальная реализация требований заказчика, что, вполне вероятно, может вызвать необходимость корректировки исходных цифр при выполнении проекта.

Визуализация

Визуализация является финальной частью документа требований. Такое модное название носят диаграммы, рисунки, изображения экранов, прототипы и другие визуальные средства, созданные для того, чтобы помочь в проектировании графического интерфейса пользователя создаваемого продукта.

При создании больших проектов бывает полезно разработать полный прототип проекта, чтобы лучше представить, как поведет себя система в тех или иных ситуациях. Часто графическое приложение к документации лучше всего справляется с функцией определения требований к проекту, так как позволяет наглядно увидеть, какой должна быть система и как она должна работать в конечном варианте.

Артефакты

К концу каждого этапа анализа и проектирования накапливается ряд документов, называемых артефактами. Приблизительный набор таких документов показан в табл. 18.1. Эти документы используются для организации взаимодействия и протоколирования отношений между заказчиком и исполнителем. Их подписание обеими сторонами гарантирует, что исполнитель четко понял и принял все требования заказчика и обязуется разработать в указанные сроки проект, который в окончательном варианте должен выполнять точно определенный ряд функций.

Таблица 18.1. Артефакты, составляющие документацию проекта

Артефакт	*Описание*
Описание ситуаций использования	Подробный документ обо всех ситуациях использования системы, сценариях, стереотипах, предварительных условиях и итогах выполнения с визуализацией интерфейса пользователя
Модель домена	Диаграммы отношений объектов домена, для которого разрабатывается проект

Артефакт	Описание
Диаграммы организации сотрудничества	Диаграммы отношений внешних объектов и систем, задействованных в решении той же заданной проблемы
Анализ существующих систем	Описание и диаграммы низкоуровневой аппаратной системы, для которых разрабатывается проект
Требования к приложению	Описание требований заказчика к данному проекту
Отчет об ограничениях	Описание ограничений системной платформы и особенности предполагаемых клиентов, что должно быть учтено при разработке проекта
Смета и календарный план	Календарный план с указанием сроков и этапов разработки проекта и материалов, предоставляемых заказчику после выполнения каждого этапа

Проектирование

Во время анализа основное внимание уделяется рассмотрению домена и определению задач и требований к проекту, в то время как проектирование сосредоточено на поиске решений. *Проектирование* — это планирование практической реализации нашей идеальной модели средствами программирования. Результатом этого процесса является создание документа проекта приложения.

Документ проекта состоит из двух разделов: проекта классов и архитектуры приложения. Первый раздел, в свою очередь, содержит статический (описание различных классов, их структуры и характеристик) и динамический (описание взаимодействий классов) подразделы.

В разделе "Архитектура приложения" определяется время жизни различных объектов, их взаимоотношения, системы передачи объектов и другие механизмы реализации классов. Далее на этом занятии основное внимание уделяется проектированию классов, а все оставшиеся занятия посвящены рассмотрению различных структур архитектуры приложения.

Что такое классы

Изучая материалы этой книги, вы уже не раз делали попытку создавать классы в программах на языке C++. Но поскольку в данный момент речь идет не о разработке программы, а о ее проектировании, следует различать проекты классов и их реализацию средствами C++, хотя, безусловно, эти моменты тесно взаимосвязаны. Каждому классу в проекте будет соответствовать класс в программном коде, но все же не надо их путать. Ведь для реализации спроектированного класса можно использовать другой язык программирования, и даже в пределах C++ для реализации класса можно использовать различные средства программирования.

Далее не будем заострять на этом внимание, просто помните, что если планируется создать класс *Кот* с методом *Мяу()*, то в программу будут добавлены класс `Cat` с методом

`Meow()`, хотя реализовать их можно по-разному. Обратите внимание, что в тексте книги для классов проекта и классов программы использованы разные стили, чтобы помочь вам отличать их. Классы модели приложения отображаются в диаграммах UML, а классы C++ — в коде программы, который можно скомпилировать и запустить.

У начинающих программистов часто возникает проблема с определением необходимого количества классов для программы и функций, которые должен выполнять каждый отдельный класс. Один из наиболее упрощенных подходов к решению этой проблемы состоит в записи сценария для ситуации использования приложения. Затем можно попытаться создать классы для каждого существительного (объекта), упоминающегося в этом сценарии. Возьмем для примера приведенный ниже сценарий.

> **Клиент** выбирает **операцию снятия наличных** с **расчетного счета**. На **счете в банке** имеется достаточная сумма, в **ATM** достаточно **наличных** и заправлена **лента для квитанций**, а **сеть** включена и работает. Кассовый аппарат ATM просит указать **сумму**, которая не должна превышать $300. **Машина** выдает указанную сумму и печатает **квитанцию** для клиента.

Из этого сценария можно извлечь такие классы:

- клиент;
- сумма;
- наличные;
- расчетный счет;
- счет;
- квитанция;
- лента для квитанций;
- банк;
- ATM;
- сеть;
- снятие со счета;
- машина.

Объединив синонимы и явно взаимосвязанные объекты, получаем следующий список:

- клиент;
- наличные (суммы на счете и снимаемая со счета);
- расчетный счет;
- счет;
- квитанции;
- ATM (кассовый аппарат);
- сеть.

Пока что неплохо для начала. Можно затем отобразить отношения между классами, как показано на рис. 18.12.

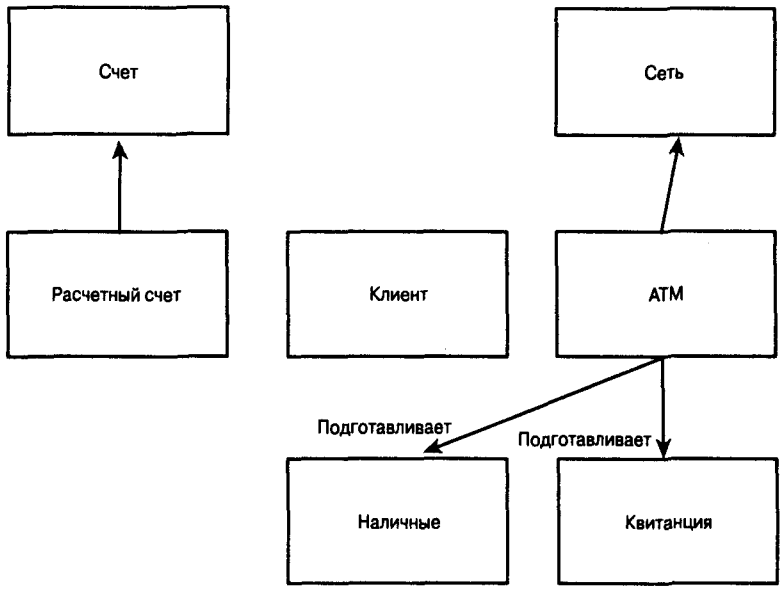


Рис. 18.12. Предварительная схема отношений между классами

Преобразования

Описанный в предыдущем разделе подход называется *преобразованием* объектов домена в объекты проекта. Большинству объектов домена в проекте соответствуют *суррогаты*. Термин "суррогат" вводится для того, чтобы отличать реальную квитанцию, выданную кассовым аппаратом, от виртуального объекта в программе, являющегося абстракцией, реализованной в программном коде.

Многие объекты домена имеют в проекте изоморфное представление, т.е. между объектами домена и проекта существует отношение один-к-одному. В других случаях, однако, один объект домена представлен в проекте целым рядом объектов. Иногда множества объектов домена могут быть представлены одним объектом в проекте.

Обратите внимание: на рис. 18.12 уже зафиксирован факт, что *Расчетный счет* является специализацией *Счета*. Аналогично, из анализа объектов домена известно, что кассовый аппарат *ATM* подготавливает и выдает *Наличные* и *Квитанцию*, поэтому данные отношения зависимости классов также отображены на рис. 18.12.

Отношение между *Клиентом* и *Расчетным счетом* менее очевидно. Известно, что такое отношение существует, но детали его пока скрыты, поэтому оставим анализ этого отношения на потом.

Другие преобразования

После преобразования объектов домена можно начинать поиск других полезных объектов этапа проектирования. Неплохо начать с создания интерфейсов. Каждый интерфейс между новой системой и любой из существующих (унаследованных) систем должен быть инкапсулирован в класс интерфейса. (Напомним, что мы занимаемся проектированием, а не написанием программы, поэтому не путайте класс интер-

фейса в проекте приложения с интерфейсом класса в коде приложения. Подмена этих терминов вызовет бессмыслицу.) Если будет осуществляться взаимодействие с базой данных определенного типа, это тоже следует зафиксировать в классе интерфейса.

Классы интерфейса инкапсулируют протоколы интерфейса и таким образом защищают код программы от изменений в другой системе. Они позволяют менять ваш собственный проект или подстраиваться к изменениям структуры других систем, не нарушая остального кода. Пока две системы продолжают поддерживать согласованный интерфейс, они могут развиваться независимо друг от друга.

Обработка данных

Аналогично создаются классы обработки данных. Если надо сделать преобразование из одного формата в другой (например, из градусов Фаренгейта в градусы Цельсия или из английской системы в метрическую), то эти операции можно инкапсулировать внутри класса обработки данных. Этот прием можно использовать при отправке данных в другие системы в определенном формате или для передачи данных в Internet. В общем, каждый раз, когда надо преобразовать данные в определенный формат, протокол следует инкапсулировать в классе обработки данных.

Отчеты

Каждый отчет, выводимый системой (или связанная группа отчетов) является кандидатом в классы. Протокол формирования отчета, куда входит сбор информации и способ ее отображения, следует инкапсулировать в класс обзора.

Устройства

Если система взаимодействует с устройствами или управляет ими (такими как принтеры, модемы, сканеры и т.п.), то особенности протокола устройства следует инкапсулировать в классе устройства. Благодаря этому, внося изменения в класс устройства, можно подключать к системе новые устройства, не нарушая остального кода.

Статическая модель

Когда создан первоначальный набор классов, пора начинать моделировать их отношения и взаимодействия. Для большей ясности сначала объясним статическую модель, а затем — динамическую. При реальном процессе проектирования можно свободно переходить от одной модели к другой, заполняя обе подробностями, фактически добавляя новые классы и описывая их по мере продвижения.

Статическая модель сосредоточена в трех областях: распределении ответственности, атрибутах и взаимодействии. Наиболее важная из них (на что в первую очередь обратим внимание) — это распределение ответственности между классами. Перед каждым классом должна быть поставлена одна конкретная задача, за выполнение которой он несет ответственность.

Это не означает, что у каждого класса есть только один метод. Класс может содержать десятки методов. Однако все они должны быть согласованными и взаимосвязанными, т.е. должны обеспечивать выполнение единой задачи.

В хорошо спроектированной системе каждый объект является экземпляром класса, имеющего четко определенный набор функций и отвечающего за выполнение конкретной задачи. Классы обычно делегируют несвойственные им задачи другим, связанным с

ними классам. Создание классов, имеющих одну область ответственности, — основа написания читабельного и легко поддерживаемого кода.

Чтобы разобраться с ответственностью классов, следует начать проектирование с создания карточек CRC.

Карточки CRC

CRC означает Class (класс), Responsibility (ответственность), Collaboration (сотрудничество). CRC представляет собой обычную бумажную карточку размером, не превышающим используемые в картотеках. Работая с такими карточками, вы, как Чапаев с помощью картошки, сможете наглядно объяснить коллегам, которым будет поручена разработка отдельных классов, как вы мыслите наладить распределение ответственности за выполнение тактических и стратегических задач между классами проекта.

Как проводить заседания с карточками

На каждое заседание с карточками следует приглашать от трех до шести человек. Если людей больше, то теряется управляемость. Кроме того, во время дискуссии гораздо проще прийти к консенсусу, если в заседании участвует не слишком много людей. Кратко остановимся на том, кто в идеале должен участвовать в разработке серьезного проекта (если вы не хотите свалить на себя всю ответственность за провал). Итак, вы главный исполнитель. Пригласите как минимум одного ведущего специалиста по программной архитектуре, имеющего опыт в анализе и проектировании объектно-ориентированных программ. Не мешает также включить в состав минимум одного или двух "экспертов по домену", не понаслышке знающих проблему, которую вы хотите решить с помощью разрабатываемой программы.

В будущем вам потребуются менеджеры (если не адвокат), но не сейчас. Это творческое непринужденное заседание не для прессы и не для рекламы. Цель состоит в том, чтобы провести исследование, высказать рискованные предложения и в ходе дискуссии решить, какой класс нагрузить той или иной проблемой.

Заседание по CRC начинается с того, что группа рассаживается за столом, на котором лежит небольшая стопка карточек. В верхней части каждой из них пишется название одного из классов. Начертите сверху вниз линию, разделив карточку на две части, и слева напишите *Ответственность*, а справа — *Сотрудничество*.

Начинайте заполнять карточки по самым важным из определенных вами классов. С обратной стороны дайте небольшое описание в одно или два предложения. Можно также указать, уточнением (производным) какого класса является данный класс, если это очевидно к моменту работы с карточкой. Просто под именем класса напишите *Надкласс:* и впишите имя класса, от которого данный класс производится.

Сфокусируемся на распределении ответственности

Основным пунктом повестки дня заседания является определение *ответственности* каждого класса. Не обращайте много внимания на атрибуты, фиксируйте по мере продвижения только самые существенные и очевидные из них. Если для выполнения задачи класс должен делегировать часть работы другому классу, то эта информация указывается в столбце *Сотрудничество*.

В ходе работы обращайте внимание, сколько пунктов появилось на карточке класса в столбце *Ответственность*. Если на карточке не хватает места, то это повод задуматься, не следует ли разделить данный класс на два. Помните, каждый класс должен отвечать за выполнение одной задачи, поэтому все пункты в столбце *Ответственность* должны быть логически и функционально взаимосвязанными.

На данном этапе проектирования не стоит задумываться над тем, каким образом будет объявлен класс в программе и сколько открытых и закрытых методов он будет содержать. Обращайте внимание только на то, за что этот класс *отвечает*.

Как сделать класс живым

Главным свойством карточек CRC является то, что их можно сделать антропоморфными, т.е. каждый класс наделяется свойствами человека. Посмотрим, как это работает. После определения первоначального набора классов разложите по кругу на столе карточки CRC в произвольном порядке и вместе пройдитесь по сценарию. Например, вернемся к предложенному ранее сценарию.

> Клиент выбирает операцию снятия наличных с расчетного счета. На счете в банке имеется достаточная сумма, в АТМ достаточно наличных и заправлена лента для квитанций, а сеть включена и работает. Кассовый аппарат АТМ просит указать сумму, которая не должна превышать $300. Машина выдает указанную сумму и печатает квитанцию для клиента.

Предположим, в заседании участвуют пять человек: Эмма — ваш помощник, сведущая в объектно-ориентированном программировании; Борис — ведущий программист; Сергей — будущий клиент вашей системы; Олег — эксперт по домену; а также Эдик — программист.

Эмма держит карточку CRC класса *Расчетный счет* и говорит: "Я сообщаю клиенту, сколько можно получить денег. Он просит меня дать $300. Я посылаю сообщение на устройство выдачи, чтобы было выдано $300 наличными". Борис держит свою карточку и говорит: "Я устройство выдачи; я выдаю $300 и посылаю Эмме сообщение, чтобы она уменьшила остаток на счете на $300. Кому я должен сообщить, что в машине стало на $300 меньше? Должен ли я это отслеживать?" Сергей: "Думаю, нужен объект для слежения за наличностью в машине". Эдик: "Нет. Кассовый аппарат сам должен знать, сколько у него осталось денег; это не должно нас волновать". Эмма возражает: "Нет. Выдачу денег кто-то должен контролировать. Программа должна знать, доступна ли наличность и достаточно ли у клиента денег на счете. Кроме того, программа должна проследить, было ли выдано аппаратом именно столько денег, сколько было заказано. Учет денег в кассовом аппарате следует делегировать некоему внутреннему счету. Необходимо также, чтобы система оповещала технический персонал банка о том, что в кассовом аппарате закончились деньги".

Спор продолжается. Каждый класс реально стал человеком, который держит соответствующую карточку в руках и заполняет столбцы ответственности и сотрудничества.

Ограничения карточек CRC

Хотя карточки CRC могут быть мощным средством в начале проектирования, их возможности ограничены. Первая проблема в том, что большой проект может состоять из большого числа классов. Тогда в карточки можно будет зарыться с головой. Взаимоотношения между классами прорабатываются недостаточно. Правда, фиксируется сотрудничество, но его природа и механизмы не моделируются. По карточкам не видно, какие классы вкладываются в другие классы, какие наследуются, а какие ассоциируются. В карточках не отмечаются атрибуты класса, определяющие его внутреннюю структуру, поэтому трудно перейти от карточек прямо к коду. Нужно помнить, что карточки статичны по своей природе и, хотя в них и записываются в общем виде отношения между классами, они не годятся для построения динамической модели.

Короче говоря, карточки CRC являются хорошим началом, но для построения более полной модели проекта нужно перейти к UML. После создания модели UML карточки можно будет отложить в сторону. Они вам больше не потребуются.

Создание модели UML по карточкам CRC

Каждой карточке будет соответствовать класс диаграммы UML. Пункты из столбца *Ответственность* становятся методами класса. Также в диаграмму переносятся все зафиксированные атрибуты класса. Определение класса с обратной стороны карточки помещается в документацию класса. На рис. 18.13 показана диаграмма отношения между классами *Счет* и *Расчетный счет*, атрибуты класса *Расчетный счет* взяты с соответствующей карточки CRC, показанной ниже.

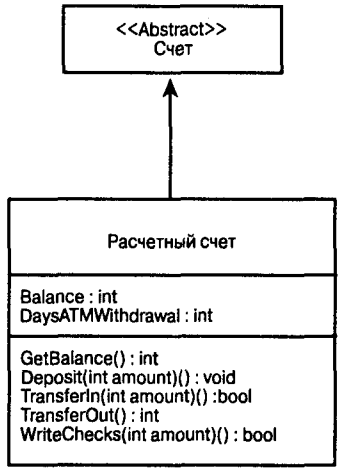


Рис. 18.13. Отображение данных карточки CRC на диаграмме

Класс: *Расчетный счет*

Надкласс: *Счет*

Ответственность:

Отслеживать текущий остаток

Принимать и переводить депозиты

Выдавать чеки

Переводить деньги при снятии со счета

Сохранять баланс выдачи кассового аппарата за текущий день

Сотрудничество:

Другие счета

Компьютерная система банка

Устройство выдачи наличных

Отношения между классами

После того как классы будут отображены средствами UML, можно заняться отношениями между ними. Рассматриваются четыре основных вида отношений.

- Обобщение.
- Ассоциация.
- Агрегирование.

- Композиция.

Обобщение реализуется в C++ с помощью открытого наследования. Но с точки зрения проектирования больше внимания следует уделять не механизму, а семантике: что именно подразумевает это отношение. Мы уже говорили об обобщении на этапе анализа, но теперь рассмотрим этот вид отношений применительно к классам проекта. Нашей задачей будет вынести общие действия за границы взаимосвязанных классов в общий базовый класс, который инкапсулирует общую ответственность.

Таким образом, если обнаружено, что расчетный и депозитный счета используют одни и те же методы для перевода денег, то в базовый класс *Счет* можно перенести метод `TransferFunds()`. Чем больше методов будет сосредоточено в базовых классах, тем более полиморфным становится проект.

Одним из средств программирования, доступных в C++, но не в Java, является *множественное наследование* (однако Java имеет похожее, хотя и ограниченное средство, позволяющее создавать множественные интерфейсы). Это средство позволяет производить класс более чем от одного базового класса, добавляя переменные-члены и методы двух и более классов.

Опыт показывает, что множественное наследование надо использовать умеренно, так как оно может усложнить программный код проекта. Многие проблемы, вначале решаемые с помощью множественного наследования, теперь решаются путем агрегирования (вложения) классов. Тем не менее множественное наследование остается мощным средством программирования, от которого не следует огульно отказываться при разработке проектов.

Множественное наследование против вложения

Является ли объект суммой его частей? Имеет ли смысл классы деталей автомобиля, такие *Руль*, *Двери* и *Колеса*, производить от общего класса *Автомобиль*, как показано на рис. 18.14?

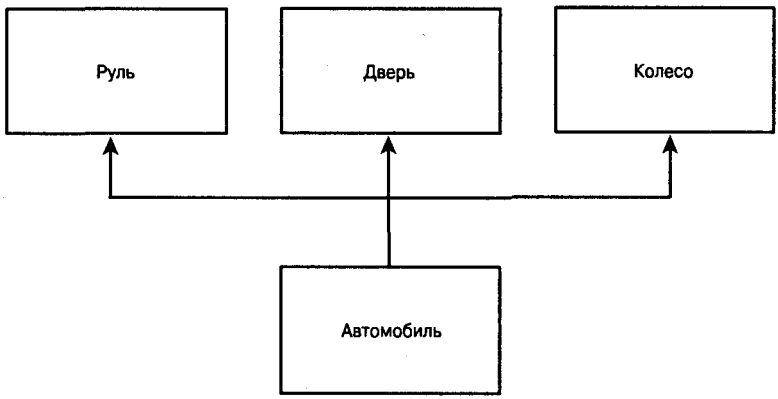


Рис. 18.14. Возможно, но не умно

Важно вернуться к основам: открытое наследование должно всегда моделировать обобщение, т.е. производный класс должен быть уточнением базового класса, чего не скажешь о приведенном выше примере. Если требуется смоделировать отношение "иметь" (например, автомобиль *имеет* руль), то это делается с помощью агрегирования (рис. 18.15).

Диаграмма на рис. 18.15 показывает, что автомобиль имеет руль, четыре колеса и от двух до пяти дверей. Это более точная модель отношения автомобиля и его частей. Обратите внимание, что ромбик на диаграмме не закрашен. Это потому, что отношение моделируется с помощью агрегирования, а не композиции. Композиция подразумевает контроль за временем жизни объекта вложенного класса. Хотя автомобиль *имеет* шины и дверь, но они могут существовать и до того, как станут частями автомобиля, и после того, как перестанут ими быть.

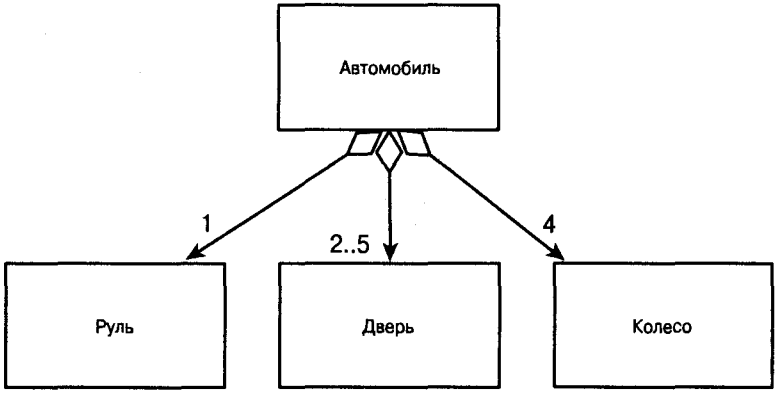


Рис. 18.15. Модель агрегирования

На рис. 18.16 показана модель композиции. Эта модель сообщает нам, что класс "тело" не только включает в себя (что можно было бы реализовать агрегированием) голову, две руки и две ноги, но что эти объекты (голова, руки и ноги) будут созданы при создании тела и исчезнут вместе с ним. Иными словами, они не имеют независимого существования.

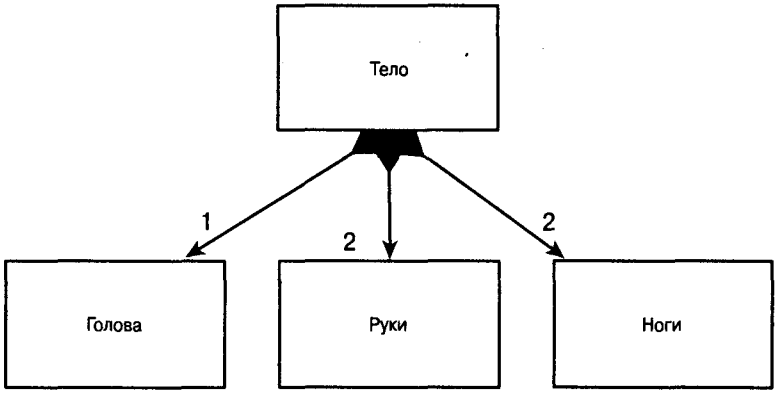


Рис. 18.16. Модель композиции

Дискриминаторы и силовые классы

Как можно спроектировать производство разных моделей автомобилей одной марки? Предположим, вас наняла фирма Acme Motors, которая производит пять автомобилей: Pluto (компактный малолитражный автомобиль для поездок за покупками),

Venus (четырехдверный "седан" с двигателем средней мощности), Mars (спортивный автомобиль типа "купе" с наиболее мощным двигателем, рассчитанный на максимальную скорость), Jupiter (мини-фургон с форсированным двигателем как у спортивного купе, правда, менее скоростной, зато более мощный) и Earth (маломощный, но скоростной фургон).

Можно было бы просто произвести все эти модели от общего класса `Car`, как показано на рис. 18.17.

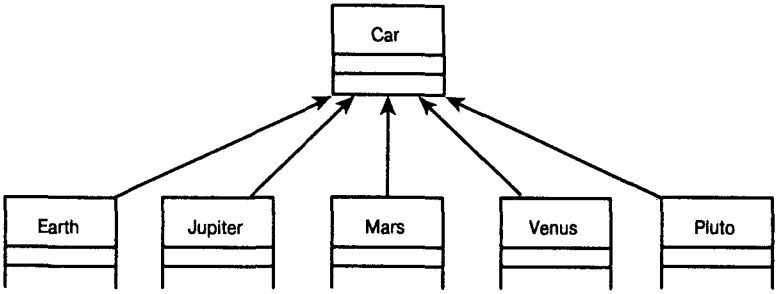


Рис. 18.17. Обобщение подклассов всех моделей в общий базовый класс

Но давайте более детально проанализируем различия между моделями. Очевидно, что они различаются мощностью двигателя, типами кузова и специализацией. Комбинируя эти основные признаки, мы получим характеристики различных моделей. Таким образом, в нашем примере важнее сконцентрировать внимание не на названиях моделей, а на их основных признаках. Такие признаки называются *дискриминаторами* и в UML отображаются особым образом (см. рис. 18.17).

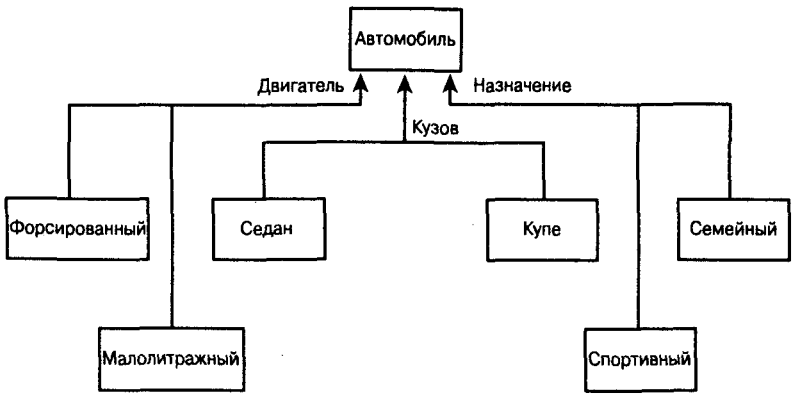


Рис. 18.18. Модель отношения дискриминаторов

Диаграмма на рис. 18.18 показывает, что классы разных моделей можно производить от класса *Автомобиль*, комбинируя такие дискриминаторы, как мощность двигателя, тип кузова и назначение автомобиля.

Каждый дискриминатор можно реализовать в виде простого перечисления. Например, объявим перечисление типов кузова:

```
enum BodyType={sedan, coupe, minivan, stationwagon}
```

Однако далеко не каждый дискриминатора можно объявить, просто назвав его. Например, назначение определяется многими параметрами. В таком случае дискриминатор можно смоделировать как класс, и разные типы дискриминатора будут возвращаться как объекты класса.

Таким образом, технические характеристики автомобиля, определяющие его использование, могут быть представлены объектом типа performance, содержащим данные о скорости, габаритах и прочих характеристиках. В UML классы, в которых инкапсулирован дискриминатор и которые используются для создания экземпляров другого класса (в нашем примере класса *Автомобиль*) таким образом, что разные экземпляры класса приобретают характеристики разных типов (например, *Спортивный автомобиль* и *Семейный автомобиль*), называются *силовыми*. В нашем примере класс *Назначение* (performance) является силовым для класса *Автомобиль*. При создании объекта класса *Автомобиль* также создается объект *Назначение*, который ассоциируется с текущим объектом *Автомобиль*, как показано на рис. 18.19.

Использование силовых классов позволяет создавать различные *логические* типы, не прибегая к наследованию. Поэтому в программе можно легко манипулировать множеством типов, не создавая класс для каждого нового типа.

Обычно в программах на C++ использование силовых классов реализуется с помощью указателей. Так, в нашем примере класс Car (соответствующий классу проекта *Автомобиль*) будет содержать указатель на объект класса PerformanceCharacteristics (рис. 18.20). Если хотите потренироваться, создайте самостоятельно силовые классы для дискриминаторов *Кузов* (body) и *Двигатель* (engine).

```
Class Car : public Vehicle
{
public:
     Car();
     -Car();
     //другие открытые методы опущены
private:
     PerformanceCharacteristics*pPerformance;
};
```

И наконец, силовые классы дают возможность создавать новые типы данных во время выполнения программы.

Поскольку каждый логический тип различается только атрибутами ассоциированных с ним силовых классов, то эти атрибуты могут быть параметрами конструкторов данных силовых классов. Это означает, что можно во время выполнения программы создавать новые *типы* автомобилей, изменяя установки атрибутов силовых классов. Число новых типов, которые можно создать во время выполнения программы, ограничивается только числом логических комбинаций атрибутов разных силовых классов.

Динамическая модель

В модели проекта важно указать не только отношения между классами, но и принципы их взаимодействия. Например, классы *Расчетный счет*, *ATM* и *Квитанция* взаимодействуют с классом *Клиент* в ситуации *Снятие со счета*. Возвращаясь к виду последовательных диаграмм, которые использовались в начале анализа (см. рис. 18.11), рассмотрим теперь взаимодействие классов на основе определенных для них методов, как показано на рис. 18.21.

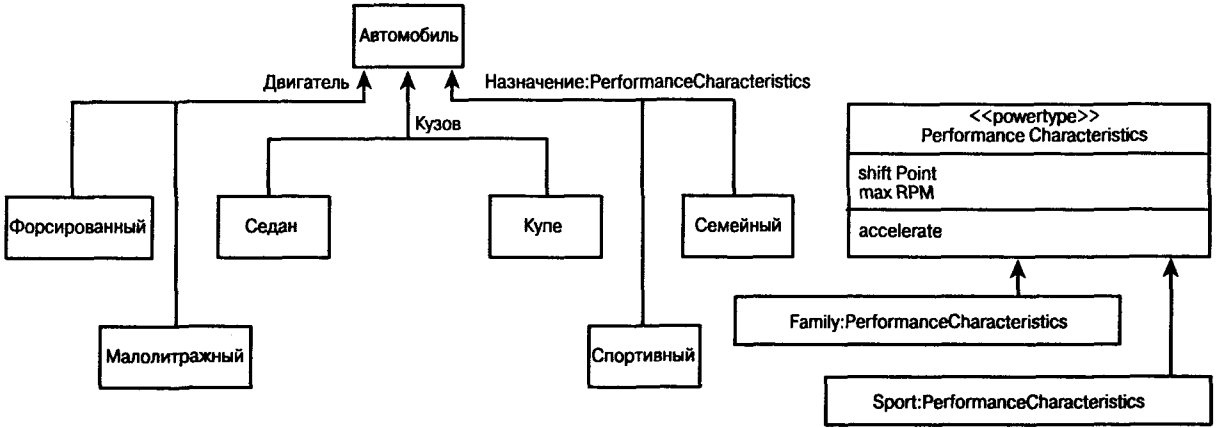


Рис. 18.19. Дискриминатор как силовой класс

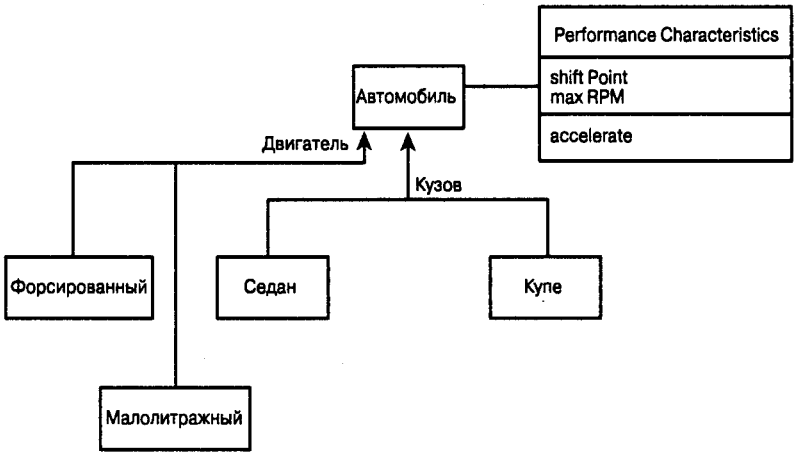


Рис. 18.20. Отношение между объектом класса Автомобиль *и связанным с ним силовым классом*

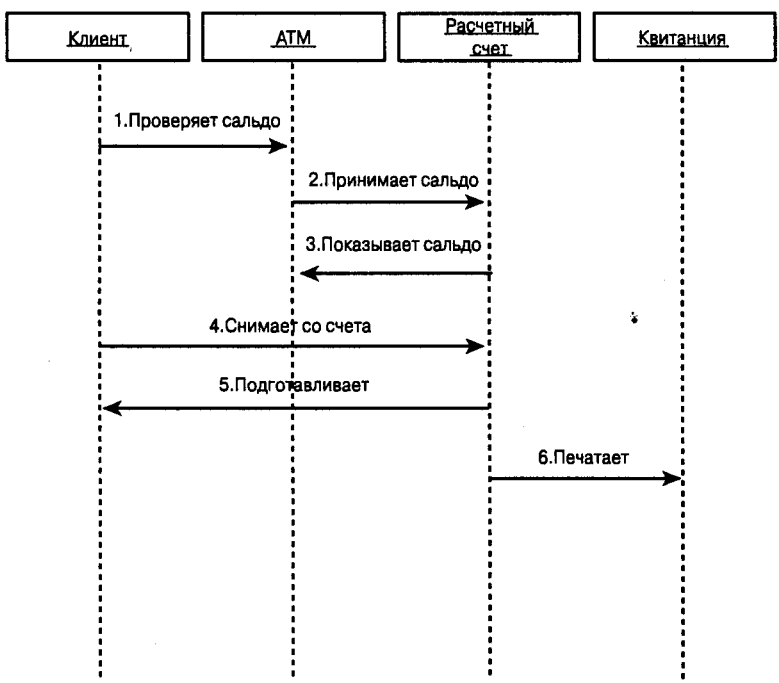


Рис. 18.21. Диаграмма взаимодействия классов

Эта простая диаграмма показывает взаимодействие между несколькими классами проекта при определенной ситуации использования программы. Предполагается, что класс *ATM* делегирует классу *Расчетный счет* ответственность за учет остатка денег на счете, в то время как *Расчетный счет* делегирует классу *ATM* ответственность за доведение этой информации пользователю.

Существует два вида диаграмм взаимодействий классов. На рис. 18.21 показана *диаграмма последовательности действий*. Та же ситуация, но в другом виде, изображена на рис. 18.22 и называется *диаграммой сотрудничества*. Диаграмма первого типа определяет последовательность событий за некоторое время, а диаграмма второго типа — принципы взаимодействия классов. Диаграмму сотрудничества можно создать прямо из диаграммы последовательности. Такие средства, как Rational Rose, автоматически выполнят это задание после щелчка на кнопке.

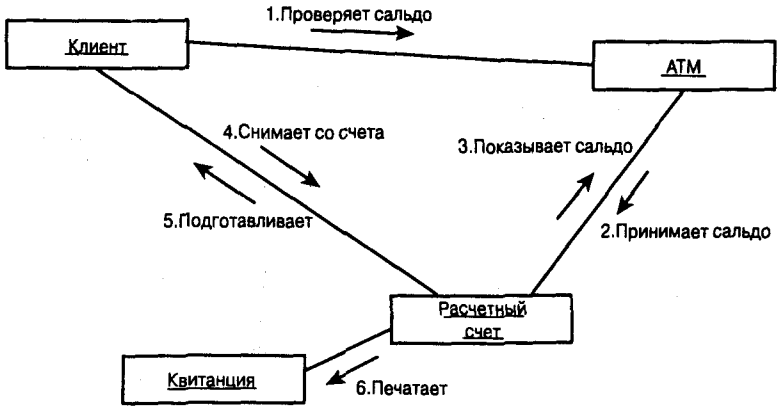


Рис. 18.22. Диаграмма сотрудничества

Диаграммы переходов состояний

После того как стали понятными взаимодействия между объектами, надо определить различные возможные *состояния* каждого из них. Моделировать переходы между различными состояниями можно в диаграмме состояний (или диаграмме переходов состояний). На рис. 18.23 показаны различные состояния класса *Расчетный счет* при регистрации клиента в системе.

Каждая диаграмма состояний начинается с состояния *Начало* и заканчивается нулем или некоторым другим концевым состоянием. Каждое состояние имеет свое имя, и в переходах между состояниями могут быть установлены *Сторожа*, представляющие собой условия, при выполнении которых возможен переход от состояния к состоянию.

Сверхсостояния

Клиент может в любое время передумать и не регистрироваться. Он может это сделать после того, как вставил карточку или после ввода пароля. В любом случае система должна принять его запрос на аннулирование и вернуться в состояние *Не зарегистрирован* (рис. 18.24).

Как видите, в более сложной диаграмме, содержащей много состояний, указание на каждом шаге возможности перехода к состоянию *Отмена* внесет сумятицу. Особенно раздражает тот факт, что отмена является исключительным состоянием, отвлекающим от анализа нормальных переходов между состояниями. Эту диаграмму можно упростить, используя *сверхсостояние* (рис. 18.25).

Диаграмма на рис. 18.25 дает ту же информацию, что и на рис. 18.24, но намного яснее и легче для чтения. В любой момент от начала регистрации и вплоть до ее завершения процесс можно отменить. Если вы это сделаете, то вернетесь в состояние *Не зарегистрирован*.

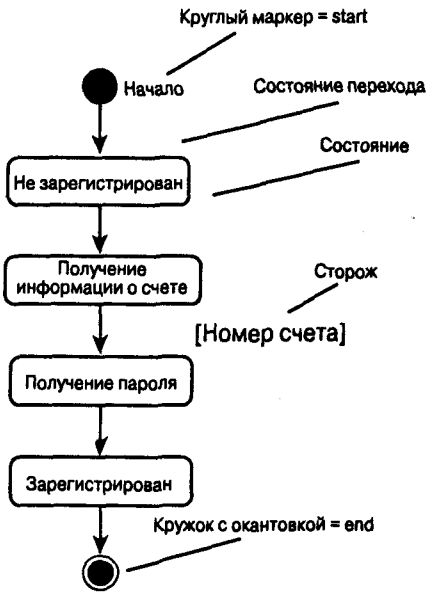


Рис. 18.23. Переходы состояний класса `Расчетный счет`

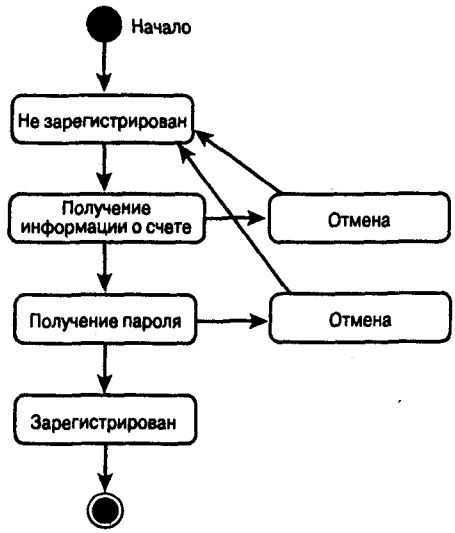


Рис. 18.24. Отмена регистрации

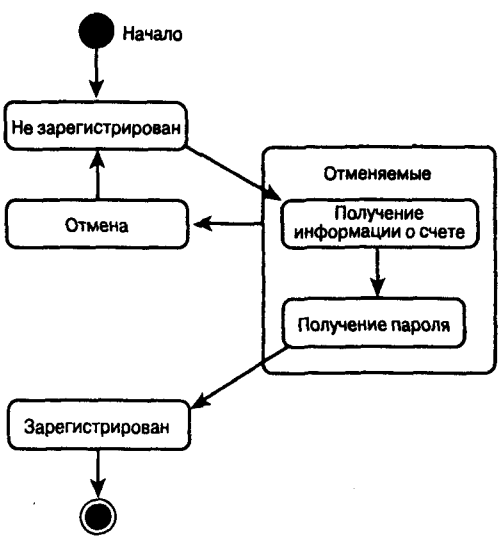


Рис. 18.25. Сверхсостояние

Резюме

На этом занятии в общих чертах рассмотрены вопросы анализа и проектирования объектно-ориентированных программ. Анализ состоит в определении ситуаций и сценариев использования программы, а проектирование заключается в определении классов и моделировании отношений и взаимодействия между ними.

Еще не так давно программист быстро набрасывал основные требования к программе и начинали писать код. Современные программы отличаются тем, что работа над ними никогда не заканчивается, если только проект не оказался нежизнеспособным и не был отвергнут. Тщательное планирование проекта в начале гарантирует возможность быстрой и безболезненной модернизации его в будущем.

На следующих занятиях рассматриваются средства реализации спланированных проектов. Вопросы тестирования и маркетинга программных продуктов выходят за пределы этой книги, хотя при составлении бизнес-плана их никак нельзя упускать.

Вопросы и ответы

Чем объектно-ориентированный анализ и проектирование фундаментально отличаются от других подходов?

До разработки объектно-ориентированной технологии аналитики и программисты были склонны думать о программах как о группах функций, работающих с данными. Объектно-ориентированное программирование рассматривает интегрированные данные и функции как самостоятельные единицы, содержащие в себе и данные, и методы манипулирования ими. При процедурном программировании внимание сконцентрировано на функциях и их работе с данными. Говорят, что программы на Pascal и C — коллекции процедур, а программы на C++ — коллекции классов.

Является ли объектно-ориентированное программирование той палочкой-выручалочкой, которая решит все проблемы программирования?

Нет, этого никогда и не ждали. Однако на современном уровне требования к программным продуктам объектно-ориентированные анализ, проектирование и программирование обеспечивают программистов средствами, которые не могло предоставить процедурное программирование.

Является ли C++ совершенным объектно-ориентированным языком?

C++, если сравнивать его с другими альтернативными объектно-ориентированными языками программирования, имеет множество преимуществ и недостатков. Но одно из безусловных преимуществ состоит в том, что это самый популярный объектно-ориентированный язык программирования на Земле. Откровенно говоря, большинство программистов решают работать на C++ не после изнурительного анализа альтернативных объектно-ориентированных языков. Они идут туда, где происходят основные события, а в 90-х основные события в мире программирования связаны с C++. Тому есть веские причины. Конечно, C++ может многое предложить программисту, но эта книга существует — и бьюсь об заклад, что вы читаете ее, — из-за того, что C++ выбран в качестве языка разработки в очень многих крупных корпорациях, таких как Microsoft.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний, а также ряд упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Какая разница между объектно-ориентированным и процедурным программированием?
2. Каковы этапы объектно-ориентированного анализа и проектирования?
3. Как связанны диаграммы последовательности и сотрудничества?

Упражнения

1. Предположим, что есть две пересекающиеся улицы с двусторонним движением, светофорами и пешеходными переходами. Нужно создать виртуальную модель, чтобы определить, позволит ли изменение частоты подачи сигнала светофора сделать дорожное движение более равномерным.
2. Какие объекты и какие классы потребуются для имитации этой ситуации?
3. Усложним ситуацию из упражнения 1. Предположим, что есть три вида водителей: таксисты, переезжающие переход на красный свет; иногородние, которые едут медленно и осторожно; и частники, которые ведут машины по-разному, в зависимости от представлений о своей "крутизне".
4. Также есть два вида пешеходов: местные, которые переходят улицу, где им заблагорассудится, и туристы, которые переходят улицу только на зеленый свет.
5. А кроме того, есть еще велосипедисты, которые ведут себя то как пешеходы, то как водители.
6. Как эти соображения изменят модель?
7. Вам заказали программу планирования времени конференций и встреч, а также бронирования мест в гостинице для визитеров компании и для участников конференций. Определите главные подсистемы.
8. Спроектируйте интерфейсы к классам той части программы, обсуждаемой в упражнении 3, которая относится к резервированию гостиничных номеров.

Шаблоны

У программистов, использующих язык C++, появился новый мощный инструмент — "параметризованные типы", или *шаблоны*. Шаблонами настолько удобно пользоваться, что стандартная библиотека шаблонов (Standard Template Library — STL) была принята в состав определений языка C++.

Итак, сегодня вы узнаете:

- Что такое шаблоны и как их использовать
- Как создать класс шаблонов
- Как создаются шаблоны функций
- Что представляет собой стандартная библиотека шаблонов (STL) и как ею пользоваться

Что такое шаблоны

При подведении итогов за вторую неделю обучения вы узнали, как построить объект `PartsList` и как его использовать для создания объекта `PartsCatalog`. Если же вы хотите воспользоваться объектом `PartsList`, чтобы составить, например, список кошек, у вас возникнет проблема: объект `PartsList` знает только о запчастях.

Чтобы решить эту проблему, можно создать базовый класс `List` и произвести из него классы `PartsList` и `CatsList`. Затем можно вырезать и вставить существенную часть класса `PartsList` в объявление нового класса `CatsList`. А через неделю, когда вы захотите составить список объектов `Car`, вам придется опять создавать новый класс и снова "вырезать и вставлять".

Очевидно, что это неприемлемое решение. Ведь через какое-то время класс `List` и его производные классы придется расширять. А работа, которую пришлось бы проделать, чтобы убедиться в том, что все изменения, коснувшиеся базового класса, распространены и на все связанные классы, превратилась бы в настоящий кошмар.

Благодаря шаблонам, эта проблема легко решается, а с принятием стандарта ANSI шаблоны стали неотъемлемой частью языка C++, подобно которому они сохраняют тип и очень гибки.

Параметризованные типы

С помощью шаблонов можно "научить" компилятор составлять список элементов любого типа, а не только заданного: `PartsList` — это список частей, `CatsList` — это список кошек. Единственное отличие между ними — тип элементов списка. При использовании шаблонов тип элементов списка становится параметром для определения класса.

Общим компонентом практически всех библиотек C++ является класс массивов. Как показано на примере с классом `List`, утомительно и крайне неэффективно создавать один класс массивов для целых, другой — для двойных слов, а еще один — для массива элементов типа `Animals`. Шаблоны позволяют объявить параметризованный класс массивов, а затем указать, какой тип объекта будет содержаться в каждом экземпляре массива. Заметьте, что стандартная библиотека шаблонов предоставляет стандартизированный набор *контейнерных* классов, включая массивы, списки и т.д. Сейчас мы выясняем, что нужно для создания вашего собственного класса, только для того, чтобы вы до конца поняли, как работают шаблоны; но в коммерческой программе вы почти стопроцентно будете использовать классы библиотеки STL, а не собственного изготовления.

Создание экземпляра шаблона

Экземпляризация (instantiation) — это операция создания определенного типа из шаблона. Отдельные классы называются экземплярами шаблона.

Параметризованные шаблоны (parameterized templates) предоставляют возможность создания общего класса и для построения конкретных экземпляров передают этому классу в качестве параметров типы данных.

Объявление шаблона

Объявляем параметризованный объект `Array` (шаблон для массива) путем записи следующих строк:

```
1: template <class T>     // объявляем шаблон и параметр
2: class Array            // параметризуемый класс
3: {
4:    public:
5:      Array();
6:   // здесь должно быть полное определение класса
7: } ;
```

Ключевое слово `template` используется в начале каждого объявления и определения класса шаблона. Параметры шаблона располагаются за ключевым словом `template`. Параметры — это элементы, которые изменяются с каждым экземпляром. Например, в приведенном выше шаблоне массивов будет изменяться тип объектов, сохраняемых в массиве. Один экземпляр шаблона может хранить массив целых чисел, а другой — массив объектов класса `Animals`.

В этом примере используется ключевое слово `class`, за которым следует идентификатор `T`. Это ключевое слово означает, что параметром является тип. Идентификатор `T` используется в остальной части определения шаблона, указывая тем самым на параметризованный тип. В одном экземпляре этого класса вместо идентификатора `T` повсюду будет стоять тип `int`, а в другом — тип `Cat`.

Чтобы объявить экземпляры параметризованного класса Array для типов int и Cat, следует написать:

```
Array<int> anIntArray;
Array<Cat> aCatArray;
```

Объект anIntArray представляет собой массив целых чисел, а объект aCatArray — массив элементов типа Cat. Теперь вы можете использовать тип Array<int> в любом месте, где обычно указывается какой-либо тип — для возвращаемого функцией значения, для параметра функции и т.д. В листинге 19.1 содержится полное объявление уже рассмотренного нами шаблона Array.

Программа в листинге 19.1 не завершена!

Листинг 19.1. Шаблон класса Array

```
1: //Листинг 19.1. Шаблон класса массивов
2:     #include <iostream.h>
3:     const int DefaultSize = 10;
4:
5:     template <class T>  // объявляем шаблон и параметр
6:     class Array         // параметризуемый класс
7:     {
8:     public:
9:        // конструкторы
10:       Array(int itsSize = DefaultSize);
11:       Array(const Array &rhs);
12:       ~Array() {  delete [] pType; }
13:
14:       // операторы
15:       Array& operator=(const Array&);
16:       T& operator[](int offSet) {  return pType[offSet]; }
17:
18:       // методы доступа
19:       int getSize() {  return itsSize; }
20:
21:    private:
22:       T *pType;
23:       int  itsSize;
24:    } ;
```

Результатов нет. Эта программа не завершена.

Определение шаблона начинается в строке 5 с ключевого слова template, за которым следует параметр. В данном случае параметр идентифицируется как тип за счет использования ключевого слова class, а идентификатор T используется для представления параметризованного типа.

Со строки 6 и до конца определения шаблона (строка 24) вся остальная часть объявления аналогична любому другому объявлению класса. Единственное отличие заключается в том, что везде, где обычно должен стоять тип объекта, используется идентификатор T. Например, можно предположить, что operator[] должен возвращать ссылку на объект в массиве, а на самом деле он объявляется для возврата ссылки на идентификатор типа T.

Если объявлен экземпляр целочисленного массива, перегруженный оператор присваивания этого класса возвратит ссылку на тип integer. А при объявлении экземпляра массива Animal оператор присваивания возвратит ссылку на объект типа Animal.

Использование имени шаблона

Внутри объявления класса слово Array может использоваться без спецификаторов. В другом месте программы этот класс будет упоминаться как Array<T>. Например, если не поместить конструктор внутри объявления класса, то вы должны записать следующее:

```
template <class T>
Array<T>::Array(int size):
itsSize = size
{
    pType = new T[size];
    for (int i = 0; i<size; i++)
    pType[i] = 0;
}
```

Объявление, занимающее первую строку этого фрагмента кода, устанавливает в качестве параметра тип данных (class T). В таком случае в программе на шаблон можно ссылаться как Array<T>, а объявленную функцию-член вызывать строкой Array(int size).

Остальная часть функции имеет такой же вид, какой мог быть у любой другой функции. Это обычный и предпочтительный метод создания класса и его функций путем простого объявления до включения в шаблон.

Выполнение шаблона

Для выполнения класса шаблона Array необходимо создать конструктор-копировщик, перегрузить оператор присваивания (operator=) и т.д. В листинге 19.2 показана простая консольная программа, предназначенная для выполнения этого шаблона.

Некоторые более старые компиляторы не поддерживают использование шаблонов. Но шаблоны являются частью стандарта ANSI C++, поэтому компиляторы всех основных производителей поддерживают шаблоны в своих текущих версиях. Если у вас очень старый компилятор, вы не сможете компилировать и выполнять упражнения, приведенные в этой главе. Однако все же стоит прочитать ее до конца, а затем вернуться к этому материалу после модернизации своего компилятора.

Листинг 19.2. Использование шаблона массива

```
1:    #include <iostream.h>
2:
3:    const int DefaultSize = 10;
4:
5:    // обычный класс Animal для
6:    // создания массива животных
7:
8:    class Animal
9:    {
10:   public:
11:      Animal(int);
12:      Animal();
13:      ~Animal() { }
14:      int GetWeight() const {  return itsWeight; }
15:      void Display() const {  cout << itsWeight; }
16:   private:
17:      int itsWeight;
18:   } ;
19:
20:   Animal::Animal(int weight):
21:   itsWeight(weight)
22:   { }
23:
24:   Animal::Animal():
25:   itsWeight(0)
26:   { }
27:
28:
29:   template <class T>  // объявляем шаблон и параметр
30:   class Array          // параметризованный класс
31:   {
32:   public:
33:      // конструкторы
34:      Array(int itsSize = DefaultSize);
35:      Array(const Array &rhs);
36:      ~Array() {  delete [] pType; }
37:
38:      // операторы
39:      Array& operator=(const Array&);
40:      T& operator[](int offSet) {  return pType[offSet]; }
41:      const T& operator[](int offSet) const
42:          {  return pType[offSet]; }
43:      // методы доступа
44:      int GetSize() const {  return itsSize; }
45:
46:   private:
47:      T *pType;
48:      int  itsSize;
49:   } ;
```

```
50:
51:    // выполнения...
52:
53:    // выполняем конструктор
54:    template <class T>
55:    Array<T>::Array(int size):
56:    itsSize(size)
57:    {
58:       pType = new T[size];
59:       for (int i = 0; i<size; i++)
60:          pType[i] = 0;
61:    }
62:
63:    // конструктор-копировщик
64:    template <class T>
65:    Array<T>::Array(const Array &rhs)
66:    {
67:       itsSize = rhs.GetSize();
68:       pType = new T[itsSize];
69:       for (int i = 0; i<itsSize; i++)
70:          pType[i] = rhs[i];
71:    }
72:
73:    // оператор присваивания
74:    template <class T>
75:    Array<T>& Array<T>::operator=(const Array &rhs)
76:    {
77:       if (this == &rhs)
78:          return *this;
79:       delete [] pType;
80:       itsSize = rhs.GetSize();
81:       pType = new T[itsSize];
82:       for (int i = 0; i<itsSize; i++)
83:          pType[i] = rhs[i];
84:       return *this;
85:    }
86:
87:    // исполняемая программа
88:    int main()
89:    {
90:       Array<int> theArray;       // массив целых
91:       Array<Animal> theZoo;      // массив животных
92:       Animal *pAnimal;
93:
94:       // заполняем массивы
95:       for (int i = 0; i < theArray.GetSize(); i++)
96:       {
97:          theArray[i] = i*2;
98:          pAnimal = new Animal(i*3);
99:          theZoo[i] = *pAnimal;
```

```
100:            delete pAnimal;
101:        }
102:        // выводим на печать содержимое массивов
103:        for (int j = 0; j < theArray.GetSize(); j++)
104:        {
105:           cout << "theArray[" << j << "]:\ t";
106:           cout << theArray[j] << "\ t\ t";
107:           cout << "theZoo[" << j << "]:\ t";
108:           theZoo[j].Display();
109:           cout << endl;
110:        }
111:
112:        return 0;
113:    }
```

РЕЗУЛЬТАТ

```
theArray[0]:    0          theZoo[0]:    0
theArray[1]:    2          theZoo[1]:    3
theArray[2]:    4          theZoo[2]:    6
theArray[3]:    6          theZoo[3]:    9
theArray[4]:    8          theZoo[4]:    12
theArray[5]:    10         theZoo[5]:    15
theArray[6]:    12         theZoo[6]:    18
theArray[7]:    14         theZoo[7]:    21
theArray[8]:    16         theZoo[8]:    24
theArray[9]:    18         theZoo[9]:    27
```

АНАЛИЗ В строках 8–26 выполняется создание класса `Animal`, благодаря которому объекты определяемого пользователем типа можно будет добавлять в массив.

Содержимое строки 29 означает, что в следующих за ней строках объявляется шаблон, параметром для которого является тип, обозначенный идентификатором `T`. Класс `Array` содержит два конструктора, причем первый конструктор принимает размер и по умолчанию устанавливает его равным значению целочисленной константы `DefaultSize`.

Затем объявляются операторы присваивания и индексирования, причем объявляются константная и неконстантная версии оператора индексирования. В качестве единственного метода доступа служит функция `GetSize()`, которая возвращает размер массива.

Можно, конечно, представить себе и более полный интерфейс. Ведь для любой серьезной программы создания массива представленный здесь вариант будет недостаточным. Как минимум, пришлось бы добавить операторы, предназначенные для удаления элементов, для распаковки и упаковки массива и т.д. Все это предусмотрено классами контейнеров библиотеки STL, но к этому мы вернемся в конце занятия.

Раздел закрытых данных содержит переменные-члены размера массива и указатель на массив объектов, реально помещенных в память.

Функции шаблона

Если вы хотите передать функции объект массива, нужно передавать конкретный экземпляр массива, а не шаблон. Поэтому, если некоторая функция `SomeFunction()` принимает в качестве параметра целочисленный массив, используйте следующую запись:

```
void SomeFunction(Array<int>&);     // правильно
```

А запись

```
void SomeFunction(Array<T>&);    // ошибка!
```

неверна, поскольку отсюда не ясно, что представляет собой выражение T&. Запись

```
void SomeFunction(Array &);      // ошибка!
```

тоже ошибочна, так как объекта класса Array не существует — есть только шаблон и его экземпляры.

Чтобы реализовать более общий подход использования объектов, созданных на основе шаблона, нужно объявить *функцию шаблона*:

```
template <class T>
void MyTemplateFunction(Array<T>&);    // верно
```

Здесь MyTemplateFunction() объявлена как функция шаблона, на что указывает первая строка объявления. Заметьте, что функции шаблонов, подобно другим функциям, могут иметь любые имена.

Функции шаблонов, помимо объектов, заданных в параметризованной форме, могут также принимать и экземпляры шаблона. Проиллюстрируем это на примере:

```
template <class T>
void MyOtherFunction(Array<T>&, Array<int>&);    // верно
```

Обратите внимание на то, что эта функция принимает два массива: параметризованный массив и массив целых чисел. Первый может быть массивом любых объектов, а второй — только массивом целых чисел.

Шаблоны и друзья

В шаблонах классов могут быть объявлены три типа друзей:

- дружественный класс или функция, не являющиеся шаблоном;
- дружественный шаблон класса или функция, входящая в шаблон;
- дружественный шаблон класса или шаблонная функция, специализированные по типу данных.

Дружественные классы и функции, не являющиеся шаблонами

Можно объявить любой класс или функцию, которые будут дружественны по отношению к вашему классу шаблона. В этом случае каждый экземпляр класса будет обращаться с другом так, как будто объявление класса-друга было сделано в этом конкретном экземпляре. В листинге 19.3 в определении шаблона класса Array добавлена тривиальная дружественная функция Intrude(), а в управляющей программе делается вызов этой функции. В качестве друга функция Intrude() получает доступ к закрытым данным класса Array. Но поскольку эта функция не является функцией шаблона, то ее можно вызывать только для массива заданного типа (в нашем примере для массива целых чисел).

Листинг 19.3. Функция-друг, не являющаяся шаблоном

```
1:    // Листинг 19.3. Использование в шаблонах функций-друзей определенного типа
2:
3:    #include <iostream.h>
4:
5:    const int DefaultSize = 10;
6:
7:    // объявляем простой класс Animal, чтобы можно
8:    // было создать массив животных
9:
10:    class Animal
11:    {
12:    public:
13:       Animal(int);
14:       Animal();
15:       ~Animal() { }
16:       int GetWeight() const {  return itsWeight; }
17:       void Display() const {  cout << itsWeight; }
18:    private:
19:       int itsWeight;
20:    } ;
21:
22:    Animal::Animal(int weight):
23:    itsWeight(weight)
24:    { }
25:
26:    Animal::Animal():
27:    itsWeight(0)
28:    { }
29:
30:      template <class T>  // объявляем шаблон и параметр
31:      class Array          // параметризованный класс
32:      {
33:      public:
34:         // конструкторы
35:         Array(int itsSize = DefaultSize);
36:         Array(const Array &rhs);
37:        ~Array() {  delete [] pType; }
38:
39:        // операторы
40:        Array& operator=(const Array&);
41:        T& operator[](int offSet) { return pType[offSet]; }
42:        const T& operator[](int offSet) const
43:            {  return pType[offSet]; }
44:        // методы доступа
45:        int GetSize() const {  return itsSize; }
46:
47:       // функция-друг
48:       friend void Intrude(Array<int>);
```

```
49:
50:     private:
51:        T *pType;
52:        int  itsSize;
53:     } ;
54:
55:       // Поскольку функция-друг не является шаблоном, ее можно использовать только
56:       // с массивами целых чисел! Но она получает доступ к закрытым данным класса.
57:      void Intrude(Array<int> theArray)
58:      {
59:       cout << "\ n*** Intrude ***\ n";
60:       for (int i = 0; i < theArray.itsSize; i++)
61:          cout << "i: " <<    theArray.pType[i] << endl;
62:       cout << "\ n";
63:      }
64:
65:    // Ряд выполнений...
66:
67:    // выполнение конструктора
68:    template <class T>
69:    Array<T>::Array(int size):
70:    itsSize(size)
71:    {
72:       pType = new T[size];
73:       for (int i = 0; i<size; i++)
74:          pType[i] = 0;
75:    }
76:
77:    // конструктор-копировщик
78:    template <class T>
79:    Array<T>::Array(const Array &rhs)
80:    {
81:       itsSize = rhs.GetSize();
82:       pType = new T[itsSize];
83:       for (int i = 0; i<itsSize; i++)
84:          pType[i] = rhs[i];
85:    }
86:
87:    // перегрузка оператора присваивания (=)
88:    template <class T>
89:    Array<T>& Array<T>::operator=(const Array &rhs)
90:    {
91:       if (this == &rhs)
92:          return *this;
93:       delete [] pType;
94:       itsSize = rhs.GetSize();
95:       pType = new T[itsSize];
96:       for (int i = 0; i<itsSize; i++)
97:          pType[i] = rhs[i];
98:       return *this;
```

```
}

// управляющая программа
int main()
{
   Array<int> theArray;       // массив целых
   Array<Animal> theZoo;      // массив животных
   Animal *pAnimal;

   // заполняем массивы
   for (int i = 0; i < theArray.GetSize(); i++)
   {
      theArray[i] = i*2;
      pAnimal = new Animal(i*3);
      theZoo[i] = *pAnimal;
   }

   int j;
   for (j = 0; j < theArray.GetSize(); j++)
   {
      cout << "theZoo[" << j << "]:\ t";
      theZoo[j].Display();
      cout << endl;
   }
   cout << "Now use the friend function to";
   cout << "find the members of Array<int>";
   Intrude(theArray);

   cout << "\ n\ nDone.\ n";
   return 0;
}
```

РЕЗУЛЬТАТ

```
theZoo[0]:      0
theZoo[1]:      3
theZoo[2]:      6
theZoo[3]:      9
theZoo[4]:      12
theZoo[5]:      15
theZoo[6]:      18
theZoo[7]:      21
theZoo[8]:      24
theZoo[9]:      27
Now use the friend function to find the members of Array<int>
*** Intrude ***
i: 0
i: 2
i: 4
i: 6
i: 8
```

```
i: 10
i: 12
i: 14
i: 16
i: 18

Done.
```

Анализ Объявление шаблона `Array` было расширено за счет включения дружественной функции `Intrude()`. Это объявление означает, что каждый экземпляр массива типа `int` будет считать функцию `Intrude()` дружественной, а следовательно, она будет иметь доступ к закрытым переменным-членам и функциям-членам экземпляра этого массива.

В строке 60 функция `Intrude()` непосредственно обращается к члену `itsSize`, а в строке 61 получает прямой доступ к переменной-члену `pType`. В данном случае без использования функции-друга можно было бы обойтись, поскольку класс `Array` предоставляет открытые методы доступа к этим данным. Этот листинг служит лишь примером того, как можно объявлять и использовать функции-друзья шаблонов.

Дружественный класс или функция как общий шаблон

В класс `Array` было бы весьма полезно добавить оператор вывода данных. Это можно сделать путем объявления оператора вывода для каждого возможного типа массива, но такой подход свел бы не нет саму идею использования класса `Array` как шаблона.

Поэтому нужно найти другое решение. Попробуем добиться того, чтобы оператор вывода работал независимо от типа экземпляра массива.

```
ostream& operator<< (ostream&, Array<T>&);
```

Чтобы этот оператор работал, нужно так объявить `operator<<`, чтобы он стал функцией шаблона:

```
template <class T> ostream& operator<< (ostream&, Array<T>&)
```

Теперь `operator<<` является функцией шаблона и его можно использовать в выполнении класса. В листинге 19.4 показано объявление шаблона `Array`, дополненное объявлением функции оператора вывода `operator<<`.

Листинг 19.4. Использование оператора вывода

```
1:    #include <iostream.h>
2:
3:    const int DefaultSize = 10;
4:
5:    class Animal
6:    {
7:    public:
8:       Animal(int);
9:       Animal();
10:      ~Animal() { }
11:      int GetWeight() const {  return itsWeight; }
12:      void Display() const {  cout << itsWeight; }
```

```
13:    private:
14:       int itsWeight;
15:    } ;
16:
17:    Animal::Animal(int weight):
18:    itsWeight(weight)
19:    { }
20:
21:    Animal::Animal():
22:    itsWeight(0)
23:    { }
24:
25:     template <class T>  // объявляем шаблон и параметр
26:     class Array            // параметризованный класс
27:     {
28:     public:
29:        // конструкторы
30:       Array(int itsSize = DefaultSize);
31:       Array(const Array &rhs);
32:       ~Array() {  delete [] pType; }
33:
34:       // операторы
35:       Array& operator=(const Array&);
36:       T& operator[](int offSet) {  return pType[offSet]; }
37:       const T& operator[](int offSet) const
38:        {  return pType[offSet]; }
39:       // методы доступа
40:       int GetSize() const {  return itsSize; }
41:
42:       friend ostream& operator<< (ostream&, Array<T>&);
43:
44:    private:
45:       T *pType;
46:       int  itsSize;
47:    } ;
48:
49:    template <class T>
50:    ostream& operator<< (ostream& output, Array<T>& theArray)
51:    {
52:       for (int i = 0; i<theArray.GetSize(); i++)
53:          output << "[" << i << "] " << theArray[i] << endl; return output;
54:    }
55:
56:    // Ряд выполнений...
57:
58:    // выполнение конструктора
59:    template <class T>
60:    Array<T>::Array(int size):
61:    itsSize(size)
62:    {
```

```
63:       pType = new T[size];
64:       for (int i = 0; i<size; i++)
65:          pType[i] = 0;
66:    }
67:
68:    // конструктор-копировщик
69:    template <class T>
70:    Array<T>::Array(const Array &rhs)
71:    {
72:       itsSize = rhs.GetSize();
73:       pType = new T[itsSize];
74:       for (int i = 0; i<itsSize; i++)
75:          pType[i] = rhs[i];
76:    }
77:
78:    // перегрузка оператора присваивания (=)
79:    template <class T>
80:    Array<T>& Array<T>::operator=(const Array &rhs)
81:    {
82:       if (this == &rhs)
83:          return *this;
84:       delete [] pType;
85:       itsSize = rhs.GetSize();
86:       pType = new T[itsSize];
87:       for (int i = 0; i<itsSize; i++)
88:          pType[i] = rhs[i];
89:       return *this;
90:    }
91:
92:    int main()
93:    {
94:       bool Stop = false;        // признак для цикла
95:       int offset, value;
96:       Array<int> theArray;
97:
98:       while (!Stop)
99:       {
100:          cout << "Enter an offset (0-9) ";
101:          cout << "and a value. (-1 to stop): " ;
102:          cin >> offset >> value;
103:
104:          if (offset < 0)
105:             break;
106:
107:          if (offset > 9)
108:          {
109:             cout << "***Please use values between 0 and 9.***\ n";
110:             continue;
111:          }
112:
```

```
113:          theArray[offset] = value;
114:       }
115:
116:       cout << "\ nHere's the entire array:\ n";
117:       cout << theArray << endl;
118:     return 0;
119:    }
```

РЕЗУЛЬТАТ

```
Enter an offset (0-9) and a value. (-1 to stop): 1 10
Enter an offset (0-9) and a value. (-1 to stop): 2 20
Enter an offset (0-9) and a value. (-1 to stop): 3 30
Enter an offset (0-9) and a value. (-1 to stop): 4 40
Enter an offset (0-9) and a value. (-1 to stop): 5 50
Enter an offset (0-9) and a value. (-1 to stop): 6 60
Enter an offset (0-9) and a value. (-1 to stop): 7 70
Enter an offset (0-9) and a value. (-1 to stop): 8 80
Enter an offset (0-9) and a value. (-1 to stop): 9 90
Enter an offset (0-9) and a value. (-1 to stop): 10 10
***Please use values between 0 and 9.***
Enter an offset (0-9) and a value. (-1 to stop): -1 -1

Here's the entire array:
[0] 0
[1] 10
[2] 20
[3] 30
[4] 40
[5] 50
[6] 60
[7] 70
[8] 80
[9] 90
```

АНАЛИЗ В строке 42 объявляется шаблон функции `operator<<()` в качестве друга шаблона класса `Array`. Поскольку `operator<<()` реализован в виде функции шаблона, то каждый экземпляр этого типа параметризованного массива будет автоматически иметь функцию `operator<<()` для вывода данных соответствующего типа. Выполнение этого оператора начинается в строке 49. Каждый член массива вызывается по очереди. Этот метод работает только в том случае, если функция `operator<<()` определена для каждого типа объекта, сохраняемого в массиве.

Использование экземпляров шаблона

С экземплярами шаблона можно обращаться так же, как с любыми другими типами данных. Их можно передавать в функции как ссылки или как значения и возвращать как результат выполнения функции (тоже как ссылки или как значения). Способы передачи экземпляров шаблона показаны в листинге 19.5.

Листинг 19.5. Передача в функцию экземпляра шаблона

```
1:    #include <iostream.h>
2:
3:    const int DefaultSize = 10;
4:
5:    // Обычный класс, из объектов которого будет состоять массив
6:    class Animal
7:    {
8:    public:
9:    // конструкторы
10:        Animal(int);
11:        Animal();
12:        ~Animal();
13:
14:        // методы доступа
15:        int GetWeight() const {  return itsWeight; }
16:        void SetWeight(int theWeight) {  itsWeight = theWeight; }
17:
18:         // дружественные операторы
19:        friend ostream& operator<< (ostream&, const Animal&);
20:
21:   private:
22:        int itsWeight;
23:   } ;
24:
25:   // оператор вывода объектов типа Animal
26:   ostream& operator<<
27:       (ostream& theStream, const Animal& theAnimal)
28   {
29:   theStream << theAnimal.GetWeight();
30:   return theStream;
31:   }
32:
33:   Animal::Animal(int weight):
34:   itsWeight(weight)
35:   {
36:      // cout << "Animal(int)\ n";
37:   }
38:
39:   Animal::Animal():
40:   itsWeight(0)
41:   {
42:      // cout << "Animal()\ n";
43:   }
44:
45:   Animal::~Animal()
46:   {
47:     // cout << "Destroyed an animal...\ n";
48:   }
```

```
49:
50:    template <class T>  // объявление шаблона и параметра
51:    class Array            // параметризованный класс
52:    {
53:    public:
54:       Array(int itsSize = DefaultSize);
55:       Array(const Array &rhs);
56:       ~Array() {  delete [] pType; }
57:
58:       Array& operator=(const Array&);
59:       T& operator[](int offSet) {  return pType[offSet]; }
60:       const T& operator[](int offSet) const
61:          {  return pType[offSet]; }
62:       int GetSize() const {  return itsSize; }
63
64:      // функция-друг
65:      friend ostream& operator<< (ostream&, const Array<T>&);
66:
67:    private:
68:       T *pType;
69:       int  itsSize;
70:    } ;
71:
70:    template <class T>
72:    ostream& operator<< (ostream& output, const Array<T>& theArray)
73:    {
74:       for (int i = 0; i<theArray.GetSize(); i++)
75:          output << "[" << i << "] " << theArray[i] << endl;
76:       return output;
77:    }
78:
79:    // Ряд выполнений...
80:
81:    // выполнение конструктора
82:    template <class T>
83:    Array<T>::Array(int size):
84:    itsSize(size)
85:    {
86:       pType = new T[size];
87:       for (int i = 0; i<size; i++)
88:          pType[i] = 0;
89:    }
90:
91:    // конструктор-копировщик
92:    template <class T>
93:    Array<T>::Array(const Array &rhs)
94:    {
95:       itsSize = rhs.GetSize();
96:       pType = new T[itsSize];
97:       for (int i = 0; i<itsSize; i++)
```

```
98:          pType[i] = rhs[i];
99:      }
100:
101:     void IntFillFunction(Array<int>& theArray);
102:     void AnimalFillFunction(Array<Animal>& theArray);
103:
104:     int main()
105:     {
106:        Array<int> intArray;
107:        Array<Animal> animalArray;
108:        IntFillFunction(intArray);
109:        AnimalFillFunction(animalArray);
110:        cout << "intArray...\ n" << intArray;
111:        cout << "\ nanimalArray...\ n" << animalArray << endl;
112:        return 0;
113:     }
114:
115:     void IntFillFunction(Array<int>& theArray)
116:     {
117:        bool Stop = false;
118:        int offset, value;
119:        while (!Stop)
120:        {
121:           cout << "Enter an offset (0-9) ";
122:           cout << "and a value. (-1 to stop): " ;
123:           cin >> offset >> value;
124:          if (offset < 0)
125:             break;
126:          if (offset > 9)
127:          {
128:             cout << "***Please use values between 0 and 9.***\ n";
129:             continue;
130:          }
131:          theArray[offset] = value;
132:       }
133:    }
134:
135:
136:    void AnimalFillFunction(Array<Animal>& theArray)
137:    {
138:       Animal * pAnimal;
139:       for (int i = 0; i<theArray.GetSize(); i++)
140:       {
141:          pAnimal = new Animal;
142:          pAnimal->SetWeight(i*100);
143:          theArray[i] = *pAnimal;
144:          delete pAnimal;  // копия была помещена в массив
145:       }
146:    }
```

РЕЗУЛЬТАТ

```
Enter an offset (0-9) and a value. (-1 to stop): 1 10
Enter an offset (0-9) and a value. (-1 to stop): 2 20
Enter an offset (0-9) and a value. (-1 to stop): 3 30
Enter an offset (0-9) and a value. (-1 to stop): 4 40
Enter an offset (0-9) and a value. (-1 to stop): 5 50
Enter an offset (0-9) and a value. (-1 to stop): 6 60
Enter an offset (0-9) and a value. (-1 to stop): 7 70
Enter an offset (0-9) and a value. (-1 to stop): 8 80
Enter an offset (0-9) and a value. (-1 to stop): 9 90
Enter an offset (0-9) and a value. (-1 to stop): 10 10
***Please use values between 0 and 9.***
Enter an offset (0-9) and a value. (-1 to stop): -1 -1

intArray:...
[0] 0
[1] 10
[2] 20
[3] 30
[4] 40
[5] 50
[6] 60
[7] 70
[8] 80
[9] 90

animalArray:...
[0] 0
[1] 100
[2] 200
[3] 300
[4] 400
[5] 500
[6] 600
[7] 700
[8] 800
[9] 900
```

АНАЛИЗ

В целях экономии места большая часть выполнения класса `Array` не показана в этом листинге. Класс `Animal` объявляется в строках 6–23. И хотя структура этого класса предельно упрощена, тем не менее в нем содержится собственный оператор вывода (<<), позволяющий выводить на экран объекты массива типа `Animal`.

Обратите внимание, что в классе `Animal` объявлен конструктор по умолчанию (конструктор без параметров, который еще называют *стандартным*). Без этого объявления нельзя обойтись, поскольку при добавлении объекта в массив используется конструктор по умолчанию данного объекта. При этом возникают определенные трудности, о которых речь пойдет ниже.

В строке 101 объявляется функция `IntFillFunction()`, параметром которой является целочисленный массив. Обратите внимание, что эта функция не принадлежит шаблону, поэтому может принять только массив целочисленных значений. Аналогичным

образом в строке 102 объявляется функция AnimalFillFunction(), которая принимает массив объектов типа Animal.

Эти функции выполняются по-разному, поскольку заполнение массива целых чисел отличается от заполнения массива объектов Animal.

Специализированные функции

Если разблокировать выражения вывода на экран в конструкторах и деструкторе класса Animal (см. листинг 19.5), то обнаружится, что конструктор и деструктор объектов Animal вызываются значительно чаще, чем ожидалось.

При добавлении объекта в массив вызывается стандартный конструктор объекта. Однако конструктор класса Array также используется для присвоения нулевых значений каждому члену массива, как показано в строках 59 и 60 листинга 19.2.

В выражении someAnimal = (Animal) 0; вызывается стандартный оператор operator= для класса Animal. Это приводит к созданию временного объекта Animal с помощью конструктора, который принимает целое число (нуль). Этот временный объект выступает правым операндом в операции присваивания, после чего удаляется деструктором.

Такой подход крайне неэффективен, поскольку объект Animal уже инициализирован должным образом. Однако эту строку нельзя удалить, потому что при создании массива целочисленные значения не будут автоматически инициализироваться нулевыми значениями. Выход состоит в том, чтобы объявить в шаблоне дополнительный специализированный конструктор для создания массива объектов Animal.

Эта идея реализована в листинге 19.6 путем явного выполнения класса Animal.

Листинг 19.6. Специальные реализации шаблона

```
1:    #include <iostream.h>
2:
3:    const int DefaultSize = 3;
4:
5:    // Обычный класс, из объектов которого создается массив
6:      class Animal
7:      {
8:      public:
9:         // конструкторы
10:        Animal(int);
11:        Animal();
12:        ~Animal();
13:
14:        // методы доступа
15:        int GetWeight() const {  return itsWeight; }
16:        void SetWeight(int theWeight) {  itsWeight = theWeight; }
17:
18:        // дружественные операторы
19:        friend ostream& operator<< (ostream&, const Animal&);
20:
21:     private:
22:        int itsWeight;
23:     } ;
24:
```

```
25:      // оператор вывода объектов типа Animal
26:     ostream& operator<<
27:         (ostream& theStream, const Animal& theAnimal)
28:     {
29:       theStream << theAnimal.GetWeight();
30:       return theStream;
31:     }
32:
33:     Animal::Animal(int weight):
34:     itsWeight(weight)
35:     {
36:        cout << "animal(int) ";
37:     }
38:
39:     Animal::Animal():
40:     itsWeight(0)
41:     {
42:        cout << "animal() ";
43:     }
44:
45:     Animal::~Animal()
46:     {
47:       cout << "Destroyed an animal...";
48:     }
49:
50:    template <class T>  // объявляем шаблон и параметр
51:    class Array         // параметризованный класс
52:    {
53:    public:
54:       Array(int itsSize = DefaultSize);
55:       Array(const Array &rhs);
56:       ~Array() {  delete [] pType; }
57:
58:       // операторы
59:       Array& operator=(const Array&);
60:       T& operator[](int offSet) { return pType[offSet]; }
61:       const T& operator[](int offSet) const
62:          {  return pType[offSet]; }
63:
64:       // методы доступа
65:       int GetSize() const {  return itsSize; }
66:
67:       // функция-друг
68:      friend ostream& operator<< (ostream&, const Array<T>&);
69:
70:    private:
71:       T *pType;
72:       int  itsSize;
73:    } ;
```

```
74:    template <class T>
75:    Array<T>::Array(int size = DefaultSize):
76:    itsSize(size)
77:    {
78:       pType = new T[size];
79:       for (int i = 0; i<size; i++)
80:         pType[i] = (T)0;
81:    }
82:
83:    template <class T>
84:    Array<T>& Array<T>::operator=(const Array &rhs)
85:    {
86:       if (this == &rhs)
87:          return *this;
88:       delete [] pType;
89:       itsSize = rhs.GetSize();
90:       pType = new T[itsSize];
91:       for (int i = 0; i<itsSize; i++)
92:          pType[i] = rhs[i];
93:       return *this;
94:    }
95:    template <class T>
96:    Array<T>::Array(const Array &rhs)
97:    {
98:       itsSize = rhs.GetSize();
99:       pType = new T[itsSize];
100:      for (int i = 0; i<itsSize; i++)
101:         pType[i] = rhs[i];
102:   }
103:
104:
105:   template <class T>
106:   ostream& operator<< (ostream& output, const Array<T>& theArray)
107:   {
108:      for (int i = 0; i<theArray.GetSize(); i++)
109:         output << "[" << i << "] " << theArray[i] << endl;
110:      return output;
111:   }
112:
113:
114:   Array<Animal>::Array(int AnimalArraySize):
115:   itsSize(AnimalArraySize)
116:   {
117:      pType = new Animal[AnimalArraySize];
118:   }
119:
120:
121:   void IntFillFunction(Array<int>& theArray);
122:   void AnimalFillFunction(Array<Animal>& theArray);
123:
```

```
int main()
{
   Array<int> intArray;
   Array<Animal> animalArray;
   IntFillFunction(intArray);
   AnimalFillFunction(animalArray);
   cout << "intArray...\ n" << intArray;
   cout << "\ nanimalArray...\ n" << animalArray << endl;
  return 0;
}

void IntFillFunction(Array<int>& theArray)
{
   bool Stop = false;
   int offset, value;
   while (!Stop)
   {
      cout << "Enter an offset (0-9) and a value. ";
      cout << "(-1 to stop): " ;
      cin >> offset >> value;
      if (offset < 0)
         break;
      if (offset > 9)
      {
         cout << "***Please use values between 0 and 9.***\ n";
         continue;
      }
      theArray[offset] = value;
   }
}


void AnimalFillFunction(Array<Animal>& theArray)
{
   Animal * pAnimal;
   for (int i = 0; i<theArray.GetSize(); i++)
   {
      pAnimal = new Animal(i*10);
      theArray[i] = *pAnimal;
      delete pAnimal;
   }
}
```

Для облегчения анализа в приведенные ниже результаты работы программы добавлены номера строк, но в действительности они не выводятся.

РЕЗУЛЬТАТ

```
1:  animal() animal() animal() Enter an offset (0-9) and a value. (-1 to
stop): 0 0
2:  Enter an offset (0-9) and a value. (-1 to stop): 1 1
3:  Enter an offset (0-9) and a value. (-1 to stop): 2 2
4:  Enter an offset (0-9) and a value. (-1 to stop): 3 3
5:  Enter an offset (0-9) and a value. (-1 to stop): -1 -1
6:  animal(int) Destroyed an animal...animal(int) Destroyed an
animal...animal(int) Destroyed an animal...initArray...
7: [0] 0
8: [1] 1
9: [2] 2
10:
11: animal array...
12: [0] 0
13: [1] 10
14: [2] 20
15:
16: Destroyed an animal...Destroyed an animal...Destroyed an animal...
17: <<< Second run >>>
18: animal(int)  Destroyed an animal...
19: animal(int)  Destroyed an animal...
20: animal(int)  Destroyed an animal...
21: Enter an offset (0-9) and a value. (-1 to stop): 0 0
22: Enter an offset (0-9) and a value. (-1 to stop): 1 1
23: Enter an offset (0-9) and a value. (-1 to stop): 2 2
24: Enter an offset (0-9) and a value. (-1 to stop): 3 3
25: animal(int)
26: Destroyed an animal...
27: animal(int)
28: Destroyed an animal...
29: animal(int)
30: Destroyed an animal...
31: initArray...
32: [0] 0
33: [1] 1
34: [2] 2
35:
36: animal array...
37: [0] 0
38: [1] 10
39: [2] 20
40:
41: Destroyed an animal...
42: Destroyed an animal...
43: Destroyed an animal...
```

АНАЛИЗ

В листинге 19.6 оба класса воспроизведены во всей своей полноте, чтобы лучше наблюдать за созданием и удалением временных объектов `Animal`. Для упрощения результатов работы значение `DefaultSize` было уменьшено до 3.

Все конструкторы и деструкторы класса `Animal` (строки 33–48) выводят на экран сообщения, сигнализирующие об их вызове.

В строках 74–81 объявляется конструктор класса Array. В строках 114–118 показан специализированный конструктор Array для массива объектов типа Animal. Обратите внимание, что в этом специализированном конструкторе не делается никаких явных присвоений и исходные значения для каждого объекта Animal устанавливаются стандартным конструктором.

При первом выполнении этой программы на экран выводится ряд сообщений. В строке 1 результатов выполнения программы зафиксированы сообщения трех стандартных конструкторов, вызванных при создании массива. Затем пользователь вводит четыре числа, которые помещаются в массив целых чисел.

После этого управление передается функции AnimalFillFunction(). Здесь в области динамического обмена создается временный объект Animal (строка 161), а его значение используется для модификации объекта Animal в массиве (строка 162). В следующей же строке (с номером 163) временный объект Animal удаляется. Этот процесс повторяется для каждого члена массива и отражен в строке 6 результатов выполнения программы.

В конце программы созданные массивы удаляются, а при вызове их деструкторов также удаляются и все их объекты. Процесс удаления отражен в строке 16 результатов выполнения программы.

При следующем выполнении программы (результаты показаны в строках 18–43) были закомментированы несколько строк программного кода (со 114 по 118), содержащие специализированный конструктор класса Array. В результате при выполнении программы для создания массива объектов Animal вызывается конструктор шаблона, показанный в строках 74–81.

Это приводит к созданию временных объектов Animal для каждого члена массива (строки программы 79 и 80), что отражается в строках 18–20 результатов выполнения программы.

Во всех остальных аспектах, результаты выполнения двух вариантов программы, как и следовало ожидать, идентичны.

Статические члены и шаблоны

В шаблоне можно объявлять статические переменные-члены. В результате каждый экземпляр шаблона будет иметь собственный набор статических данных. Например, если добавить статическую переменную-член в шаблон Array (например, для подсчета количества созданных массивов), то в рассмотренной выше программе будут созданы две статические переменные-члена: одна для подсчета массивов объектов типа Animal и другая для массивов целых чисел. Добавление статической переменной-члена и статической функции в шаблон Array показано в листинге 19.7.

Листинг 19.7. Использование статических переменных-членов и функций-членов с шаблонами

```
1:    #include <iostream.h>
2:
3:    const int DefaultSize = 3;
4:
5:    // Обычный класс, из объектов которого создается массив
6:      class Animal
7:      {
8:      public:
```

```
9:          // конструкторы
10:         Animal(int);
11:         Animal();
12:         ~Animal();
13:
14:         // методы доступа
15:         int GetWeight() const {  return itsWeight; }
16:         void SetWeight(int theWeight) {  itsWeight = theWeight; }
17:
18:         // дружественные операторы
19:         friend ostream& operator<< (ostream&, const Animal&);
20:
21:      private:
22:         int itsWeight;
23:      } ;
24:
25:       // оператор вывода объектов типа Animal
26:      ostream& operator<<
27:          (ostream& theStream, const Animal& theAnimal)
28:      {
29:        theStream << theAnimal.GetWeight();
30:        return theStream;
31:      }
32:
33:      Animal::Animal(int weight):
34:      itsWeight(weight)
35:      {
36:         //cout << "animal(int) ";
37:      }
38:
39:      Animal::Animal():
40:      itsWeight(0)
41:      {
42:         //cout << "animal() ";
43:      }
44:
45:      Animal::~Animal()
46:      {
47:        //cout << "Destroyed an animal...";
48:      }
49:
50:     template <class T>  // объявляем шаблон и параметр
51:     class Array          // параметризованный класс
52:     {
53:     public:
54:        // конструкторы
55:        Array(int itsSize = DefaultSize);
56:        Array(const Array &rhs);
57:       ~Array() {  delete [] pType;   itsNumberArrays--; }
58:
59:       // операторы
```

```
60:       Array& operator=(const Array&);
61:       T& operator[](int offSet) {  return pType[offSet]; }
62:       const T& operator[](int offSet) const
63:          {  return pType[offSet]; }
64:       // аксессоры
65:       int GetSize() const {  return itsSize; }
66:       static int GetNumberArrays() {  return itsNumberArrays; }
67:
68:       // функция-друг
69:      friend ostream& operator<< (ostream&, const Array<T>&);
70:
71:    private:
72:       T *pType;
73:       int  itsSize;
74:       static int itsNumberArrays;
75:    } ;
76:
77:    template <class T>
78:       int Array<T>::itsNumberArrays = 0;
79:
80:    template <class T>
81:    Array<T>::Array(int size = DefaultSize):
82:    itsSize(size)
83:    {
84:       pType = new T[size];
85:       for (int i = 0; i<size; i++)
86:         pType[i] = (T)0;
87:       itsNumberArrays++;
88:    }
89:
90:    template <class T>
91:    Array<T>& Array<T>::operator=(const Array &rhs)
92:    {
93:       if (this == &rhs)
94:          return *this;
95:       delete [] pType;
96:       itsSize = rhs.GetSize();
97:       pType = new T[itsSize];
98:       for (int i = 0; i<itsSize; i++)
99:          pType[i] = rhs[i];
100:   }
101:
102:   template <class T>
103:   Array<T>::Array(const Array &rhs)
104:   {
105:      itsSize = rhs.GetSize();
106:      pType = new T[itsSize];
107:      for (int i = 0; i<itsSize; i++)
108:         pType[i] = rhs[i];
109:      itsNumberArrays++;
110:   }
```

```
111:
112:
113:    template <class T>
114:    ostream& operator<< (ostream& output, const Array<T>& theArray)
115:    {
116:       for (int i = 0; i<theArray.GetSize(); i++)
117:          output << "[" << i << "] " << theArray[i] << endl;
118:       return output;
119:    }
120:
121:
122:
123:    int main()
124:    {
125:
126:       cout << Array<int>::GetNumberArrays() << " integer arrays\ n";
127:       cout << Array<Animal>::GetNumberArrays();
128:       cout << " animal arrays\ n\ n";
129:       Array<int> intArray;
130:       Array<Animal> animalArray;
131:
132:       cout << intArray.GetNumberArrays() << " integer arrays\ n";
133:       cout << animalArray.GetNumberArrays();
134:       cout << " animal arrays\ n\ n";
135:
136:       Array<int> *pIntArray = new Array<int>;
137:
138:       cout << Array<int>::GetNumberArrays() << " integer arrays\ n";
139:       cout << Array<Animal>::GetNumberArrays();
140:       cout << " animal arrays\ n\ n";
141:
142:       delete pIntArray;
143:
144:      cout << Array<int>::GetNumberArrays() << " integer arrays\ n";
145:      cout << Array<Animal>::GetNumberArrays();
146:      cout << " animal arrays\ n\ n";
147:     return 0;
148:   }
```

РЕЗУЛЬТАТ

```
0 integer arrays
0 animal arrays

1 integer arrays
1 animal arrays

2 integer arrays
1 animal arrays

1 integer arrays
1 animal arrays
```

АНАЛИЗ Для экономии места в листинге опущено объявление класса `Animal`. В класс `Array` добавлена статическая переменная `itsNumberArrays` (в строке 74), а поскольку эта переменная объявляется в разделе закрытых членов, в строке 66 добавлен открытый статический метод доступа `GetNumberArrays()`.

Инициализация статической переменной-члена выполняется явно в строках 77 и 78. Конструкторы и деструктор класса `Array` изменены таким образом, чтобы могли отслеживать число массивов, существующих в любой момент времени.

Доступ к статической переменной, заданной в шаблоне, можно получить так же, как и при работе со статическими переменными-членами обычного класса: с помощью метода доступа, вызванного для объекта класса, как показано в строках 132 и 133, или явным обращением к переменной класса, как показано в строках 126 и 127. Обратите внимание, что при обращении к статической переменной-члену необходимо указать тип массива, так как для каждого типа будет создана своя статическая переменная-член.

Рекомендуется

Используйте статические члены в шаблонах.
Специализируйте выполнение шаблона путем замещения функций шаблона для разных типов.

Рекомендуется

Указывайте параметр типа при вызове статических функций шаблона, чтобы получить доступ к функции требуемого типа.

Стандартная библиотека шаблонов

Отличительной чертой новой версии языка C++ является принятие стандартной библиотеки шаблонов (Standard Template Library — STL). Все основные разработчики компиляторов теперь предлагают библиотеку STL как составную часть своих программных продуктов. *STL* — это библиотека классов контейнеров, базирующихся на шаблонах. Она включает векторы, списки, очереди и стеки, а также ряд таких общих алгоритмов, как сортировка и поиск.

Цель включения библиотеки STL состоит в том, чтобы избавить вас от очередного изобретения колеса и при разработке выполнить за вас рутинные общепринятые процессы. Библиотека STL оттестирована и отлажена, отличается высокой эффективностью и не требует дополнительных затрат. Важнее всего то, что библиотеку STL можно использовать многократно для разработки собственных приложений. Необходимо только один раз разобраться в принципах использования библиотеки STL и классов-контейнеров.

Контейнеры

Контейнер — это объект, который содержит другие объекты. Стандартная библиотека C++ предоставляет ряд классов-контейнеров, являющихся мощными инструментальными средствами, которые помогают разработчикам C++ решать наиболее общие задачи программирования. Среди классов контейнеров стандартной библиотеки шаблонов (STL) различаются два типа: последовательные и ассоциативные. *Последовательные* контейнеры предназначены для обеспечения последовательного или произ-

вольного доступа к своим членам, или *элементам*. *Ассоциативные* контейнеры оптимизированы таким образом, чтобы получать доступ к своим элементам по ключевым значениям. Подобно другим компонентам стандартной библиотеки C++, библиотека STL совместима с различными операционными системами. Все классы-контейнеры библиотеки STL определены в пространстве имен `std`.

Последовательные контейнеры

Такие контейнеры стандартной библиотеки шаблонов обеспечивают эффективный последовательный доступ к списку объектов. Стандартная библиотека C++ предоставляет три вида последовательных контейнеров: векторы, списки и двухсторонние очереди.

Вектор

Массивы часто используются для хранения ряда элементов и обеспечивают возможность прямого доступа к ним. Элементы в массиве имеют один и тот же тип, а обратиться к ним можно с помощью индекса. Библиотека STL обеспечивает класс-контейнер `vector`, который ведет себя подобно массиву, но его использование отличается большей мощностью и безопасностью по сравнению со стандартным массивом C++.

Вектор — это контейнер, оптимизированный таким образом, чтобы обеспечить быстрый доступ к его элементам по индексу. Класс-контейнер `vector` определен в файле заголовка `<vector>` в пространстве имен `std` (подробнее об использовании пространств имен см. главу 17). Вектор можно наращивать по мере необходимости. Предположим, был создан вектор для 10 элементов. После того как в вектор поместили 10 объектов, он оказался целиком заполненным. Если затем к вектору добавить еще один объект, он автоматически увеличит свою вместимость так, что сможет разместить одиннадцатый объект. Вот как выглядит определение класса `vector`:

```
template <class T, class A = allocator<T>> class vector
{
    // члены класса
} ;
```

Первый аргумент (`class T`) означает тип элементов в векторе. Второй аргумент (`class A`) — это *класс распределения*, который берет на себя функции диспетчера памяти, ответственного за распределение и освобождение памяти для элементов контейнеров. Принципы построения и выполнения классов распределения затрагивают более сложные темы, которые выходят за рамки этой книги.

По умолчанию элементы создаются с помощью оператора `new()` и освобождаются с помощью оператора `delete()`, т.е. для создания нового элемента вызывается стандартный конструктор класса `T`. Это служит еще одним аргументом в пользу явного определения стандартного конструктора для ваших собственных классов. Если этого не сделать, то нельзя будет использовать стандартный векторный контейнер для хранения объектов пользовательского класса.

Определить векторы для содержания целых и вещественных чисел можно следующим образом:

```
vector<int>     vInts;          // вектор для хранения целых элементов
vector<float>   vFloats;        // вектор для хранения вещественных элементов
```

Обычно пользователь имеет представление о том, сколько элементов будет содержаться в векторе. Предположим, на курс прикладной математики в институте набирается не более 50 студентов. Прежде чем создавать вектор для массива студентов, следует побеспокоиться о том, чтобы он был достаточно большим и мог содержать 50 элементов. Стандартный класс `vector` предоставляет конструктор, который принимает число элементов в качестве параметра. Так что можно определить вектор для 50 студентов следующим образом:

```
vector<Student> MathClass(50);
```

Компилятор автоматически выделит достаточный объем памяти для хранения записей о 50 студентах. Каждый элемент вектора создается с использованием стандартного конструктора `Student::Student()`.

Количество элементов в векторе можно узнать с помощью функции-члена `size()`. В данном примере функция-член `vStudent.size()` возвратит значение `50`.

Другая функция-член, `capacity()`, сообщает, сколько в точности элементов может принять вектор, прежде чем потребуется увеличение его размера. Но об этом речь впереди.

Вектор называется *пустым*, если он не содержит ни одного элемента, т.е. если его размер равен нулю. Чтобы определить, не является ли вектор пустым, в классе вектора предусмотрена функция-член `empty()`, которая принимает значение, равное истине, если вектор пустой.

Чтобы записать студента Гарри на курс прикладной математики, т.е. (говоря языком программирования) чтобы назначить объект `Harry` класса `Student` вектору `MathClass`, можно использовать оператор индексирования (`[]`):

```
MathClass[5] = Harry;
```

Индексы начинаются с нуля. Для назначения объекта `Harry` шестым элементом вектора `MathClass` здесь используется перегруженный оператор присваивания класса `Student`. Аналогично, чтобы определить возраст объекта `Harry`, можно получить доступ к соответствующей записи, используя следующее выражение:

```
MathClass[5].GetAge();
```

Как упоминалось выше, при добавлении в вектор большего числа элементов, чем было указано при создании вектора, дополнительное место для нового элемента будет добавлено автоматически. Предположим, курс прикладной математики стал таким популярным, что количество принятых студентов превысило число 50. Возможно, за 51-го студента кто-то замолвил словечко, и декану не осталось ничего другого, как увеличить число студентов на курсе. Так вот, если на курс (в вектор `MathClass`) захочет записаться 51-я студентка Салли (объект `Sally`), компилятор спокойно расширит пределы вектора, чтобы "впустить" новое молодое дарование.

Добавлять элемент в вектор можно различными способами. Один из них — с помощью функции-члена `push_back()`:

```
MathClass.push_back(Sally);
```

Эта функция-член добавляет новый объект `Sally` класса `Student` в конец вектора `MathClass`. И теперь в векторе `MathClass` содержится уже 51 элемент, причем к объекту `Sally` можно обратиться по индексу `MathClass[50]`.

Чтобы функция `push_back()` была работоспособной, в классе `Student` нужно определить конструктор-копировщик. В противном случае эта функция не сможет создать копию объекта `Sally`.

В векторе из библиотеки STL не задается максимальное число элементов, так как это решение лучше переложить на плечи создателей компиляторов. Векторный класс предоставляет функцию-член `max_size()`, которая способна сообщить это магическое число, определенное в вашем компиляторе.

В листинге 19.8 демонстрируется использование векторного класса. Вы увидите, что для упрощения обработки строк в этом листинге используется стандартный класс `string`. Для получения более подробной информации о классе `string` обратитесь к документации, прилагаемой к вашему компилятору.

Листинг 19.8. Создание вектора и обеспечение доступа к его элементам

```
1:      #include <iostream>
2:      #include <string>
3:      #include <vector>
4:      using namespace std;
5:
6:      class Student
7:      {
8:      public:
9:            Student();
10:           Student(const string& name, const int age);
11:           Student(const Student& rhs);
12:           ~Student();
13:
14:           void    SetName(const string& name);
15:           string  GetName()      const;
16:           void    SetAge(const int age);
17:           int           GetAge()       const;
18:
19:           Student& operator=(const Student& rhs);
20:
21:     private:
22:           string itsName;
23:           int itsAge;
24:     } ;
25:
26:     Student::Student()
27:     : itsName("New Student"), itsAge(16)
28:     { }
29:
30:     Student::Student(const string& name, const int age)
31:     : itsName(name), itsAge(age)
32:     { }
33:
34:     Student::Student(const Student& rhs)
35:     : itsName(rhs.GetName()), itsAge(rhs.GetAge())
36:     { }
37:
38:     Student::~Student()
39:     { }
40:
```

```
41:    void Student::SetName(const string& name)
42:    {
43:            itsName = name;
44:    }
45:
46:    string Student::GetName() const
47:    {
48:            return itsName;
49:    }
50:
51:    void Student::SetAge(const int age)
52:    {
53:            itsAge = age;
54:    }
55:
56:    int Student::GetAge() const
57:    {
58:            return itsAge;
59:    }
60:
61:    Student& Student::operator=(const Student& rhs)
62:    {
63:            itsName = rhs.GetName();
64:            itsAge = rhs.GetAge();
65:            return *this;
66:    }
67:
68:    ostream& operator<<(ostream& os, const Student& rhs)
69:    {
70:        os << rhs.GetName() << " is " << rhs.GetAge() << " years old";
71:           return os;
72:    }
73:
74:    template<class T>
75:    void ShowVector(const vector<T>& v);   // Отображает свойства вектора
76:
77:    typedef vector<Student>        SchoolClass;
78:
79:    int main()
80:    {
81:            Student Harry;
82:            Student Sally("Sally", 15);
83:            Student Bill("Bill", 17);
84:            Student Peter("Peter", 16);
85:
86:            SchoolClass    EmptyClass;
87:            cout << "EmptyClass:\ n";
88:            ShowVector(EmptyClass);
89:
90:            SchoolClass GrowingClass(3);
```

```
91:          cout << "GrowingClass(3):\ n";
92:          ShowVector(GrowingClass);
93:
94:          GrowingClass[0] = Harry;
95:          GrowingClass[1] = Sally;
96:          GrowingClass[2] = Bill;
97:          cout << "GrowingClass(3) after assigning students:\ n";
98:          ShowVector(GrowingClass);
99:
100:         GrowingClass.push_back(Peter);
101:         cout << "GrowingClass() after added 4th student:\ n";
102:         ShowVector(GrowingClass);
103:
104:         GrowingClass[0].SetName("Harry");
105:         GrowingClass[0].SetAge(18);
106:         cout << "GrowingClass() after Set\n:";
107:         ShowVector(GrowingClass);
108:
109:         return 0;
110:   }
111:
112:   //
113:   // Отображает свойства вектора
114:   //
115:   template<class T>
116:   void ShowVector(const vector<T>& v)
117:   {
118:         cout << "max_size() = " << v.max_size();
119:         cout << "\ tsize() = " << v.size();
120:         cout << "\ tcapacity() = " << v.capacity();
121:         cout << "\ t" << (v.empty()? "empty": "not empty");
122:         cout << "\ n";
123:
124:         for (int i = 0; i < v.size(); ++i)
125:               cout << v[i] << "\ n";
126:
127:         cout << endl;
128:   }
129:
```

РЕЗУЛЬТАТ

```
EmptyClass:
max_size() = 214748364  size() = 0      capacity() = 0  empty

GrowingClass(3):
max_size() = 214748364  size() = 3      capacity() = 3  not empty
New Student is 16 years old
New Student is 16 years old
New Student is 16 years old
```

РЕЗУЛЬТАТ

```
GrowingClass(3) after assigning students:
max_size() = 214748364  size() = 3      capacity() = 3  not empty
New Student is 16 years old
Sally is 15 years old
Bill is 17 years old

GrowingClass() after added 4th student:
max_size() = 214748364  size() = 4      capacity() = 6  not empty
New Student is 16 years old
Sally is 15 years old
Bill is 17 years old
Peter is 16 years old
GrowingClass() after Set:
max_size() = 214748364 size() = 4      capacity() = 6  not empty
Harry is 18 years old
Sally is 15 years old
Bill is 17 years old
Peter is 16 years old
```

АНАЛИЗ

Определение класса `Student` занимает строки 6–24, а выполнение его функций-членов показано в строках 26–66. Структура этого класса проста и дружественна по отношению к классу `vector`. По рассмотренным ранее причинам были определены стандартный конструктор, конструктор-копировщик и перегруженный оператор присваивания. Обратите внимание, что переменная-член `itsName` определена как экземпляр базового строкового класса C++ `string`. Как видите, со строками в C++ намного проще работать, подобное было в языке C (с типом `char*`).

Функция шаблона `ShowVector()` объявлена в строках 74–75 и определена в строках 115–128. Она используется для вызова функций-членов вектора, отображающих его свойства: `max_size()`, `size()`, `capacity()` и `empty()`. Насколько можно судить по результатам работы этой программы, максимальное число объектов класса `Student`, которое может принять этот вектор, в Visual C++ составляет 214 748 364. Для других типов элементов это число может быть другим. Например, вектор целых чисел может вместить до 1 073 741 823 элементов. Если же вы используете другие компиляторы, то максимальное число элементов у вас может отличаться от приведенных здесь значений.

В строках 124 и 125 выполняется цикл, опрашивающий все элементы вектора и отображающий их значения, используя оператор вывода (`<<`), который перегружен в строках 68–72.

В строках 81–84 создаются четыре объекта класса `Student`. В строке 86 с помощью стандартного конструктора векторного класса определяется пустой вектор с именем `EmptyClass`. Когда вектор создается таким способом, то компилятор для него совсем не выделяет места в памяти. Как видно по результатам работы функции `ShowVector(EmptyClass)`, как размер, так и вместимость этого вектора равны нулю.

Строка 90 содержит определение вектора для включения трех объектов класса `Student`. Размер и вместимость этого вектора, как и ожидалось, равны трем. В строках 94–96 с помощью оператора индексирования (`[]`) элементы вектора `GrowingClass` заполняются объектами класса `Student`.

В строке 100 к вектору добавляется четвертый студент (`Peter`). Это увеличивает размер вектора до четырех элементов. Интересно, что его вместимость теперь установлена равной шести. Это означает, что компилятор автоматически выделил доста-

точно пространства, которого хватит даже для шести объектов класса Student. Поскольку векторам должен быть выделен непрерывный блок памяти, для их расширения требуется выполнить целый ряд операций. Сначала выделяется новый блок памяти, достаточно большой для всех четырех объектов класса Student. Затем в только что выделенную память копируются эти три элемента, а четвертый добавляется после третьего элемента. И наконец, исходный блок памяти возвращается в область динамического обмена. При большом количестве элементов в векторе процесс перераспределения и освобождения памяти может оказаться весьма длительным. Поэтому в целях сокращения вероятности выполнения таких дорогих (по времени) операций компилятор использует стратегию оптимизации. В данном примере, если сразу добавить к вектору еще один или два объекта, отпадает необходимость в дополнительных операциях, связанных с освобождением и перераспределением памяти.

В строках 104 и 105 вновь используется оператор индексирования ([]), чтобы изменить переменные-члены первого объекта в векторе GrowingClass.

Рекомендуется	Рекомендуется
Определяйте стандартный конструктор для класса, если его объекты будут содержаться в векторе. **Определяйте** конструктор-копировщик для такого класса.	**Определяйте** для такого класса перегруженный оператор присваивания.

Класс-контейнер вектора имеет и другие функции-члены. Функция front() возвращает ссылку на первый элемент в списке, а функция back() — на последний. Функция at() работает подобно оператору индексирования ([]). Она более безопасна, поскольку проверяет, попадает ли переданный ей индекс в диапазон доступных элементов. Если адрес оказывается вне диапазона, эта функция генерирует исключение out_of_range. (Исключительные ситуации рассматриваются на следующем занятии.)

Функция insert() вставляет один или несколько узлов (элементов) в текущую позицию вектора. Функция Pop_back() удаляет из вектора последний элемент. Наконец, функция remove() удаляет из вектора один или несколько элементов.

Список

Список — это контейнер, разработанный для обеспечения оптимального выполнения частых вставок и удалений элементов.

Класс-контейнер библиотеки STL list определен в файле заголовка <list> в пространстве имен std. Класс list выполнен как двунаправленный связанный список, в котором каждый узел содержит указатели как на предыдущий, так и на последующий узел списка.

Класс list имеет все функции-члены, предоставляемые векторным классом. Как вы помните, список можно пройти, следуя по связям между узлами, реализованным с помощью указателей. Стандартный класс-контейнер list с той же целью использует алгоритм, называемый итератором.

Итератор — это обобщение указателя. Чтобы отыскать узел, на который указывает итератор, его нужно разыменовывать. Использование итераторов для доступа к узлам списка демонстрируется в листинге 19.9.

Листинг 19.9. Навигация по списку с помощью итератора

```
1:    #include <iostream>
2:    #include <list>
3:    using namespace std;
4:
5:    typedef list<int> IntegerList;
6:
7:    int main()
8:    {
9:          IntegerList    intList;
10:
11:         for (int i = 1; i <= 10; ++i)
12:               intList.push_back(i * 2);
13:
14:         for (IntegerList::const_iterator ci = intList.begin();
15:                                          ci != intList.end(); ++ci)
16:               cout << *ci << " ";
17:
18:         return 0;
19:   }
```

РЕЗУЛЬТАТ 2 4 6 8 10 12 14 16 18 20

АНАЛИЗ В строке 9 объект `intList` определен как список целых чисел. В строках 11 и 12 с помощью функции `push_back()` в список добавляются первые 10 положительных четных чисел.

В строках 14–16 мы обращаемся к каждому узлу в списке, используя константный итератор. Константность указывает, что мы не собираемся изменять узлы с помощью этого итератора. Если бы мы хотели изменить узел, на который указывает итератор, пришлось бы использовать переменный итератор:

```
intList::iterator
```

Функция-член `begin()` возвращает итератор на первый узел списка. Оператор инкремента (`++`) можно использовать для перехода к итератору следующего узла. Функция-член `end()`, что может показаться странным, возвращает итератор на узел, расположенный за последним узлом списка. Часто метод `end()` используют для определения допустимых границ списка.

Разыменование итератора (для возвращения связанного с ним узла) происходит аналогично разыменованию указателя, как показано в строке 16.

Хотя понятие итератора было введено только при рассмотрении класса `list`, итераторы можно использовать и с векторными классами. В дополнение к функциям-членам, с которыми вы познакомились в векторном классе, базовый класс списка тоже представляет функции `push_front()` и `pop_front()`, которые работают точно так же, как и функции `push_back()` и `pop_back()`. Но вместо добавления и удаления элементов в конце списка, они добавляют и удаляют элементы в его начале.

Контейнер двухсторонней очереди

Двухсторонняя очередь подобна двунаправленному вектору — она наследует эффективность класса-контейнера `vector` по операциям последовательного чтения и записи. Но, кроме того, класс контейнер `deque` обеспечивает оптимизированное добавление и удаление узлов с обоих концов очереди. Эти операции реализованы аналогично классу-контейнеру `list`, где процесс выделения памяти запускается только для новых элементов. Эта особенность класса двухсторонней очереди устраняет потребность перераспределения целого контейнера в новую область памяти, как это приходится делать в векторном классе. Поэтому двухсторонние очереди идеально подходят для приложений, в которых вставки и удаления происходят с двух концов массива и для которых имеет важное значение последовательный доступ к элементам. Примером такого приложения может служить имитатор сборки поезда, в котором вагоны могут присоединяться к поезду с обоих концов.

Стеки

Одной из самых распространенных в программировании структур данных является стек. Однако стек не используется как независимый контейнерный класс, скорее, его можно назвать оболочкой контейнера. Шаблонный класс `stack` определен в файле заголовка `<stack>` в пространстве имен `std`.

Стек — это непрерывный выделенный блок памяти, который может расширяться или сжиматься в хвостовой части, т.е. к элементам стека можно обращаться или удалять только с одного конца. Вы уже видели подобные характеристики в последовательных контейнерах, особенно в классах `vector` и `deque`. Фактически для реализации стека можно использовать любой последовательный контейнер, который поддерживает функции `back()`, `push_back()` и `pop_back()`. Большинство других методов контейнеров для работы стека не используются, поэтому они и не предоставляются классом `stack`.

Базовый шаблонный класс `stack` библиотеки STL шаблона разработан для поддержания объектов любого типа. Единственное ограничение состоит в том, что все элементы должны иметь один и тот же тип.

Данные в стеке организованы по принципу "последним вошел — первым вышел". Ее можно сравнить с переполненным лифтом: первый человек, вошедший в лифт, припирается к стене, а последний втиснувшийся стоит прямо у двери. Когда лифт поднимается на указанный кем-то из пассажиров этаж, тот, кто зашел последним, должен выйти первым. Если кто-нибудь (из стоящих посередине пассажиров) захочет выйти из лифта раньше других, то все, кто находится между ним и дверью, должны выйти из лифта, выпустив его, а затем вернуться обратно.

Открытый конец стека называется *вершиной стека*, а действия, выполняемые с элементами стека, — операциями *помещения* (push) и *выталкивания* (pop) из стека. Для класса `stack` эти общепринятые термины остаются в силе.

> Класс `stack` из библиотеки STL не соответствует стекам памяти, используемым компиляторами и операционными системами, которые могут содержать объекты различных типов, хотя они работают сходным образом.

Очередь

Очередь — это еще одна распространенная в программировании структура данных. В этом случае элементы добавляются к очереди с одного конца, а вынимаются с другого. Приведем классическую аналогию. Вспомним стек. Его можно сравнить со стопкой тарелок на столе. При добавлении в стек тарелка ставится сверху всей стопки (помещение в стек), и взять тарелку из стопки (стека) можно тоже только сверху (выталкивание из стека), т.е. берется тарелка, которая была положена в стопку самой последней.

Очередь же можно сравнить с любой очередью людей, например при входе в театр. Вы занимаете очередь сзади, а покидаете ее спереди. Конечно, каждому из нас приходилось стоять предпоследним в какой-нибудь очереди (например, в магазине), когда вдруг начинает работать еще одна касса, к которой подбегает стоявший за вами, что скорее напоминает стек, чем очередь. Но в компьютерах такого не случается.

Подобно классу `stack`, класс `queue` реализован как класс оболочки контейнера. Контейнер должен поддерживать такие функции, как `front()`, `back()`, `push_back()` и `pop_front()`.

Ассоциативные контейнеры

Тогда как последовательные контейнеры предназначены для последовательного и произвольного доступа к элементам с помощью индексов или итераторов, ассоциативные контейнеры разработаны для быстрого произвольного доступа к элементам с помощью ключей. Стандартная библиотека C++ предоставляет четыре ассоциативных контейнера: карту, мультикарту, множество и мультимножество.

Карта

Вектор можно сравнить с расширенной версией массива. Он имеет все характеристики массива и ряд дополнительных возможностей. К сожалению, вектор также страдает от одного из существенных недостатков массивов: он не предоставляет возможности для произвольного доступа к элементам с помощью ключа, а лишь использует для этого индекс или итератор. Ассоциативные контейнеры как раз обеспечивают быстрый произвольный доступ, основанный на ключевых значениях.

В листинге 19.10 для создания списка студентов, который мы рассматривали в листинге 19.8, используется карта.

Листинг 19.10. Класс-контейнер `map`

```
1:      #include <iostream>
2:      #include <string>
3:      #include <map>
4:      using namespace std;
5:
6:      class Student
7:      {
8:      public:
9:            Student();
10:           Student(const string& name, const int age);
11:           Student(const Student& rhs);
```

```
12:          ~Student();
13:
14:          void    SetName(const string& name);
15:          string  GetName()      const;
16:          void    SetAge(const int age);
17:          int             GetAge()       const;
18:
19:          Student& operator=(const Student& rhs);
20:
21:     private:
22:          string itsName;
23:          int itsAge;
24:    } ;
25:
26:     Student::Student()
27:     : itsName("New Student"), itsAge(16)
28:     { }
29:
30:     Student::Student(const string& name, const int age)
31:     : itsName(name), itsAge(age)
32:     { }
33:
34:     Student::Student(const Student& rhs)
35:     : itsName(rhs.GetName()), itsAge(rhs.GetAge())
36:     { }
37:
38:     Student::~Student()
39:     { }
40:
41:     void Student::SetName(const string& name)
42:     {
43:          itsName = name;
44:     }
45:
46:     string Student::GetName() const
47:     {
48:          return itsName;
49:     }
50:
51:     void Student::SetAge(const int age)
52:     {
53:          itsAge = age;
54:     }
55:
56:     int Student::GetAge() const
57:     {
58:          return itsAge;
59:     }
60:
61:     Student& Student::operator=(const Student& rhs)
62:     {
```

```
63:         itsName = rhs.GetName();
64:         itsAge = rhs.GetAge();
65:         return *this;
66:     }
67:
68:     ostream& operator<<(ostream& os, const Student& rhs)
69:     {
70:         os << rhs.GetName() << " is " << rhs.GetAge() << " years old";
71:         return os;
72:     }
73:
74:     template<class T, class A>
75:     void ShowMap(const map<T, A>& v);     // отображает свойства карты
76:
77:     typedef map<string, Student> SchoolClass;
78:
79:     int main()
80:     {
81:         Student Harry("Harry", 18);
82:         Student Sally("Sally", 15);
83:         Student Bill("Bill", 17);
84:         Student Peter("Peter", 16);
85:
86:         SchoolClass MathClass;
87:         MathClass[Harry.GetName()] = Harry;
88:         MathClass[Sally.GetName()] = Sally;
89:         MathClass[Bill.GetName()] = Bill;
90:         MathClass[Peter.GetName()] = Peter;
91:
92:         cout << "MathClass:\ n";
93:         ShowMap(MathClass);
94:
95:         cout << "We know that " << MathClass["Bill"].GetName()
96:              << " is " << MathClass["Bill"].GetAge() << "years old\ n";
97:
98:         return 0;
99:     }
100:
101:    //
102:    // Отображает свойства карты
103:    //
104:    template<class T, class A>
105:    void ShowMap(const map<T, A>& v)
106:    {
107:        for (map<T, A>::const_iterator ci = v.begin();
108:                                       ci != v.end(); ++ci)
109:            cout << ci->first << ": " << ci->second << "\ n";
110:
111:        cout << endl;
112:    }
```

РЕЗУЛЬТАТ

```
MathClass:
Bill: Bill is 17 years old
Harry: Harry is 18 years old
Peter: Peter is 16 years old
Sally: Sally is 15 years old

We know that Bill is 17 years old
```

АНАЛИЗ

В строке 3 в программу добавляется файл заголовка `<map>`, поскольку будет использоваться стандартный класс-контейнер `map`. Для отображения элементов карты определяется шаблонная функция `ShowMap`. В строке 77 класс `SchoolClass` определяется как карта элементов, каждый из которых состоит из пары (ключ, значение). Первая составляющая пары — это значение ключа. В нашем классе `SchoolClass` имена студентов используются в качестве ключевых значений, которые имеют тип `string`. Ключевое значение элемента в контейнере карты должно быть уникальным, т.е. никакие два элемента не могут иметь одно и то же ключевое значение. Вторая составляющая пары — фактический объект, в данном примере это объект класса `Student`. Парный тип данных реализован в библиотеке STL как структура (тип данных `struct`), состоящая из двух членов, а именно: `first` и `second`. Эти члены можно использовать для получения доступа к ключу и значению узла.

Пропустим пока функцию `main()` и рассмотрим функцию `ShowMap`, которая открывает доступ к объектам карты с помощью константного итератора. Выражение `ci->first` (строка 109) указывает на ключ (имя студента), а выражение `ci->second` — на объект класса `Student`.

В строках 81—84 создаются четыре объекта класса `Student`. Класс `MathClass` определяется как экземпляр класса `SchoolClass` (строка 86), а в строках 87—90 уже имеющиеся четыре студента добавляются в класс `MathClass`:

```
map_object[key_value] = object_value;
```

Для добавления в карту пары ключ—значение можно было бы также использовать функции `push_back()` или `insert()` (за более подробной информацией обратитесь к документации, прилагаемой к вашему компилятору).

После добавления к карте всех объектов класса `Student` можно обращаться к любому из них, используя их ключевые значения. В строках 95 и 96 для считывания записи, относящейся к студенту Биллу (объекту `Bill`), используется выражение `MathClass["Bill"]`.

Другие ассоциативные контейнеры

Класс-контейнер мультикарты — это класс карты, не ограниченный уникальностью ключей. Это значит, что одно и то же ключевое значение могут иметь не один, а несколько элементов.

Класс-контейнер множества также подобен классу карты. Единственное отличие в том, что его элементы представляют собой не пары ключ—значение, а только ключи.

Наконец, класс-контейнер мультимножества — это класс множества, который позволяет иметь несколько ключевых значений.

Классы алгоритмов

Контейнер — это удобное место для хранения последовательности элементов. Все стандартные контейнеры содержат методы управления контейнерами и их элементами. Однако манипулирование собственными данными в программах с помощью этих методов может потребовать от программиста написания обширного программного кода, что чревато появлением ошибок. Но поскольку большинство операций, выполняемых над данными, рутинны и повторяются от программы к программе, то подборка универсальных алгоритмов может существенно облегчить написание программ обработки данных контейнера. Стандартная библиотека предоставляет около 60 стандартных алгоритмов, которые выполняют большинство базовых и часто используемых операций, характерных для контейнеров.

Стандартные алгоритмы определены в файле `<algorithm>` в пространстве имен `std`.

Чтобы понять, как работают стандартные алгоритмы, необходимо познакомиться с понятием объектов функций. *Объект функции* — это экземпляр класса, в котором определен перегруженный оператор вызова функции(). В результате этот класс может вызываться как функция. Использование объекта функции показано в листинге 19.11.

Листинг 19.11. Объект функции

```
1:    #include <iostream>
2:    using namespace std;
3:
4:    template<class T>
5:    class Print    {
6:    public:
7:        void operator()(const T& t)
8:        {
9:                    cout << t << " ";
10:           }
11:   } ;
12:
13:   int main()
14:   {
15:       Print<int> DoPrint;
16:       for (int i = 0; i < 5; ++i)
17:           DoPrint(i);
18:       return 0;
19:   }
```

РЕЗУЛЬТАТ

```
0 1 2 3 4
```

АНАЛИЗ

В строках 4—11 определяется шаблонный класс `Print`. Перегруженный в строках 7—10 оператор вызова функции ()принимает объект и перенаправляет его в стандартный поток вывода. В строке 15 определяется объект `DoPrint` как экземпляр класса `Print`. После этого, чтобы вывести на печать любые целочисленные значения, объект `DoPrint` можно использовать подобно обычной функции, как показано в строке 17.

Операции, не изменяющие последовательность

Операции, не изменяющие последовательность данных в структуре, реализуются с помощью таких функций, как `for_each()` и `find()`, `search()`, `count()` и т.д. В листинге 19.12 показан пример использования объекта функции и алгоритм `for_each`, предназначенный для печати элементов вектора.

Листинг 19.12. Использование алгоритма `for_each()`

```
1:    #include <iostream>
2:    #include <vector>
3:    #include <algorithm>
4:    using namespace std;
5:
6:    template<class T>
7:    class Print
8:    {
9:    public:
10:       void operator()(const T& t)
11:       {
12:           cout << t << " ";
13:       }
14:   } ;
15:
16:   int main()
17:   {
18:       Print<int>    DoPrint;
19:       vector<int>    vInt(5);
20:
21:       for (int i = 0; i < 5; ++i)
22:           vInt[i] = i * 3;
23:
24:       cout << "for_each()\ n";
25:       for_each(vInt.begin(), vInt.end(), DoPrint);
26:       cout << "\ n";
27:
28:       return 0;
29:   }
```

РЕЗУЛЬТАТ

```
for_each()
0 3 6 9 12
```

АНАЛИЗ Обратите внимание, что все стандартные алгоритмы C++ определены в файле заголовка `<algorithm>`, поэтому следует включить его в нашу программу. Большая часть программы не должна вызывать никаких трудностей. В строке 25 вызывается функция `for_each()`, чтобы опросить каждый элемент в векторе `vInt`. Для каждого элемента она вызывает объект функции `DoPrint` и передает этот элемент оператору `DoPrint.operator()`, что приводит к выводу на экран значения данного элемента.

Алгоритмы изменения последовательности

Под изменением последовательности понимают изменение порядка элементов в структуре данных. Изменять последовательность способны операции, связанные с заполнением или переупорядочением коллекций. Алгоритм заполнения показан в листинге 19.13.

Листинг 19.13. Алгоритм изменения последовательности

```
1:    #include <iostream>
2:    #include <vector>
3:    #include <algorithm>
4:    using namespace std;
5:
6:    template<class T>
7:    class Print
8:    {
9:    public:
10:        void operator()(const T& t)
11:        {
12:            cout << t << " ";
13:        }
14:    } ;
15:
16:    int main()
17:    {
18:        Print<int>    DoPrint;
19:        vector<int>    vInt(10);
20:
21:        fill(vInt.begin(), vInt.begin() + 5, 1);
22:        fill(vInt.begin() + 5, vInt.end(), 2);
23:
24:        for_each(vInt.begin(), vInt.end(), DoPrint);
25:        cout << "\ n\ n";
26:
27:        return 0;
28:    }
```

РЕЗУЛЬТАТ

```
1 1 1 1 1 2 2 2 2 2
```

АНАЛИЗ Единственная новая деталь в этом листинге содержится в строках 21 и 22, где используется алгоритм `fill()`. Алгоритм заполнения предназначен для заполнения элементов последовательности заданным значением. В строке 21 целое значение 1 присваивается первым пяти элементам в векторе `vInt`. А последним пяти элементам вектора `vInt` присваивается целое число 2 (в строке 22).

Резюме

Сегодня вы узнали, как создавать и использовать шаблоны — встроенное средство языка C++, используемое для создания параметризованных типов, т.е. типов, которые изменяют свое выполнение в зависимости от параметров, переданных при создании класса. Таким образом, шаблоны — это возможность многократного использования программного кода, причем безопасным и эффективным способом.

В определении шаблона устанавливается параметризованный тип. Каждый экземпляр шаблона — это реальный объект, который можно использовать подобно любому другому объекту: в качестве параметра функции, возвращаемого значения и т.д.

Классы шаблонов могут объявлять три типа функций-друзей: не относящихся к шаблону, шаблонных и специализированных по типу. В шаблоне можно объявлять статические члены. Тогда каждый экземпляр шаблона будет иметь собственный набор статических данных.

Если нужно специализировать выполнение функции шаблона в зависимости от типа, то ее можно замещать для разных типов.

Вопросы и ответы

Чем использование шаблонов лучше использования макросов?

Шаблоны обеспечивают более безопасное использование разных типов и встроены в язык.

Какова разница между параметризованным типом функции шаблона и параметрами обычной функции?

Обычная функция (не шаблонная) принимает параметры, с которыми может выполнять заданные действия. Функция шаблона позволяет с помощью параметра шаблона устанавливать тип передаваемого параметра функции. Так, в функцию можно передать массив объектов, тип которых будет уникален для разных экземпляров шаблона.

Когда следует использовать шаблоны, а когда наследование?

Используйте шаблоны в том случае, когда все или почти все выполнение класса остается неизменным, а изменяется только тип данных, используемых в классе.

Когда использовать дружественные шаблонные классы и функции?

Когда каждый экземпляр, независимо от типа, должен быть другом по отношению к этому классу или функции.

Когда использовать дружественные шаблонные классы или функции, специализированные по типу?

Когда между двумя классами нужно установить отношения по типу *один-к-одному*. Например, массив `array<int>` должен соответствовать итератору `iterator<int>`, но не `iterator<Animal>`.

Каковы два типа стандартных контейнеров?

Последовательные и ассоциативные контейнеры. Последовательные контейнеры обеспечивают оптимизированный последовательный и произвольный доступ к своим элементам. Ассоциативные контейнеры обеспечивают оптимизированный доступ к элементам на основе ключевых значений.

Какими атрибутами должен обладать класс, чтобы его можно было использовать со стандартными контейнерами?

В классе должны быть явно определены стандартный конструктор, конструктор-копировщик и перегруженный оператор присваивания.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний, а также ряд упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Какова разница между шаблоном и макросом?
2. В чем состоит отличие параметра шаблона от параметра функции?
3. Чем отличается обычный дружественный шаблонный класс от дружественного шаблонного класса, специализированного по типу?
4. Можно ли обеспечить особое выполнение для определенного экземпляра шаблона?
5. Сколько статических переменных-членов будет создано, если в определение класса шаблона поместить один статический член?
6. Что представляют собой итераторы?
7. Что такое объект функции?

Упражнения

1. Создайте шаблон на основе данного класса `List`:

```
class List
{
private:

public:
     List():head(0),tail(0),theCount(0) { }
     virtual ~List();
     void insert( int value );
     void append( int value );
     int is_present( int value ) const;
     int is_empty() const {  return head == 0; }
     int count() const {  return theCount; }
private:
     class ListCell
     {
     public:
          ListCell(int value, ListCell *cell =
):val(value),next(cell){ }
```

```
        int val;
        ListCell *next;
    } ;
    ListCell *head;
    ListCell *tail;
    int theCount;
} ;
```

2. Напишите выполнение обычной (не шаблонной) версии класса `List`.
3. Напишите шаблонный вариант выполнения.
4. Объявите три списка объектов: типа `Strings`, типа `Cat` и типа `int`.
5. **Жучки**: что неправильно в приведенном ниже программном коде? (Предположите, что определяется шаблон класса `List`, а `Cat` — это класс, определенный на одном из предыдущих занятий.)

```
List<Cat> Cat_List;
Cat Felix;
CatList.append( Felix );
cout << "Felix is " <<
     ( Cat_List.is_present( Felix ) ) ? "" : "not " << "present\ n";
```

6. ПОДСКАЗКА (поскольку задание не из самых легких): подумайте, чем тип `Cat` *отличается от типа* `int`?
7. Объявите дружественный оператор `operator==` для класса `List`.
8. Напишите выполнение дружественного оператора `operator==` для класса `List`.
9. Грешит ли оператор `operator==` той же проблемой, которая существует в упражнении 5?
10. Напишите выполнение функции шаблона, осуществляющей операцию обмена данными, в результате чего две переменные должны обменяться содержимым.
11. Напишите выполнение класса `SchoolClass`, показанного в листинге 19.8, как списка. Для добавления в список четырех студентов используйте функцию `push_back()`. Затем пройдитесь по полученному списку и увеличьте возраст каждого студента на один год.
12. Измените код из упражнения 10 таким образом, чтобы для отображения данных о каждом студенте использовался объект функции.

Отслеживание исключительных ситуаций и ошибок

Программный код, представленный в этой книге, был создан в иллюстративных целях. Мы не упоминали о возможных ошибках, чтобы не отвлекать вас от основных моментов программирования, представленных в том или ином фрагменте программы. Реальные же программы должны обязательно предусматривать возможные аварийные ситуации.

Сегодня вы узнаете:

- Что представляют собой исключительные ситуации
- Как перехватываются и обрабатываются исключения
- Что такое наследование исключений
- Как использовать исключения в общей структуре отслеживания и устранения ошибок
- Что представляет собой отладка программы

Ошибки, погрешности, ляпсусы и "гнилой" код

К сожалению, все программисты допускают ошибки. Чем больше программа, тем выше вероятность возникновения в ней ошибок, многие из которых до поры до времени остаются незамеченными и попадают в конечный программный продукт, уже выпущенный на рынок. С этой печальной истиной трудно смириться, поэтому создание надежных, свободных от ошибок программ должно быть задачей номер один для каждого программиста, серьезно относящегося к своему делу.

Одна из наиболее острых проблем в индустрии создания программ — это нестабильный программный код, нафаршированный ошибками. Обычно самые большие расходы во многих работах, связанных с программированием, приходятся на тестирование программ и исправление ошибок. Тот, кто решит проблему создания добротных, надежных и безотказных программ за короткий срок и при низких затратах, произведет революцию во всей индустрии программных продуктов.

Все ошибки в программах можно разделить на несколько групп. Первый тип ошибок вызван недостаточно проработанной логикой алгоритма выполнения программы.

Второй тип — синтаксические ошибки, т.е. использование неправильной идиомы, функции или структуры. Эти два типа ошибок самые распространенные, поэтому именно на них сосредоточено внимание программистов.

Теория и практика неопровержимо доказали, что чем позже в процессе разработки обнаруживается проблема, тем дороже стоит ее устранение. Оказывается, что проблемы или ошибки в программах дешевле всего обойдутся компании в том случае, если своевременно принять меры по предупреждению их появления. Не слишком дорого обойдутся и те ошибки, которые распознаются компилятором. Стандарты языка C++ заставляют разработчиков создавать такие компиляторы, которые способны обнаруживать как можно больше ошибок на этапе компиляции программы.

Ошибки, которые прошли этап компиляции, но были выявлены при первом же тестировании и обнаруживались регулярно, также легко устранимы, чего не скажешь о "минах замедленного действия", проявляющих себя внезапно в самый неподходящий момент.

Еще большей проблемой, чем логические или синтаксические ошибки, является недостаточная устойчивость программ, т.е. программа сносно работает в том случае, если пользователь вводит данные, которые предусматривались, но дает сбой, если по ошибке, например, вместо числа будет введена буква. Другие программы внезапно зависают из-за переполнении памяти, или при извлечении из дисковода гибкого диска, или при потере линии модемом.

Чтобы повысить устойчивость программ, программисты стремятся предупредить все варианты непредвиденных ситуаций. Устойчивой считают программу, которая может справляться во время работы с любыми неожиданностями: от получения нестандартных данных, введенных пользователем, до переполнения памяти компьютера.

Важно различать ошибки, которые возникают вследствие некорректного синтаксиса программного кода, логические ошибки, которые возникают потому, что программист неправильно истолковал проблему или неправильно решил ее, и исключительные ситуации, которые возникают из-за необычных, но предсказуемых проблем, например связанных с конфликтами ресурсов (имеется в виду недостача памяти или дискового пространства).

Исключительные ситуации

Для выявления синтаксических ошибок программисты используют встроенные средства компилятора и добавляют в программы различные ловушки ошибок, которые подробнее обсуждаются на следующем занятии. Для обнаружения логических ошибок проводится критический анализ проектных решений и всестороннее тестирование программного продукта.

Однако ни в одной программе нельзя устранить возможность возникновения исключительных ситуаций. Единственное, что может сделать программист, это подготовить программу к их возникновению. Например, невозможно средствами программирования предупредить переполнение памяти компьютера во время выполнения программы, но от программиста зависит, как поведет себя программа в этой ситуации. Можно выбрать следующие варианты ответа программы:

- привести программу к аварийному останову;
- информировать пользователя о случившемся и корректно выйти из программы;
- информировать пользователя и позволить ему сделать попытку восстановить рабочее состояние программы и продолжить работу;
- выбрать обходной путь и продолжить работу программы, не беспокоя пользователя.

Последний вариант, т.е. выбор обходного пути, безусловно, предпочтительнее аварийного останова программы. Хотя этот вариант не является необходимым или даже желательным в каждой программе, все-таки стоит написать такой код, который бы самостоятельно, автоматически, без лишнего шума справлялся со всеми исключительными ситуациями и продолжал работу.

Язык C++ предоставляет безопасные интегрированные методы отслеживания всех возможных исключительных ситуаций, возникающих во время выполнения программы.

Несколько слов о "гнилом" коде

То, что программный продукт может портиться со временем, прямо как яблоко на вашем столе, — вполне доказанный факт. Это явление возникает не по вине злых бактерий или грибков и не из-за пыли на компьютере, а потому, что практически любой код содержит скрытые внутренние ошибки, к которым добавляется нарастающее несоответствие старой программы новому компьютерному обеспечению и программному окружению. Идеально написанная и хорошо отлаженная программа очень быстро может превратиться в безделицу, больше не привлекающую внимания пользователя.

Чтобы иметь возможность быстро исправлять возникающие ошибки и модернизировать программу в соответствии с требованиями текущего дня, необходимо так писать программный код, чтобы разобраться в нем по прошествию некоторого времени могли не только вы, но и любой другой программист.

"Гнилой" код — это шутливый термин, придуманный программистами для объяснения того, как хорошо отлаженные программы вдруг становятся ненадежными и неэффективными. Об этом явлении не стоит забывать, ведь программы часто бывают чрезвычайно сложными, из-за чего многие ошибки, погрешности и ляпсусы могут долгое время оставаться в тени, пока не проявят себя во всей красе. Для защиты от подобной "плесени" нужно писать код таким образом, чтобы самим было несложно поддерживать его работоспособность.

Это означает, что ваш код должен быть подробно прокомментирован, даже если вы и не предполагаете, что кто-то другой, кроме вас, может заглянуть в него. Когда пройдет месяцев шесть после того, как вы передадите свой код заказчику, вы сами будете смотреть на свою программу глазами постороннего человека и удивляться тому, как можно было написать такой непонятный и извилистый код, надеясь при этом на успешную работу.

Исключения

В C++ *исключение* — это объект, который передается из области кода, где возникла проблема, в ту часть кода, где эта проблема обрабатывается. Тип исключения определяет, какая область кода будет обрабатывать проблему и как содержимое переданного объекта, если он существует, может использоваться для обратной связи с пользователем.

Основная идея использования исключений довольно проста.

- Фактическое распределение ресурсов (например, распределение памяти или захват файла) обычно осуществляется в программе на низком уровне.

- Выход из исключительной ситуации, возникшей при сбое операции из-за нехватки памяти или захвата файла другим приложением обычно реализуется на высоком уровне программирования в коде, описывающем взаимодействие программы с пользователем.
- Исключения обеспечивают переход от кода распределения ресурсов к коду обработки исключительной ситуации. Желательно, чтобы код обработки исключительной ситуации не только отслеживал ее появление, но и мог обеспечить элегантный выход из исключительной ситуации, например отмену выделения памяти в случае ее нехватки.

Как используются исключения

Создаются блоки `try` для помещения в них фрагментов кода, которые могут вызвать проблему, например:

```
try
{
SomeDangerousFunction();
}
```

Исключения, возникшие в блоках `try`, обрабатываются в блоках `catch`, например:

```
try
{
SomeDangerousFunction();
}
catch(OutOfMemory)
{
// предпринимаем некоторые действия
}
catch(FileNotFound)
{
// предпринимаем другие действия
}
```

Ниже приведены основные принципы использовании исключений.

1. Идентифицируйте те области программы, где начинается выполнение операции, которая могла бы вызвать исключительную ситуацию, и поместите их в блоки `try`.
2. Создайте блоки `catch` для перехвата исключений, если таковые возникнут, очистки выделенной памяти и информирования пользователя соответствующим образом. В листинге 20.1 иллюстрируется использование блоков `try` и `catch`.

Исключения — это объекты, которые используются для передачи информации о проблеме.

Блок `try` — это заключенный в фигурные скобки блок, содержащий фрагменты программы, способные вызвать исключительные ситуации.

Блок `catch` — это блок, который следует за блоком `try` и в котором выполняется обработка исключений.

При возникновении исключительной ситуации управление передается блоку `catch`, который следует сразу за текущим блоком `try`.

ПРИМЕЧАНИЕ

Некоторые очень старые компиляторы не поддерживают обработку исключений. Однако обработка исключений является частью стандарта ANSI C++. Все современные версии компиляторов полностью поддерживают эту возможность. Если у вас устаревший компилятор, вы не сможете скомпилировать и выполнить листинги, приведенные на этом занятии. Однако все же стоит прочитать представленный материал до конца, а затем вернуться к нему после обновления своего компилятора.

Листинг 20.1. Возникновение исключительной ситуации

```
1:      #include <iostream.h>
2:
3:      const int DefaultSize = 10;
4:
5:      class Array
6:      {
7:      public:
8:         // конструкторы
9:         Array(int itsSize = DefaultSize);
10:         Array(const Array &rhs);
11:         ~Array() {  delete [] pType;}
12:
13:         // операторы
14:         Array& operator=(const Array&);
15:         int& operator[](int offSet);
16:         const int& operator[](int offSet) const;
17:
18:         // методы доступа
19:         int GetitsSize() const {  return itsSize; }
20:
21:         // функция-друг
22:        friend ostream& operator<< (ostream&, const Array&);
23:
24:         class xBoundary { } ;  // определяем класс исключений
25:      private:
26:         int *pType;
27:         int  itsSize;
28:      } ;
29:
30:
31:      Array::Array(int size):
32:      itsSize(size)
33:      {
34:         pType = new int[size];
35:         for (int i = 0; i<size; i++)
36:           pType[i] = 0;
37:      }
38:
39:
40:      Array& Array::operator=(const Array &rhs)
41:      {
42:         if (this == &rhs)
```

```
43:          return *this;
44:       delete [] pType;
45:       itsSize = rhs.GetitsSize();
46:       pType = new int[itsSize];
47:       for (int i = 0; i<itsSize; i++)
48:          pType[i] = rhs[i];
49:       return *this;
50:    }
51:
52:    Array::Array(const Array &rhs)
53:    {
54:       itsSize = rhs.GetitsSize();
55:       pType = new int[itsSize];
56:       for (int i = 0; i<itsSize; i++)
57:          pType[i] = rhs[i];
58:    }
59:
60:
61:    int& Array::operator[](int offSet)
62:    {
63:       int size = GetitsSize();
64:       if (offSet >= 0 && offSet < GetitsSize())
65:          return pType[offSet];
66:       throw xBoundary();
67:       return pType[0]; // требование компилятора
68:    }
69:
70:
71:    const int& Array::operator[](int offSet) const
72:    {
73:       int mysize = GetitsSize();
74:       if (offSet >= 0 && offSet < GetitsSize())
75:          return pType[offSet];
76:       throw xBoundary();
77:       return pType[0]; // требование компилятора
78:    }
79:
80:    ostream& operator<< (ostream& output, const Array& theArray)
81:    {
82:       for (int i = 0; i<theArray.GetitsSize(); i++)
83:          output << "[" << i << "] " << theArray[i] << endl;
84:       return output;
85:    }
86:
87:    int main()
88:    {
89:       Array intArray(20);
90:       try
91:       {
92:          for (int j = 0; j< 100; j++)
93:          {
94:             intArray[j] = j;
```

```
95:                cout << "intArray[" << j << "] okay..." << endl;
96:             }
97:          }
98:          catch (Array::xBoundary)
99:          {
100:             cout << "Unable to process your input!\ n";
101:          }
102:          cout << "Done.\ n";
103:          return 0;
104:       }
```

РЕЗУЛЬТАТ

```
intArray[0] okay...
intArray[1] okay...
intArray[2] okay...
intArray[3] okay...
intArray[4] okay...
intArray[5] okay...
intArray[6] okay...
intArray[7] okay...
intArray[8] okay...
intArray[9] okay...
intArray[10] okay...
intArray[11] okay...
intArray[12] okay...
intArray[13] okay...
intArray[14] okay...
intArray[15] okay...
intArray[16] okay...
intArray[17] okay...
intArray[18] okay...
intArray[19] okay...
Unable to process your input!
Done.
```

АНАЛИЗ В листинге 20.1 представлен несколько усеченный класс `Array`, основанный на шаблоне, разработанном на занятии 19.

В строке 24 объявляется новый класс `xBoundary` внутри объявления внешнего класса `Array`.

В этом новом классе ни по каким внешним признакам нельзя узнать класс обработки исключительных ситуаций. Он чрезвычайно прост и не содержит никаких данных и методов. Тем не менее это вполне работоспособный класс.

На самом деле было бы неправильно говорить, что он не содержит никаких методов, потому что компилятор автоматически назначает ему стандартный конструктор, деструктор, конструктор-копировщик и оператор присваивания (=), поэтому у него фактически есть четыре метода, но нет данных.

Обратите внимание на то, что его объявление внутри класса `Array` служит только для объединения двух классов. Как описано в главе 15, класс `Array` не имеет никакого особого доступа к классу `xBoundary`, да и класс `xBoundary` не наделен преимущественным доступом к членам класса `Array`.

В строках 61–68 и 71–78 операторы индексирования ([]) замещены таким образом, чтобы предварительно анализировать введенный индекс смещения и, если оно окажется вне допустимого диапазона, обратиться к классу xBoundary для создания исключения. Назначение круглых скобок состоит в том, чтобы отделить обращение к конструктору класса xBoundary от использования константы перечисления. Обратите внимание, что некоторые компиляторы компании Microsoft требуют, чтобы определение функции в любом случае заканчивалось строкой с оператором return, согласующейся по типу с прототипом функции (в данном случае возвращение ссылки на целочисленное значение), несмотря на то что в случае возникновения исключительной ситуации в строке 66 выполнение программы никогда не достигнет строки 67. Этот пример говорит о том, что логические ошибки не чужды даже компании Microsoft!

В строке 90 ключевым словом try начинается блок отслеживания исключительных ситуаций, который оканчивается в строке 97. Внутри этого блока в массив, объявленный в строке 89, добавляется 101 целое число.

В строке 98 объявлен блок catch для перехвата исключений класса xBoundary.

В управляющей программе в строках 87–104 создается блок try, в котором инициализируется каждый член массива. Когда переменная j (строка 92) увеличится до 20, осуществляется доступ к члену, соответствующему смещению 20. Это приводит к невыполнению условия проверки в строке 64, в результате чего замещенный оператор индексирования operator[] генерирует исключение класса xBoundary (строка 66).

Управление программой передается к блоку catch в строке 98, и исключение перехватывается или обрабатывается оператором catch в той же строке, которая печатает сообщение об ошибках. Программа доходит до конца блока catch в строке 100.

Блок отслеживания исключительных ситуаций

Этот блок представляет собой набор выражений, начинающийся ключевым словом try, за которым следует открывающая фигурная скобка; завершается блок закрывающей фигурной скобкой.

Пример:

```
try
{
Function();
}
```

Блок обработки исключительных ситуаций

Этот блок представляет собой набор строк, каждая из них начинается ключевым словом catch, за которым следует тип исключения, заданный в круглых скобках. Затем идет открывающая фигурная скобка. Завершается блок catch закрывающей фигурной скобкой.

Пример:

```
try
{
Function();
}
catch (OutOfMemory)
{
// выполняем действие
}
```

Использование блоков try и catch

Часто не так уж просто решить, куда поместить блоки try, поскольку не всегда очевидно, какие действия могут вызвать исключительную ситуацию. Следующий вопрос состоит в том, где перехватывать исключение. Может быть, вы захотите генерировать исключения, связанные с памятью, там, где память распределяется, но в то же время перехватывать исключения стоит только в высокоуровневой части программы, связанной с интерфейсом пользователя.

При попытке определить местоположение блока try выясните, где в программе происходит распределение памяти или других ресурсов. При ошибках, связанных с выходом значений за допустимые пределы, вводом некорректных данных и пр., нужно использовать другие подходы.

Перехват исключений

Перехват исключений происходит следующим образом. Когда генерируется исключение, исследуется стек вызовов. Он представляет собой список обращений к функциям, создаваемый по мере того, как одна часть программы вызывает другую функцию.

Стек вызовов отслеживает путь выполнения программы. Если функция main() вызывает функцию Animal::GetFavoriteFood(), а функция GetFavoriteFood() — функцию Animal::LookupPreferences(), которая, в свою очередь, вызывает функцию fstream::operator>>(), то все эти вызовы заносятся в стек вызовов. Рекурсивная функция может оказаться в стеке вызовов много раз.

Исключение передается в стек вызовов для каждого вложенного блока. По мере прохождения стека вызываются деструкторы для локальных объектов, в результате чего эти объекты разрушаются.

За каждым блоком try следует один или несколько блоков catch. Если сгенерированное исключение соответствует одному из исключений операторов catch, то выполняется код блока этого оператора. Если же исключению не соответствует ни один из операторов catch, прохождение стека продолжается.

Если исключение пройдет весь путь к началу программы (функции main()) и все еще не будет перехвачено, вызывается встроенный обработчик, который завершит программу.

Прохождение исключения по стеку можно сравнить с поездкой по улице с односторонним движением. По мере прохождения стека его объекты разрушаются. Назад дороги нет. Если исключение перехвачено и обработано, программа продолжит работу после блока catch, который перехватил это исключение.

Таким образом, в листинге 20.1 выполнение программы продолжится со строки 101 — первой строки после блока tr catch, перехватившего исключение xBoundary. Помните, что при возникновении исключительной ситуации выполнение программы продолжается после блока catch, а не после того места, где она возникла.

Использование нескольких операторов catch

В некоторых случаях выполнение одного выражения потенциально может быть причиной возникновения нескольких исключительных ситуаций. В этом случае нужно использовать несколько операторов catch, следующих друг за другом, подобно конструкции с оператором switch. При этом эквивалентом оператора default будет выражение catch(...), которое следует понимать как "перехватить все". Отслеживание нескольких возможных исключений показано в листинге 20.2.

Листинг 20.2. Множественные исключения

```
1:    #include <iostream.h>
2:
3:    const int DefaultSize = 10;
4:
5:    class Array
6:    {
7:    public:
8:       // конструкторы
9:       Array(int itsSize = DefaultSize);
10:       Array(const Array &rhs);
11:       ~Array() {  delete [] pType;}
12:
13:       // операторы
14:       Array& operator=(const Array&);
15:       int& operator[](int offSet);
16:       const int& operator[](int offSet) const;
17:
18:       // методы доступа
19:       int GetitsSize() const {  return itsSize; }
20:
21:       // функция-друг
22:      friend ostream& operator<< (ostream&, const Array&);
23:
24:     // определение классов исключений
25:       class xBoundary { } ;
26:       class xTooBig { } ;
27:       class xTooSmall{ } ;
28:       class xZero { } ;
29:       class xNegative { } ;
30:    private:
31:       int *pType;
32:       int  itsSize;
33:    } ;
34:
35:       int& Array::operator[](int offSet)
36:       {
37:            int size = GetitsSize();
38:            if (offSet >= 0 && offSet < GetitsSize())
39:               return pType[offSet];
40:            throw xBoundary();
41:            return pType[0];  // требование компилятора
42:       }
43:
44:
45:       const int& Array::operator[](int offSet) const
46:       {
47:            int mysize = GetitsSize();
48:            if (offSet >= 0 && offSet < GetitsSize())
```

```
49:             return pType[offSet];
50:          throw xBoundary();
51:
52:          return pType[0];  // требование компилятора
53:       }
54:
55:
56:    Array::Array(int size):
57:    itsSize(size)
58:    {
59:       if (size == 0)
60:          throw xZero();
61:       if (size < 10)
62:          throw xTooSmall();
63:       if (size > 30000)
64:          throw xTooBig();
65:       if (size < 1)
66:          throw xNegative();
67:
68:       pType = new int[size];
69:       for (int i = 0; i<size; i++)
70:         pType[i] = 0;
71:    }
72:
73:
74:
75:    int main()
76:    {
77:
78:       try
79:       {
80:          Array intArray(0);
81:          for (int j = 0; j< 100; j++)
82:          {
83:             intArray[j] = j;
84:             cout << "intArray[" << j << "] okay...\ n";
85:          }
86:       }
87:       catch (Array::xBoundary)
88:       {
89:          cout << "Unable to process your input!\ n";
90:       }
91:       catch (Array::xTooBig)
92:       {
93:          cout << "This array is too big...\ n";
94:       }
95:       catch (Array::xTooSmall)
96:       {
97:          cout << "This array is too small...\ n";
98:       }
```

```
99:       catch (Array::xZero)
100:       {
101:          cout << "You asked for an array";
102:          cout << " of zero objects!\ n";
103:       }
104:       catch (...)
105:       {
106:          cout << "Something went wrong!\ n";
107:        }
108:         cout << "Done.\ n";
109:       return 0;
110:      }
```

Результат

```
You asked for an array of zero objects!
Done
```

Анализ В строках 26–29 создается четыре новых класса: `xTooBig`, `xTooSmall`, `xZero` и `xNegative`. В строках 56–71 проверяется размер массива, переданный конструктору. Если он слишком велик или мал, а также отрицательный или нулевой, генерируется исключение.

За блоком `try` следует несколько операторов `catch` для каждой исключительной ситуации, кроме исключения, связанного с передачей отрицательного размера. Данное исключение перехватывается оператором `catch(...)` в строке 104.

Опробуйте эту программу с рядом значений для размера массива. Затем попытайтесь ввести значение `-5`. Вы могли бы ожидать, что будет вызвано исключение `xNegative`, но этому помешает порядок проверок, заданный в конструкторе: проверка `size < 10` выполняется до проверки `size < 1`. Чтобы исправить этот недостаток, поменяйте строки 61 и 62 со строками 65 и 66 и перекомпилируйте программу.

Наследование исключений

Исключения — это классы, а раз так, то от них можно производить другие классы. Предположим, что нам нужно создать класс `xSize` и произвести от него классы `xZero`, `xTooSmall`, `xTooBig` и `xNegative`. В результате для одних функций можно установить перехват ошибки `xSize`, а для других — перехват типов ошибок, произведенных от `xSize`. Реализация этой идеи показана в листинге 20.3.

Листинг 20.3. Наследование исключений

```
1:       #include <iostream.h>
2:
3:       const int DefaultSize = 10;
4:
5:       class Array
6:       {
7:       public:
8:          // конструкторы
9:          Array(int itsSize = DefaultSize);
10:          Array(const Array &rhs);
```

```
11:        ~Array() {  delete [] pType;}
12:
13:        // операторы
14:        Array& operator=(const Array&);
15:        int& operator[](int offSet);
16:        const int& operator[](int offSet) const;
17:
18:        // методы доступа
19:        int GetitsSize() const {  return itsSize; }
20:
21:        // функция-друг
22:       friend ostream& operator<< (ostream&, const Array&);
23:
24:       // определения классов исключений
25:        class xBoundary { } ;
26:        class xSize { } ;
27:        class xTooBig : public xSize { } ;
28:        class xTooSmall : public xSize { } ;
29:        class xZero  : public xTooSmall { } ;
30:        class xNegative  : public xSize { } ;
31:     private:
32:        int *pType;
33:        int  itsSize;
34:     } ;
35:
36:
37:     Array::Array(int size):
38:     itsSize(size)
39:     {
40:        if (size == 0)
41:           throw xZero();
42:        if (size > 30000)
43:           throw xTooBig();
44:        if (size <1)
45:           throw xNegative();
46:        if (size < 10)
47:           throw xTooSmall();
48:
49:        pType = new int[size];
50:        for (int i = 0; i<size; i++)
51:          pType[i] = 0;
52:     }
53:
54:         int& Array::operator[](int offSet)
55:       {
56:            int size = GetitsSize();
57:            if (offSet >= 0 && offSet < GetitsSize())
58:               return pType[offSet];
59:            throw xBoundary();
60:            return pType[0]; // требование компилятора
61:       }
```

```
62:
63:
64:      const int& Array::operator[](int offSet) const
65:      {
66:           int mysize = GetitsSize();
67:           if (offSet >= 0 && offSet < GetitsSize())
68:              return pType[offSet];
69:           throw xBoundary();
70:
71:           return pType[0];  // требование компилятора
72:      }
73:
74:    int main()
75:    {
76:
77:       try
78:       {
79:          Array intArray(0);
80:          for (int j = 0; j< 100; j++)
81:          {
82:             intArray[j] = j;
83:             cout << "intArray[" << j << "] okay...\ n";
84:          }
85:       }
86:       catch (Array::xBoundary)
87:       {
88:          cout << "Unable to process your input!\ n";
89:       }
90:       catch (Array::xTooBig)
91:       {
92:          cout << "This array is too big...\ n";
93:       }
94:
95:       catch (Array::xTooSmall)
96:       {
97:          cout << "This array is too small...\ n";
98:       }
99:       catch (Array::xZero)
100:       {
101:          cout << "You asked for an array";
102:          cout << " of zero objects!\ n";
103:       }
104:
105:
106:       catch (...)
107:       {
108:          cout << "Something went wrong!\ n";
109:        }
110:         cout << "Done.\ n";
111:       return 0;
112:       }
```

```
This array is too small...
Done.
```

Анализ: Здесь существенно изменены строки 27–30, где устанавливается иерархия классов. Классы `xTooBig`, `xTooSmall` и `xNegative` произведены от класса `xSize`, а класс `xZero` — от класса `xTooSmall`.

Класс `Array` создается с нулевым размером, но что это значит? Казалось бы, неправильное исключение будет тут же перехвачено! Однако тщательно исследуйте блок `catch`, и вы поймете, что, прежде чем искать исключение типа `xZero`, в нем ищется исключение типа `xTooSmall`. А поскольку возник объект класса `xZero`, который также является объектом класса `xTooSmall`, то он перехватывается обработчиком исключения `xTooSmall`. Будучи уже обработанным, это исключение не передается другим обработчикам, так что обработчик исключений типа `xZero` никогда не вызывается.

Решение этой проблемы лежит в тщательном упорядочении обработчиков таким образом, чтобы самые специфические из них стояли в начале, а более общие следовали за ними. В данном примере для решения проблемы достаточно поменять местами два обработчика — `xZero` и `xTooSmall`.

Данные в классах исключений и присвоение имен объектам исключений

Часто для того, чтобы программа могла отреагировать должным образом на ошибку, полезно знать несколько больше, чем просто тип возникшего исключения. Классы исключений — это такие же классы, как и любые другие. Вы абсолютно свободно можете добавлять любые данные в эти классы, инициализировать их с помощью конструктора и считывать их значения в любое время, как показано в листинге 20.4.

Листинг 20.4. Возвращение данных из объекта исключения

```
1:      #include <iostream.h>
2:
3:      const int DefaultSize = 10;
4:
5:      class Array
6:      {
7:      public:
8:         // конструкторы
9:         Array(int itsSize = DefaultSize);
10:         Array(const Array &rhs);
11:         ~Array() {  delete [] pType;}
12:
13:         // операторы
14:         Array& operator=(const Array&);
15:         int& operator[](int offSet);
16:         const int& operator[](int offSet) const;
17:
18:         // методы доступа
```

```
19:        int GetitsSize() const {  return itsSize; }
20:
21:        // функция-друг
22:       friend ostream& operator<< (ostream&, const Array&);
23:
24:      // определение классов исключений
25:        class xBoundary { } ;
26:        class xSize
27:        {
28:        public:
29:           xSize(int size):itsSize(size) { }
30:           ~xSize(){ }
31:           int GetSize() {  return itsSize; }
32:        private:
33:           int itsSize;
34:        } ;
35:
36:        class xTooBig : public xSize
37:        {
38:        public:
39:           xTooBig(int size):xSize(size){ }
40:        } ;
41:
42:        class xTooSmall : public xSize
43:        {
44:        public:
45:           xTooSmall(int size):xSize(size){ }
46:        } ;
47:
48:        class xZero  : public xTooSmall
49:        {
50:        public:
51:           xZero(int size):xTooSmall(size){ }
52:        } ;
53:
54:        class xNegative : public xSize
55:        {
56:        public:
57:           xNegative(int size):xSize(size){ }
58:        } ;
59:
60:     private:
61:        int *pType;
62:        int  itsSize;
63:     } ;
64:
65:
66:     Array::Array(int size):
67:     itsSize(size)
68:     {
```

```
69:         if (size == 0)
70:            throw xZero(size);
71:         if (size > 30000)
72:            throw xTooBig(size);
73:         if (size <1)
74:            throw xNegative(size);
75:         if (size < 10)
76:            throw xTooSmall(size);
77:
78:         pType = new int[size];
79:         for (int i = 0; i<size; i++)
80:           pType[i] = 0;
81:      }
82:
83:
84:      int& Array::operator[] (int offSet)
85:      {
86:          int size = GetitsSize();
87:          if (offSet >= 0 && offSet < GetitsSize())
88:            return pType[offSet];
89:          throw xBoundary();
90:          return pType[0];
91:      }
92:
93:      const int& Array::operator[] (int offSet) const
94:      {
95:          int size = GetitsSize();
96:          if (offSet >= 0 && offSet < GetitsSize())
97:            return pType[offSet];
98:          throw xBoundary();
99:          return pType[0];
100:     }
101:
102:      int main()
103:      {
104:
105:         try
106:         {
107:            Array intArray(9);
108:            for (int j = 0; j< 100; j++)
109:            {
110:               intArray[j] = j;
111:               cout << "intArray[" << j << "] okay..." << endl;
112:            }
113:         }
114:         catch (Array::xBoundary)
115:         {
116:            cout << "Unable to process your input!\ n";
117:         }
118:        catch (Array::xZero theException)
```

```
119:        {
120:           cout << "You asked for an Array of zero objects! " << endl;
121:           cout << "Received " << theException.GetSize() << endl;
122:        }
123:        catch (Array::xTooBig theException)
124:        {
125:           cout << "This Array is too big... " << endl;
126:           cout << "Received " << theException.GetSize() << endl;
127:        }
128:        catch (Array::xTooSmall theException)
129:        {
130:           cout << "This Array is too small... " << endl;
131:           cout << "Received " << theException.GetSize() << endl;
132:        }
133:        catch (...)
134:        {
135:           cout << "Something went wrong, but I've no idea what!\ n";
136:        }
137:        cout << "Done.\ n";
138:        return 0;
139:     }
```

РЕЗУЛЬТАТ

```
This array is too small...
Received 9
Done.
```

АНАЛИЗ

Объявление класса `xSize` было изменено таким образом, чтобы включить в него переменную-член `itsSize` (строка 33) и функцию-член `GetSize()` (строка 31). Кроме того, был добавлен конструктор, который принимает целое число и инициализирует переменную-член, как показано в строке 29.

Производные классы объявляют конструктор, который лишь инициализирует базовый класс. При этом никакие другие функции объявлены не были (частично из экономии места в листинге).

Операторы `catch` в строках 114–136 изменены таким образом, чтобы создавать именованный объект исключения (`theException`), который используется в теле блока `catch` для доступа к данным, сохраняемым в переменной-члене `itsSize`.

ПРИМЕЧАНИЕ

При работе с исключениями следует помнить об их сути: если уж оно возникло, значит, что-то не в порядке с распределением ресурсов, и обработку этого исключения нужно записать таким образом, чтобы вновь не создать ту же проблему. Следовательно, если вы создаете исключение `OutOfMemory`, то не стоит в конструкторе этого класса пытаться выделить память для какого-либо объекта.

Весьма утомительно писать вручную все эти конструкции с операторами `catch`, каждый из которых должен выводить свое сообщение. Тем более, что при увеличении объема программы стремительно возрастает вероятность возникновения в ней ошибок. Лучше переложить эту работу на объект исключения, который сам должен определять тип

исключения и выбирать соответствующее сообщение. В листинге 20.5 для решения этой проблемы использован подход, который в большей степени отвечает принципам объектно-ориентированного программирования. В классах исключений применяются виртуальные функции, обеспечивающие полиморфизм объекта исключения.

Листинг 20.5. Передача аргументов как ссылок и использование виртуальных функций в классах исключений

```
1:      #include <iostream.h>
2:
3:      const int DefaultSize = 10;
4:
5:      class Array
6:      {
7:      public:
8:         // конструкторы
9:         Array(int itsSize = DefaultSize);
10:         Array(const Array &rhs);
11:         ~Array() {  delete [] pType;}
12:
13:         // операторы
14:         Array& operator=(const Array&);
15:         int& operator[](int offSet);
16:         const int& operator[](int offSet) const;
17:
18:         // методы доступа
19:         int GetitsSize() const {  return itsSize; }
20:
21:         // функция-друг
22:        friend ostream& operator<<
23:            (ostream&, const Array&);
24:
25:       // определение классов исключений
26:         class xBoundary { } ;
27:         class xSize
28:         {
29:         public:
30:            xSize(int size):itsSize(size) { }
31:            ~xSize(){ }
32:            virtual int GetSize() {  return itsSize; }
33:            virtual void PrintError()
34:            {
35:                cout << "Size error. Received: ";
36:                cout << itsSize << endl;
37:            }
38:         protected:
39:            int itsSize;
40:         } ;
41:
42:         class xTooBig : public xSize
```

```
43:        {
44:        public:
45:           xTooBig(int size):xSize(size){ }
46:           virtual void PrintError()
47:           {
48:               cout << "Too big. Received: ";
49:               cout << xSize::itsSize << endl;
50:           }
51:        } ;
52:
53:        class xTooSmall : public xSize
54:        {
55:        public:
56:           xTooSmall(int size):xSize(size){ }
57:           virtual void PrintError()
58:           {
59:               cout << "Too small. Received: ";
60:               cout << xSize::itsSize << endl;
61:           }
62:        } ;
63:
64:        class xZero  : public xTooSmall
65:        {
66:        public:
67:           xZero(int size):xTooSmall(size){ }
68:           virtual void PrintError()
69:           {
70:               cout << "Zero!. Received: " ;
71:               cout << xSize::itsSize << endl;
72:           }
73:        } ;
74:
75:        class xNegative : public xSize76:         {
77:        public:
78:           xNegative(int size):xSize(size){ }
79:           virtual void PrintError()
80:           {
81:               cout << "Negative! Received: ";
82:               cout << xSize::itsSize << endl;
83:           }
84:        } ;
85:
86:     private:
87:        int *pType;
88:        int  itsSize;
89:     } ;
90:
91:     Array::Array(int size):
92:     itsSize(size)
93:     {
```

```
94:       if (size == 0)
95:          throw xZero(size);
96:       if (size > 30000)
97:          throw xTooBig(size);
98:       if (size <1)
99:          throw xNegative(size);
100:       if (size < 10)
101:          throw xTooSmall(size);
102:
103:       pType = new int[size];
104:       for (int i = 0; i<size; i++)
105:         pType[i] = 0;
106:    }
107:
108:    int& Array::operator[] (int offSet)
109:    {
110:        int size = GetitsSize();
111:        if (offSet >= 0 && offSet < GetitsSize())
112:          return pType[offSet];
113:        throw xBoundary();
114:        return pType[0];
115:    }
116:
117:    const int& Array::operator[] (int offSet) const
118:    {
119:        int size = GetitsSize();
120:        if (offSet >= 0 && offSet < GetitsSize())
121:          return pType[offSet];
122:        throw xBoundary();
123:        return pType[0];
124:    }
125:
126:     int main()
127:     {
128:
129:        try
130:        {
131:           Array intArray(9);
132:           for (int j = 0; j< 100; j++)
133:           {
134:              intArray[j] = j;
135:              cout << "intArray[" << j << "] okay...\ n";
136:           }
137:        }
138:        catch (Array::xBoundary)
139:       {
140:          cout << "Unable to process your input!\ n";
141:       }
142:       catch (Array::xSize& theException)
143:       {
```

```
144:            theException.PrintError();
145:         }
146:         catch (...)
147:         {
148:            cout << "Something went wrong!\ n";
149:         }
150:         cout << "Done.\ n";
151:        return 0;
152:      }
```

РЕЗУЛЬТАТ

```
Too small! Received: 9
Done.
```

АНАЛИЗ В листинге 20.5 показано объявление виртуального метода `PrintError()` в классе `xSize`, который выводит сообщения об ошибках и истинный размер класса. Этот метод замещается в каждом производном классе исключения.

В строке 142 объявляется объект исключения, который является ссылкой. При вызове функции `PrintError()` со ссылкой на объект благодаря полиморфизму вызывается нужная версия функции `PrintError()`. В результате программный код становится яснее, проще для понимания, а следовательно, и для дальнейшей поддержки.

Исключения и шаблоны

При создании исключений, предназначенных для работы с шаблонами, есть два варианта решений. Можно создавать исключение прямо в шаблоне, и тогда они будут доступны для каждого экземпляра шаблона, а можно использовать классы исключений, созданные вне объявления шаблона. Оба этих подхода показаны в листинге 20.6.

Листинг 20.6. Использование исключений с шаблонами

```
1:    #include <iostream.h>
2:
3:    const int DefaultSize = 10;
4:    class xBoundary { } ;
5:
6:    template <class T>
7:    class Array
8:    {
9:    public:
10:       // конструкторы
11:       Array(int itsSize = DefaultSize);
12:       Array(const Array &rhs);
13:       ~Array() {  delete [] pType;}
14:
15:       // операторы
16:       Array& operator=(const Array<T>&);
17:       T& operator[](int offSet);
18:       const T& operator[](int offSet) const;
19:
```

```
20:       // методы доступа
21:       int GetitsSize() const {  return itsSize; }
22:
23:       // функция-друг
24:      friend ostream& operator<< (ostream&, const Array<T>&);
25:
26:     // определение классов исключений
27:
28:       class xSize { } ;
29:
30:    private:
31:       int *pType;
32:       int  itsSize;
33:    } ;
34:
35:    template <class T>
36:    Array<T>::Array(int size):
37:    itsSize(size)
38:    {
39:       if (size <10 || size > 30000)
40:          throw xSize();
41:       pType = new T[size];
42:       for (int i = 0; i<size; i++)
43:         pType[i] = 0;
44:    }
45:
46:    template <class T>
47:    Array<T>& Array<T>::operator=(const Array<T> &rhs)
48:    {
49:       if (this == &rhs)
50:          return *this;
51:       delete [] pType;
52:       itsSize = rhs.GetitsSize();
53:       pType = new T[itsSize];
54:       for (int i = 0; i<itsSize; i++)
55:          pType[i] = rhs[i];
56:    }
57:    template <class T>
58:    Array<T>::Array(const Array<T> &rhs)
59:    {
60:       itsSize = rhs.GetitsSize();
61:       pType = new T[itsSize];
62:       for (int i = 0; i<itsSize; i++)
63:          pType[i] = rhs[i];
64:    }
65:
66:    template <class T>
67:    T& Array<T>::operator[](int offSet)
68:    {
69:       int size = GetitsSize();
70:       if (offSet >= 0 && offSet < GetitsSize())
```

```
71:          return pType[offSet];
72:       throw xBoundary();
73:       return pType[0];
74:    }
75:
76:    template <class T>
77:    const T& Array<T>::operator[](int offSet) const
78:    {
79:       int mysize = GetitsSize();
80:       if (offSet >= 0 && offSet < GetitsSize())
81:          return pType[offSet];
82:       throw xBoundary();
83:    }
84:
85:    template <class T>
86:    ostream& operator<< (ostream& output, const Array<T>& theArray)
87:    {
88:       for (int i = 0; i<theArray.GetitsSize(); i++)
89:          output << "[" << i << "] " << theArray[i] << endl;
90:       return output;
91:    }
92:
93:
94:    int main()
95:    {
96:
97:       try
98:       {
99:          Array<int> intArray(9);
100:          for (int j = 0; j< 100; j++)
101:          {
102:             intArray[j] = j;
103:             cout << "intArray[" << j << "] okay..." << endl;
104:          }
105:       }
106:       catch (xBoundary)
107:       {
108:          cout << "Unable to process your input!\ n";
109:       }
110:       catch (Array<int>::xSize)
111:       {
112:          cout << "Bad Size!\ n";
113:       }
114:
115:       cout << "Done.\ n";
116:       return 0;
117:    }
```

РЕЗУЛЬТАТ:

```
You asked for an array of zero objects!
Done
```

АНАЛИЗ Первое исключение, xBoundary, объявлено вне определения шаблона в строке 4; второе исключение, xSize, — внутри определения шаблона в строке 28.

Исключение xBoundary не связано с классом шаблона, но его можно использовать *так же, как и любой другой класс*. *Исключение* XSize *связано с шаблоном и должно* вызываться для экземпляра класса Array. Обратите внимание на разницу в синтаксисе двух операторов catch. Строка 106 содержит выражение catch (xBoundary), а строка 110 — выражение catch (Array<int>::xSize). Второй вариант связан с обращением к исключению экземпляра целочисленного массива.

Исключения без ошибок

Когда программисты C++ после работы собираются за чаркой виртуального пива в баре киберпространства, в их задушевных беседах часто затрагивается вопрос, можно ли использовать исключения не только для отслеживания ошибок, но и для выполнения рутинных процедур. Есть мнение, что использование исключений следует ограничить только отслеживанием предсказуемых исключительных ситуаций, для чего, собственно, исключения и создавались.

В то же время другие считают, что исключения предоставляют эффективный способ возврата сквозь несколько уровней вызовов функций, не подвергаясь при этом опасности утечки памяти. Чаще всего приводится следующий пример. Пользователь формирует запрос на некоторую операцию в среде GUI (графический интерфейс пользователя). Часть кода, которая перехватывает этот запрос, должна вызвать функцию-член менеджера диалоговых окон, которая, в свою очередь, вызывает код, обрабатывающий этот запрос. Этот код вызывает другой код, который решает, какое диалоговое окно использовать, и, в свою очередь, вызывает код, чтобы отобразить на экране это диалоговое окно. И теперь уже этот код наконец-то вызывает другой код, который обрабатывает данные, вводимые пользователем. Если пользователь щелкнет на кнопке Cancel (Отменить), код должен возвратиться к самому первому вызывающему методу, где обрабатывался первоначальный запрос.

Один подход к решению этой проблемы состоит в том, чтобы поместить блок try сразу за тем блоком программы, где формируется исходный запрос, и перехватывать объект исключения CancelDialog, который генерируется обработчиком сообщений для кнопки Cancel. Это безопасно и эффективно, хотя щелчок на кнопке Cancel по сути своей не относится к исключительной ситуации.

Чтобы решить, насколько правомочно такое использование исключений, попытайтесь ответить на следующие вопросы: станет ли в результате программа проще или, наоборот, труднее для понимания; действительно ли уменьшится риск возникновения ошибок и утечки памяти; труднее или проще теперь станет поддержка такой программы? Безусловно, объективно ответить на эти вопросы сложно: многое зависит от привычек и субъективных взглядов программиста, из-за чего, кстати, и возникают споры вокруг этих вопросов.

Ошибки и отладка программы

Почти все современные среды разработки содержат один или несколько встроенных эффективных отладчиков. Основная идея использования отладчика такова: отладчик загружает и выполняет исходный код программы в режиме, удобном для отслеживания выполнения отдельных строк программы и выявления ошибок.

Все компиляторы позволяют компилировать программы с использованием символов или без них. Компилирование с символами указывает компилятору на необходимость установки взаимосвязей между исходным кодом файлов источников и сгенерированной программой, благодаря чему отладчик может указать на строку исходного кода, которая соответствует следующему действию в вашей программе.

Полноэкранные символьные отладчики превосходно справляются с этой сложной работой. После загрузки отладчик считывает весь исходный код программы и отображает его в окне. Отладчик позволяет проходить в пошаговом режиме через все строки программы в порядке их выполнения.

При работе с большинством отладчиков можно переключаться между исходным кодом и выводом на экран, чтобы видеть результаты выполнения каждой команды. Полезной также является возможность определения текущего значения любой переменной, в том числе переменных-членов классов и значений в ячейках области динамического обмена, на которые ссылаются указатели программы, а также просмотр сложных структур данных. Отладчики предоставляют ряд утилит, позволяющих устанавливать в коде программы точки останова, выводить контрольные значения переменных, исследовать особенности распределения памяти и просматривать код ассемблера.

Точки останова

Точки останова — это команды, предназначенные для отладчика и означающие, что программа должна остановиться перед выполнением указанной строки. Это средство позволяет экономить время при отладке, выполняя программу в обычном режиме до того места, где установлена точка останова. После остановки выполнения программы можно проанализировать текущие значения переменных или продолжить работу программы в пошаговом режиме.

Анализ значений переменных

Можно указать отладчику на отображение значения конкретной переменной или на останов программы, когда заданная переменная будет читаться или записываться. Отладчик даже позволяет изменить значение переменной в процессе выполнения программы.

Исследование памяти

Время от времени важно просматривать реальные значения, содержащиеся в памяти. Современные отладчики могут отображать эти значения в понятном для пользователя виде, т.е. строки отображаются как символы, а числовые значения — как десятичные цифры, а не в двоичном коде. Современные отладчики C++ могут даже показывать целые классы с текущими значениями всех переменных-членов, включая указатель `this`.

Код ассемблера

Хотя чтения исходного кода иногда бывает достаточно для обнаружения ошибки, тем не менее можно указать отладчику на отображение реального кода ассемблера, сгенерированного для каждой строки исходного кода. Вы можете просмотреть значения регистраторов памяти и флагов и при желании настолько углубиться в дебри машинного кода, насколько нужно.

Научитесь пользоваться своим отладчиком. Это может оказаться самым мощным оружием в вашей священной войне с ошибками. Ошибки выполнения программы считаются наиболее трудными для поиска и устранения, и мощный отладчик в состоянии помочь вам в этом.

Резюме

Сегодня вы узнали, как создавать и использовать исключения, т.е. объекты, которые могут быть созданы в тех местах программы, где исполняемый код не может обработать ошибку или другую исключительную ситуацию, возникшую во время выполнения программы. Другие части программы, расположенные выше в стеке вызовов, выполняют блоки `catch`, которые перехватывают исключение и отвечают на возникшую исключительную ситуацию соответствующим образом.

Исключения — это нормальные созданные пользователем объекты, которые можно передавать в функции как значения или как ссылки. Они могут содержать данные и методы, а блок `catch` может использовать эти данные, чтобы определить, как справиться с возникшими проблемами.

Можно создать конструкции из нескольких блоков `catch`, но следует учитывать, что, как только исключение будет перехвачено отдельным оператором `catch`, оно не будет передаваться последующим блокам `catch`. Очень важно правильно упорядочить блоки `catch`, чтобы специфические блоки стояли выше более общих блоков.

На этом занятии также рассматривались некоторые основные принципы работы символьных отладчиков, включая использование таких средств, как точки останова, анализ значений переменных и т.д. Эти средства позволяют выполнить останов программы в той части, которая вызывает появление ошибки, и просмотреть значения переменных в ходе программы.

Вопросы и ответы

Зачем тратить время на программирование исключений? Не лучше ли устранять ошибки по мере их возникновения?

Часто одна и та же ошибка может возникать при выполнении разных функций программы. Использование исключений позволяет собрать коды отслеживания ошибок в одном месте программы. Кроме того, далеко не всегда возможно вписать код устранения ошибки в том месте программы, где эта ошибка возникает.

Зачем создавать исключения как объекты? Не проще ли записать код устранения ошибки?

Объекты более гибки и универсальны в использовании, чем обычные программные блоки. С объектами можно передать больше информации и снабдить конструктор и деструктор класса исключения функциями устранения возникшей ошибки.

Почему бы не использовать исключения не только для отслеживания исключительных ситуаций, но и для выполнения рутинных процессов? Разве не удобно использовать исключения для быстрого и безопасного возвращения по стеку вызовов к исходному состоянию программы?

Безусловно, и многие программисты на C++ используют исключения именно в этих целях. Но следует помнить, что прохождение исключения по стеку вызовов может оказаться не таким уж безопасным. Так, если объект был создан в области динамического обмена, а потом удален в стеке вызовов, это может привести к утечке памяти. Впрочем, при тщательном анализе программы и использовании современного компилятора эту проблему можно предупредить.

Кроме того, многие программисты считают, что использование исключений не по прямому назначению делает программу слишком запутанной и нелогичной.

Всегда ли следует перехватывать исключения сразу за блоком `try`, генерирующим это исключение?

Нет, в стеке вызовов перехват исключения может осуществляться в любом месте, после чего стек вызовов будет пройден то того места, где происходит обработка исключения.

Зачем использовать утилиту отладки, если те же функции можно осуществлять прямо во время компиляции с помощью объекта `cout` и условного выражения `#ifdef debug`?

В действительности утилита отладки предоставляет значительно больше средств и возможностей, таких как пошаговое выполнение программы, установка точек останова и анализ текущих значений переменных. При этом вам не приходится перегружать свой код многочисленными командами препроцессора и выражениями, которые никак не связаны с основным назначением программы.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и укрепления полученных знаний, а также ряд упражнений, которые помогут закрепить ваши практические навыки. Попытайтесь самостоятельно ответить на вопросы теста и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г. Не приступайте к изучению материала следующей главы, если для вас остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Что такое исключение?
2. Для чего нужен блок `try`?
3. Для чего используется оператор `catch`?
4. Какую информацию может содержать исключение?
5. Когда создается объект исключения?
6. Следует ли передавать исключения как значения или как ссылки?
7. Будет ли оператор `catch` перехватывать производные исключения, если он настроен на базовый класс исключения?
8. Если используются два оператора `catch`, один из которых настроен на базовое сообщение, а второй — на производное, то в каком порядке их следует расположить?
9. Что означает оператор `catch(...)`?
10. Что такое точка останова?

Упражнения

1. Запишите блок `try` и оператор `catch` для отслеживания и обработки простого исключения.
2. Добавьте в исключение, полученное в упражнении 1, переменную-член и метод доступа и используйте их в блоке оператора `catch`.

3. Унаследуйте новое исключение от исключения, полученного в упражнении 2. Измените блок оператора `catch` таким образом, чтобы в нем происходила обработка как производного, так и базового исключений.
4. Измените код упражнения 3, чтобы получить трехуровневый вызов функции.
5. **Жучки:** что не правильно в следующем коде?

```
#include "string"       //класс строк

class xOutOfMemory
{
public:
        xOutOfMemory(){  theMsg = new char[20];
             strcpy(theMsg, "error in memory");}
        ~xOutOfMemory(){  delete [] theMsg; cout
           << "Memory restored. " << endl; }
    char * Message()  {  return theMsg; }
private:
        char * theMsg;
} ;

main()
{
     try
     {
          char * var = new char;
          if ( var == 0 )
          {
             xOutOfMemory * px =
             new xOutOfMemory;
             throw px;
          }
     }

     catch( xOutOfMemory * theException )
     {
         cout << theException->Message() <<endl;
         delete theException;
     }
        return 0;
}
```

6. Данный пример содержит потенциальную ошибку, подобную возникающей при попытке выделить память для показа сообщения об ошибке в случае обнаружения нехватки свободной памяти. Вы можете протестировать эту программу, изменив строку `if (var == 0)` на `if (1)`, которая вызовет создание исключения.

Что дальше

Примите наши поздравления! Вы почти завершили изучение полного трехнедельного интенсивного курса введения в C++. К этому моменту у вас должно быть ясное понимание языка C++, но в современном программировании всегда найдутся еще не изученные области. В этой главе будут рассмотрены некоторые опущенные выше подробности, а затем намечен курс для дальнейшего освоения C++.

Большая часть кода файлов источника представлена командами на языке C++. *Компилятор превращает этот код в программу на машинном языке. Однако перед* запуском компилятора запускается препроцессор, благодаря чему можно воспользоваться возможностями условной компиляции. Итак, сегодня вы узнаете:

- Что представляет собой условная компиляция и как с ней обращаться
- Как записывать макросы препроцессора
- Как использовать препроцессор для обнаружения ошибок
- Как управлять значениями отдельных битов и использовать их в качестве флагов
- Какие шаги следует предпринять для дальнейшего эффективного изучения C++

Препроцессор и компилятор

При каждом запуске компилятора сначала запускается препроцессор, который ищет команды препроцессора, начинающиеся с символа фунта (#). При выполнении любой из этих команд в текст исходного кода вносятся некоторые изменения, в результате чего создается новый файл исходного кода. Этот новый файл является временным, и вы обычно его не видите, но можете дать команду компилятору сохранить его для последующего просмотра и использования.

Компилятор читает не исходный файл источника, а результат работы препроцессора и компилирует его в исполняемый файл программы. Вам уже приходилось встречаться с директивой препроцессора `#include`: она предписывает найти файл, имя которого следует за ней, и вставить текст этого файла по месту вызова. Этот эффект подобен следующему: вы полностью вводите данный файл прямо в свою исходную программу, причем к тому времени, когда компилятор получит исходный код, файл будет уже на месте.

Просмотр промежуточного файла

Почти каждый компилятор имеет ключ, который можно устанавливать или в интегрированной среде разработки, или в командной строке. С помощью этого ключа можно сообщить компилятору о том, что вы хотите сохранить промежуточный файл. Если вас действительно интересует содержимое этого файла, обратитесь к руководству по использованию компилятора, чтобы узнать, какие ключи можно для него устанавливать.

Использование директивы #define

Команда `#define` определяет строку подстановки. Строка

```
#define BIG 512
```

означает, что вы предписываете препроцессору заменять лексему `BIG` строкой `512` в любом месте программы. Эта запись *не является командой языка C++. Строка* `512` вставляются в исходную программу везде, где встречается лексема `BIG`. *Лексема* — это строка символов, которую можно применить там, где может использоваться любая строка, константа или какой-нибудь другой набор символов. Таким образом, при записи строк

```
#define BIG 512
int myArray[BIG];
```

промежуточный файл, создаваемый препроцессором, будет иметь такой вид:

```
int myArray[512];
```

Обратите внимание, что в коде исчезла команда `#define`. Из промежуточного файла все директивы препроцессора удаляются, поэтому они отсутствуют в конечном варианте кода источника.

Использование директивы #define для создания констант

Один вариант использования директивы `#define` - это создание констант. Однако этим не стоит злоупотреблять, поскольку директива `#define` просто выполняет замену строки и не осуществляет никакого контроля за *соответствием типов*. Как пояснялось на занятии, посвященном константам, гораздо безопаснее вместо директивы `#define` использовать ключевое слово `const`.

Использование директивы #define для тестирования

Второй способ использования директивы `#define` *состоит в простом объявлении того,* что данная лексема определена в программе. Например, можно записать следующее:

```
#define BIG
```

В программе можно проверить, была ли определена лексема `BIG`, и предпринять соответствующие меры. Для подобной проверки используются такие команды препроцессора, как `#ifdef` (если определена) и `#ifndef` (если не определена). За обеими

командами должна следовать команда #endif, которую необходимо установить до завершения блока (до следующей закрывающей фигурной скобки).

Директива #ifdef принимает значение, равное истине, если тестируемая лексема уже была определена. Поэтому можем записать следующее:

```
#ifdef DEBUG
cout << "Строка DEBUG определена";
#endif
```

Когда препроцессор читает директиву #ifdef, он проверяет построенную им самим таблицу, чтобы узнать, была ли уже определена в программе лексема DEBUG. Если да, то #ifdef возвращает значение true, и все, что находится до следующей директивы #else или #endif, записывается в промежуточный файл для компиляции. Если эта директива возвращает значение false, то ни одна строка кода, находящаяся между директивами #ifdef DEBUG и #endif, не будет записана в промежуточный файл, т.е. вы получите такой вариант промежуточного файла, как будто этих строк никогда и не было в исходном коде.

Обратите внимание, что директива #ifndef является логической противоположностью директивы #ifdef. Директива #ifndef возвращает true в том случае, если до этой точки в программе заданная лексема не была определена.

Команда препроцессора #else

Как вы правильно предположили, директиву #else можно вставить между #ifdef (или #ifndef) и завершающей директивой #endif. Использование этих директив показано в листинге 21.1.

Листинг 21.1. Использование директивы #define

```
1:    #define DemoVersion
2:    #define NT_VERSION 5
3:    #include <iostream.h>
4:
5:
6:    int main()
7:    {
8:
9:    cout << "Checking on the definitions of DemoVersion,
               NT_VERSION _and WINDOWS_VERSION...\ n";
10:
11:   #ifdef DemoVersion
12:      cout << "DemoVersion defined.\ n";
13:   #else
14:      cout << "DemoVersion not defined.\ n";
15:   #endif
16:
17:   #ifndef NT_VERSION
18:      cout << "NT_VERSION not defined!\ n";
19:   #else
20:      cout << "NT_VERSION defined as: " << NT_VERSION << endl;
21:   #endif
```

```
22:
23:    #ifdef WINDOWS_VERSION
24:       cout << "WINDOWS_VERSION defined!\ n";
25:    #else
26:       cout << "WINDOWS_VERSION was not defined.\ n";
27:    #endif
28:
29:     cout << "Done.\ n";
30:     return 0;
31:    }
```

```
hecking on the definitions of DemoVersion, NT_VERSION_and
WINDOWS_VERSION...
DemoVersion defined.
NT_VERSION defined as: 5
WINDOWS_VERSION was not defined.
Done.
```

АНАЛИЗ В строках 1 и 2 определяются лексемы `DemoVersion` и `NT_VERSION`, причем лексеме `NT_VERSION` назначается литерал `5`. В строке 11 проверяется определение лексемы `DemoVersion`, а поскольку она определена (хотя и без значения), то результат тестирования принимает истинное значение и строка 12 выводит соответствующее сообщение.

В строке 17 определенность лексемы `NT_VERSION` проверяется с помощью директивы `#ifndef`. Поскольку данная лексема определена, возвращается значение `false` и выполнение программы продолжается со строки 20. Именно здесь слово `NT_VERSION` заменяется символом `5`, т.е. компилятор воспринимает эту строку кода в следующем виде:

```
cout << " NT_VERSION defined as: " << 5 << endl;
```

Обратите внимание, что первое слово в сообщении `NT_VERSION` не замещается строкой `5`, поскольку является частью текстовой строки, заключенной в кавычки. Но лексема `NT_VERSION` между операторами вывода замещается; таким образом, компилятор видит вместо нее символ `5`, точно так же, как если бы вы ввели этот символ в выражение вывода.

Наконец, в строке 23 программа проверяет определенность лексемы `WINDOWS_VERSION`. Поскольку эта лексема в программе не определена, возвращается значение `false` и строкой 26 выводится соответствующее сообщение.

Включение файлов и предупреждение ошибок включения

Вы обязательно будете создавать проекты, состоящие из нескольких различных файлов. Традиционно в проектах приложения каждый класс имеет собственный файл заголовка с объявлением класса (обычно такие файлы имеют расширение `.hpp`) и файл источника с кодом выполнения методов класса (обычно с расширением `.cpp`).

Функцию main() программы помещают в свой собственный файл .cpp, а все файлы .cpp компилируются в файлы .obj, которые затем компоновщик связывает в единую программу.

Поскольку программы обычно используют методы из многих классов, основной файл программы будет содержать включения многих файлов заголовков. Кроме того, файлы заголовков часто включают в себя другие файлы заголовков. Например, файл заголовка с объявлением производного класса должен включить файл заголовка базового класса.

Представьте себе, что класс Animal объявляется в файле ANIMAL.hpp. Чтобы объявить класс Dog (который производится от класса Animal), следует в файл DOG.HPP включить файл ANIMAL.hpp, в противном случае класс Dog нельзя будет произвести от класса Animal. Файл заголовка Cat также включает файл ANIMAL.hpp по той же причине.

Если существует метод, который использует оба класса — Cat и Dog, то вы столкнетесь с опасностью двойного включения файла ANIMAL.hpp. Это сгенерирует ошибку в процессе компиляции, поскольку компилятор не позволит дважды объявить класс Animal, даже несмотря на идентичность объявлений. Эту проблему можно решить с помощью директив препроцессора. Код файла заголовка ANIMAL необходимо заключить между следующими директивами:

```
#ifndef ANIMAL_HPP
#define ANIMAL_HPP
...                    // далее следует код файла заголовка
#endif
```

Эта запись означает: если лексема ANIMAL_HPP еще не определена в программе, продолжайте выполнение кода, следующая строка которого определяет эту лексему. Между директивой #define и директивой завершения блока условной компиляции #endif включается содержимое файла заголовка.

Когда ваша программа включает этот файл в первый раз, препроцессор читает первую строку и результат проверки, конечно же, оказывается истинным, т.е. до этого момента лексема еще не была определена как ANIMAL_HPP. Следующая директива препроцессора #define определяет эту лексему, после чего включается код файла.

Если программа включает файл ANIMAL.HPP во второй раз, препроцессор читает первую строку, которая возвращает значение FALSE, поскольку строка ANIMAL.hpp уже была определена. Поэтому управление программой переходит к следующей директиве — #else (в данном случае таковая отсутствует) или #endif (которая находится в конце файла). Следовательно, в этот раз пропускается все содержимое файла и класс дважды не объявляется.

Совершенно не важно реальное имя лексемы (в данном случае ANIMAL_HPP), хотя общепринято использовать имя файла, записанное прописными буквами, а точка (.), отделяющая имя от расширения, заменяется при этом символом подчеркивания. Однако это не закон, а общепринятое соглашение, которое следует рассматривать лишь как рекомендацию.

Никогда не повредит использовать средства защиты от многократного включения. Нередко они способны сэкономить часы работы, потраченные на поиск ошибок и отладку программы.

Макросы

Директиву `#define` можно также использовать для создания макросов. *Макрос* — это лексема, созданная с помощью директивы `#define`. Он принимает параметры подобно обычной функции. Препроцессор заменяет строку подстановки любым заданным параметром. Например, макрокоманду `TWICE` можно определить следующим образом:

```
#define TWICE(x) ( (x) * 2 )
```

А затем в программе можно записать следующую строку:

```
TWICE(4)
```

Целая строка `TWICE(4)` будет удалена, а вместо нее будет стоять значение `8`! Когда препроцессор считывает параметр `4`, он выполняет следующую подстановку: `((4) * 2)`, это выражение затем вычисляется как `4 * 2` и в результате получается число `8`.

Макрос может иметь больше одного параметра, причем каждый параметр в тексте замены может использоваться неоднократно. Вот как можно определить два часто используемых макроса — `MAX` и `MIN`:

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
#define MIN(x,y) ( (x) < (y) ? (x) : (y) )
```

Обратите внимание, что в определении макроса открывающая круглая скобка для списка параметров должна немедленно следовать за именем макроса, т.е. между ними не должно быть никаких пробелов. Препроцессор, в отличие от компилятора, не прощает присутствия ненужных пробелов.

Если записать

```
#define MAX (x,y) ( (x) > (y) ? (x) : (y) )
```

и попытаться использовать макрос `MAX`

```
int x = 5, y = 7, z;
z = MAX(x,y);
```

то промежуточный код будет иметь следующий вид:

```
int x = 5, y = 7, z;
z = (x,y) ( (x) > (y) ? (x) : (y) )(x,y)
```

В этом случае сделана простая текстовая замена, а не вызов макроса, т.е. лексема `MAX` была заменена выражением `(x,y) ( (x) > (y) ? (x) : (y) )`, за которым сохранилась строка `(x,y)`.

Однако после удаления пробела между словом `MAX` и списком параметров `(x,y)` промежуточный код выглядит уже по-другому:

```
int x = 5, y = 7, z;
z =7;
```

Зачем нужны все эти круглые скобки

Вам может показаться странным, что в макросах используется так много круглых скобок. На самом деле препроцессор совсем не требует, чтобы вокруг параметров в строке подстановки ставились круглые скобки, но эти скобки помогают избежать не-

желательных побочных эффектов при передаче макросу сложных значений. Например, если определить MAX как

```
#define MAX(x,y) x > y ? x : y
```

и передать значения 5 и 7, то макрос MAX будет нормально работать. Но если передать более сложные выражения, можно получить неожиданные результаты, как показано в листинге 21.2.

Листинг 21.2. Использование в макросе круглых скобок

```
1:    // Листинг 21.2. Использование в макросе круглых скобок
2:    #include <iostream.h>
3:
4:    #define CUBE(a) ( (a) * (a) * (a) )
5:    #define THREE(a) a * a * a
6:
7:    int main()
8:    {
9:       long x = 5;
10:      long y = CUBE(x);
11:      long z = THREE(x);
12:
13:      cout << "y: " << y << endl;
14:      cout << "z: " << z << endl;
15:
16:      long a = 5, b = 7;
17:      y = CUBE(a+b);
18:      z = THREE(a+b);
19:
20:      cout << "y: " << y << endl;
21:      cout << "z: " << z << endl;
22:    return 0;
23:   }
```

РЕЗУЛЬТАТ

```
y: 125
z: 125
y: 1728
z: 82
```

АНАЛИЗ В строке 4 определяется макрос CUBE с параметром x, который заключается в круглые скобки при каждом его использовании в выражении. В строке 5 определяется макрос THREE, параметр которого используется без круглых скобок.

При первом использовании этих макросов параметру передается значение 5, и оба макроса прекрасно справляются со своей работой. Макрос CUBE(5) преобразуется в выражение ((5) * (5) * (5)), которое при вычислении дает значение 125, а макрос THREE(5) преобразуется в выражение 5 * 5 * 5, которое также возвращает значение 125.

При повторном обращении к этим макросам в строках 16–18 параметру передается выражение 5 + 7. В этом случае макрос CUBE(5+7) преобразуется в следующее выражение:

```
( (5+7) * (5+7) * (5+7) )
```

Оно соответствует выражению

```
( (12) * (12) * (12) )
```

При вычислении этого выражения получаем значение 1728. Однако макрос `THREE(5+7)` преобразуется в выражение иного вида:

```
5 + 7 * 5 + 7 * 5 + 7
```

А поскольку операция умножения имеет более высокий приоритет по сравнению с операцией сложения, то предыдущее выражение эквивалентно следующему:

```
5 + (7 * 5) + (7 * 5) + 7
```

После вычисления произведений в круглых скобках получаем выражение

```
5 + (35) + (35) + 7
```

После суммирования оно возвращает значение 82.

Макросы в сравнении с функциями шаблонов

При работе с макросами в языке C++ можно столкнуться с четырьмя проблемами. Первая состоит в возможных неудобствах при увеличении самого выражения макроса, поскольку любой макрос должен быть определен в одной строке. Безусловно, эту строку можно продлить с помощью символа обратной косой черты (\), но большие макросы сложны для понимания и с ними трудно работать.

Вторая проблема состоит в том, что макросы выполняются путем подстановки их выражений в код программы при каждом вызове. Это означает, что если макрос используется 12 раз, то столько же раз в вашу программу будет вставлено соответствующее выражение (вместо одного раза, как при обращении к обычной функции). Хотя, с другой стороны, подставляемые выражения обычно работают быстрее, чем вызовы функций, поскольку не тратится время на само обращение к функции.

Тот факт, что макросы выполняются путем подстановки выражений в код программы, приводит к третьей проблеме, которая проявляется в том, что макросы отсутствуют в исходном коде программы, используемом компилятором для ее тестирования. Это может существенно затруднить отладку программы.

Однако наиболее существенна последняя проблема: в макросах не поддерживается контроль за соответствием типов данных. Хотя возможность использования в макросе абсолютно любого параметра кажется удобной, этот факт полностью подрывает строгий контроль типов в C++ и является проклятием для программистов на C++. Конечно, существует корректный способ решить и эту проблему — нужно воспользоваться услугами шаблонов, как было показано на занятии 19.

Подставляемые функции

Часто вместо макросов удобно объявить подставляемую функцию. Например, в листинге 21.3 создается функция `CUBE`, которая выполняет ту же работу, что и макрос `CUBE` в листинге 21.2, но в данном случае это делается способом, обеспечивающим контроль за соответствием типов.

Листинг 21.3. Использование подставляемой функции вместо макроса

```
1:    #include <iostream.h>
2:
3:    inline unsigned long Square(unsigned long a) {  return a * a; }
4:    inline unsigned long Cube(unsigned long a)
5:        {  return a * a * a; }
6:    int main()
7:    {
8:       unsigned long x=1 ;
9:       for (;;)
10:      {
11:         cout << "Enter a number (0 to quit): ";
12:         cin >> x;
13:         if (x == 0)
14:            break;
15:         cout << "You entered: " << x;
16:         cout << ".  Square(" << x << "): ";
17:         cout  << Square(x);
18:         cout<< ". Cube(" << x << "): ";
19:         cout << Cube(x) << "." << endl;
20:      }
21:    return 0;
22:   }
```

```
Enter a number (0 to quit): 1
You entered: 1.  Square(1): 1. Cube(1): 1.
Enter a number (0 to quit): 2
You entered: 2.  Square(2): 4. Cube(2): 8.
Enter a number (0 to quit): 3
You entered: 3.  Square(3): 9. Cube(3): 27.
Enter a number (0 to quit): 4
You entered: 4.  Square(4): 16. Cube(4): 64.
Enter a number (0 to quit): 5
You entered: 5.  Square(5): 25. Cube(5): 125.
Enter a number (0 to quit): 6
You entered: 6.  Square(6): 36. Cube(6): 216.
Enter a number (0 to quit): 0
```

Анализ В строках 3 и 4 определяются две подставляемые функции: `Square()` и `Cube()`. Поскольку обе функции объявлены подставляемыми с помощью ключевого слова `inline`, они, как и макросы, будут вставлены в код программы по месту каждого вызова, и никаких временных затрат при выполнении программы, связанных с обращениями к функциям, не возникнет.

Напомним, что подставляемые функции помещаются во время компиляции в программу всюду, где делается обращение к функции (например, в строке 17). А поскольку реального вызова функции никогда не происходит, отсутствуют и временные затраты, связанные с помещением в стек адреса возврата и параметров функции.

В строке 17 вызывается функция Square, а в строке 19 — функция Cube. И вновь-таки, поскольку эти функции подставляемые, реально строка их вызова после компиляции будут выглядеть следующим образом:

```
16:  cout << ".  Square(" << x << "): "  << x * x << ". Cube (" << x << "): " << x * x * x
<< "." << endl;
```

Операции со строками

Препроцессор предоставляет два специальных оператора для управления строками в макросах. Оператор взятия в кавычки (#) берет в кавычки любую строку, которая следует за ним. Оператор конкатенации (##) объединяет две строки в одну.

Оператор взятия в кавычки

Этот оператор берет в кавычки любые следующие за ним символы вплоть до очередно символа пробела. Следовательно, если написать

```
#define WRITESTRING(x) cout << #x
```

и выполнить следующий вызов макроса:

```
WRITESTRING(This is a string);
```

то препроцессор превратит его в такую строку кода:

```
cout << "This is a string";
```

Обратите внимание, что строка This is a string заключается в кавычки, что и требуется для объекта cout.

Конкатенация

Оператор конкатенации позволяет связывать несколько строк в одну. Новая строка на самом деле представляет собой лексему, которую можно использовать как имя класса, имя переменной, смещение в массиве или другом объекте, где может содержаться ряд символов.

Предположим на мгновение, что у вас есть пять функций с такими именами, как *fOnePrint*, *fTwoPrint*, *fThreePrint*, *fFourPrint* и *fFivePrint*. *Теперь можно сделать* следующее объявление:

```
#define fPRINT(x) f ## x ## Print
```

Затем использовать макрос fPRINT(x) с параметром Two, чтобы сгенерировать строку fTwoPrint, и с параметром Three, чтобы сгенерировать строку fThreePrint.

В конце второй недели обучения был разработан класс PartsList. Этот список мог обрабатывать объекты только типа List. Предположим, что этот список зарекомендовал себя хорошей работой и вам захотелось так же хорошо создавать списки животных, автомобилей, компьютеров и т.д.

Один метод решения этой задачи мог бы состоять в создании списков AnimalList, CarList, ComputerList и прочих путем вырезки и вставки кода в нужное место. Однако такой вариант решения быстро превратит вашу жизнь в кошмар, поскольку каждое изменение, вносимое в один список, нужно будет вносить во все другие.

Но, к счастью, существует альтернативное решение — использование макросов и оператора конкатенации. Например, можно определить следующий макрос:

```
#define Listof(Type)  class Type##List \
{  \
public: \
Type##List(){ }  \
private:          \
int itsLength; \
} ;
```

Суть этого примера состоит в том, чтобы включить в одно определение все необходимые методы и данные. Когда нужно будет создать список животных (AnimalList), достаточно записать

```
Listof(Animal)
```

и приведенная выше запись превратится в объявление класса AnimalList. В процессе применения этого подхода не обходится без некоторых проблем, подробно рассмотренных на занятии 19.

Встроенные макросы

Многие компиляторы используют ряд встроенных макросов, таких как __DATE__, __TIME__, __LINE__ и __FILE__. Каждое из этих имен окружено двумя символами подчеркивания, чтобы снизить вероятность того, что они войдут в противоречие с именами, использованными в вашей программе.

Когда препроцессор встречает один из этих макросов, он делает соответствующую подстановку. Вместо лексемы __DATE__ ставится текущая дата. Вместо __TIME__ — текущее время. Лексемы __LINE__ и __FILE__ заменяются номером строки исходного кода и именем файла соответственно. Следует отметить, что эти замены выполняются еще до компиляции. Учтите, что при выполнении программы вместо лексемы __DATE__ будет стоять не текущая дата, а дата компиляции программы. Встроенные макросы часто используют при отладке.

Макрос assert()

Во многих компиляторах предусмотрен макрос assert, который возвращает значение TRUE, если его параметр принимает значение TRUE, и выполняет установленные действия, если его параметр принимает значение FALSE. Многие компиляторы в этом случае прерывают выполнение программы, другие же генерируют исключительную ситуацию (см. занятие 20).

Одна из важных особенностей макроса assert() состоит в том, что препроцессор вообще не замещает его никаким кодом, если не определена лексема DEBUG. Это свойство — большое подспорье в период разработки и при передаче заказчику конечного продукта. Быстродействие программы не страдает и размер исполняемой версии не увеличивается в результате использования этого макроса.

Чтобы не зависеть от конкретной версии компилятора, т.е. от его реакции на макрос assert(), можно написать собственный вариант этого макроса. В листинге 21.4 содержится простой макрос assert() и показано его использование.

Листинг 21.4. Простой макрос assert()

```
1:    // Листинг 21.4. Макрос ASSERT
2:    #define DEBUG
3:    #include <iostream.h>
4:
5:    #ifndef DEBUG
6:       #define ASSERT(x)
7:    #else
8:       #define ASSERT(x) \
9:               if (! (x)) \
10:              { \
11:                 cout << "ERROR!! Assert " << #x << " failed\ n"; \
12:                 cout << " on line " << __LINE__  << "\ n"; \
13:                 cout << " in file " << __FILE__ << "\ n";  \
14:              }
15:   #endif
16:
17:
18:   int main()
19:   {
20:      int x = 5;
21:      cout << "Первый макрос assert: \ n";
22:      ASSERT(x==5);
23:      cout << "\ nВторой макрос assert: \ n";
24:      ASSERT(x != 5);
25:      cout << "\ nВыполнено.\ n";
26:    return 0;
27:   }
```

РЕЗУЛЬТАТ

```
First assert:

Second assert:
ERROR!! Assert x !=5 failed
 on line 24
 in file test1704.cpp
Done.
```

АНАЛИЗ В строке 2 определяется лексема DEBUG. Обычно это делается из командной строки (или в интегрированной среде разработки) во время компиляции, что позволяет управлять этим процессом. В строках 8–14 определяется макрос assert(). Как правило, это делается в файле заголовка ASSERT.hpp, который следует включить во все файлы источников.

В строке 5 проверяется определение лексемы DEBUG. Если она не определена, макрос assert() определяется таким образом, чтобы вообще не создавался никакой код. Если же лексема DEBUG определена, то выполняются строки кода 8–14.

Сам макрос assert() представляет собой цельное выражение, разбитое на семь строк исходного кода. В строке 9 проверяется значение, переданное как параметр. Если передано значение FALSE, выводится сообщение об ошибках (строки 11–13). Если передано значение TRUE, никакие действия не выполняются.

Отладка программы с помощью макроса assert()

Многие ошибки допускаются программистами, поскольку они верят в то, что функция возвратит определенное значение, а указатель будет ссылаться на объект, так как это логически очевидно, и забывают о том, что компилятор не подчиняется человеческой логике, а слепо следует командам и инструкциям, даже если они противоречат всякой логике. Программа может работать самым непонятным образом из-за того, что вы забыли инициализировать указатель при объявлении, и поэтому он ссылается на случайные данные, сохранившиеся в связанных с ним ячейках памяти. Макрос assert()поможет в поиске ошибок такого типа при условии, что вы научитесь правильно использовать этот макрос в своих программах. Каждый раз, когда в программе указатель передается как параметр или в виде возврата функции, имеет смысл проверить, действительно ли этот указатель ссылается на реальное значение. В любом месте программы, если ее выполнение зависит от значения некоторой переменной, с помощью макроса assert() вы сможете убедиться в том, что на это значение можно полагаться.

При этом от частого использования макроса assert() вы не несете никаких убытков, поскольку он автоматически удаляется из программы, если не будет определена лексема DEBUG. Более того, присутствие макроса assert() также обеспечивает хорошее внутреннее документирование программы, поскольку выделяет в коде важные моменты, на которые следует обратить внимание в случае модернизации программы.

Макрос assert() вместо исключений

На прошлом занятии вы узнали, как с помощью исключений можно отслеживать и обрабатывать аварийные ситуации. Важно заметить, что макрос assert() не предназначен для обработки таких исключительных ситуаций, как ввод ошибочных данных, нехватка памяти, невозможность открыть файл и т.д, которые возникают во время выполнения программы. Макрос assert() создан для отслеживания логических и синтаксических ошибок программирования. Следовательно, если макрос assert() срабатывает, это сигнализирует об ошибке непосредственно в коде программы.

Важно понимать, что при передаче программы заказчикам макросы assert() в коде будут удалены. Поэтому если с ошибками выполнения программы удавалось справляться только благодаря макросу assert(), то у заказчика эта программа просто не будет работать.

Распространенной ошибкой является использование макроса assert() для тестирования возвращаемого значения при выполнении операции выделения памяти:

```
Animal *pCat = new Cat;
Assert(pCat);   // неправильное использование макроса
pCat->SomeFunction();
```

Это пример классической ошибки при отладке программы. В данном случае программист пытается с помощью макроса assert() предупредить возникновение исключительной ситуации нехватки свободной памяти. Обычно программист тестирует программу на компьютере с достаточным объемом памяти, поэтому макрос assert()в этом месте программы никогда не сработает. У заказчика может быть устаревшая версия компьютера, поэтому, когда программа доходит до этой точки, обращение к оператору

new терпит крах и программа возвращает NULL (пустой указатель). Однако макроса assert() больше нет в коде, и некому сообщить пользователю о том, что указатель ссылается на NULL. Поэтому, как только дойдет очередь до выражения pCat->SomeFunction(), программа дает сбой.

Возвращение значения NULL при выделения памяти — это не ошибка программирования, а исключительная ситуация. Чтобы программа смогла с честью выйти из этой ситуации, необходимо использовать исключение. Помните, что макрос assert() полностью удаляется из программы, если лексема DEBUG не определена. (Исключения были подробно описаны на занятии 20.)

Побочные эффекты

Нередко случается так, что ошибка проявляется только после удаления экземпляров макроса assert(). Почти всегда это происходит из-за того, что программа попадает в зависимость от побочных эффектов, вызванных выполнением макроса assert() или другими частями кода, используемыми только для отладки. Например, если записать

```
ASSERT (x = 5)
```

при том, что имелась в виду проверка x == 5, вы тем самым создадите чрезвычайно противную ошибку.

Предположим, что как раз до выполнения макроса assert() вызывалась функция, *которая установила переменную* x *равной* 0. *Используя данный макрос, вы полагали, что* выполняете проверку равенства переменной x значению 5. На самом же деле вы устанавливаете значение x равным 5. Тем не менее эта ложная проверка возвращает значение TRUE, поскольку выражение x = 5 не только устанавливает переменную x равной 5, но одновременно и возвращает значение 5, а так как 5 не равно нулю, то это значение расценивается как истинное.

Во время отладки программы макрос assert() не выполняет проверку равенства переменной x значению 5, а присваивает ей это значение, поэтому программа работает прекрасно. Вы готовы передать ее заказчику и отключаете отладку. Теперь макрос assert() удаляется из кода и переменная x не устанавливается равной 5. Но поскольку в результате ошибки в функции переменная x устанавливается равной 0, программа дает сбой.

Рассерженный заказчик возвращает программу, вы восстанавливаете средства отладки, но не тут-то было! Ошибка исчезла. Такие вещи довольно забавно наблюдать со стороны, но не переживать самим, поэтому остерегайтесь побочных эффектов при использовании средств отладки. Если вы видите, что ошибка появляется только при отключении средств отладки, внимательно просмотрите команды отладки с учетом проявления возможных побочных эффектов.

Инварианты класса

Для многих классов существует ряд условий, которые всегда должны выполняться при завершении работы с функцией-членом класса. Эти обязательные условия выполнения класса называются *инвариантами* класса. Например, обязательными могут быть следующие условия: объект CIRCLE никогда не должен иметь нулевой радиус или объект ANIMAL всегда должен иметь возраст больше нуля и меньше 100.

Может быть весьма полезным объявление метода Invariants(), который возвращает значение TRUE только в том случае, если каждое из этих условий является истинным. Затем можно вставить макрос ASSERT(Invariants()) в начале и в конце каждого

метода класса. В качестве исключения следует помнить, что метод Invariants() не возвращает TRUE до вызова конструктора и после выполнения деструктора. Использование метода Invariants() для обычного класса показано в листинге 21.5.

Листинг 21.5. Использование метода Invariants()

```
1:    #define DEBUG
2:    #define SHOW_INVARIANTS
3:    #include <iostream.h>
4:    #include <string.h>
5:
6:    #ifndef DEBUG
7:    #define ASSERT(x)
8:    #else
9:    #define ASSERT(x) \
10:                if (! (x)) \
11:                { \
12:                   cout << "ERROR!! Assert " << #x << " failed\ n"; \
13:                   cout << " on line " << __LINE__  << "\ n"; \
14:                   cout << " in file " << __FILE__ << "\ n";  \
15:                }
16:   #endif
17:
18:
19:   const int FALSE = 0;
20:   const int TRUE = 1;
21:   typedef int bool;
22:
23:
24:   class String
25:   {
26:      public:
27:         // конструкторы
28:         String();
29:         String(const char *const);
30:         String(const String &);
31:         ~String();
32:
33:         char & operator[](int offset);
34:         char operator[](int offset) const;
35:
36:         String & operator= (const String &);
37:         int GetLen()const { return itsLen; }
38:         const char * GetString() const { return itsString; }
39:         bool Invariants() const;
40:
41:      private:
42:         String (int);          // закрытый конструктор
43:         char * itsString;
44:        // беззнаковая целочисленная переменная itsLen;
```

```
45:         int itsLen;
46:    } ;
47:
48:    // стандартный конструктор создает строку нулевой длины
49:    String::String()
50:    {
51:       itsString = new char[1];
52:       itsString[0] = '\ 0';
53:       itsLen=0;
54:       ASSERT(Invariants());
55:    }
56:
57:    // закрытый (вспомогательный) конструктор, используется
58:    // методами класса только для создания новой строки
59:    // требуемого размера.  При этом вставляется концевой нулевой символ.
60:    String::String(int len)
61:    {
62:       itsString = new char[len+1];
63:       for (int i = 0; i<=len; i++)
64:          itsString[i] = '\ 0';
65:       itsLen=len;
66:       ASSERT(Invariants());
67:    }
68:
69:    // Преобразует массив символов к типу String
70:    String::String(const char * const cString)
71:    {
72:       itsLen = strlen(cString);
73:       itsString = new char[itsLen+1];
74:       for (int i = 0; i<itsLen; i++)
75:          itsString[i] = cString[i];
76:       itsString[itsLen]='\ 0';
77:       ASSERT(Invariants());
78:    }
79:
80:    // конструктор-копировщик
81:    String::String (const String & rhs)
82:    {
83:       itsLen=rhs.GetLen();
84:       itsString = new char[itsLen+1];
85:       for (int i = 0; i<itsLen;i++)
86:          itsString[i] = rhs[i];
87:       itsString[itsLen] = '\ 0';
88:       ASSERT(Invariants());
89:    }
90:
91:    // деструктор, освобождает выделенную память
92:    String::~String ()
93:    {
94:       ASSERT(Invariants());
```

```
95:       delete [] itsString;
96:       itsLen = 0;
97:    }
98:
99:    // оператор выполняет сравнение, освобождает занятую
100:    // память, а затем копирует строку и ее размер
101:    String& String::operator=(const String & rhs)
102:    {
103:       ASSERT(Invariants());
104:       if (this == &rhs)
105:          return *this;
106:       delete [] itsString;
107:       itsLen=rhs.GetLen();
108:       itsString = new char[itsLen+1];
109:       for (int i = 0; i<itsLen;i++)
110:          itsString[i] = rhs[i];
111:       itsString[itsLen] = '\ 0';
112:       ASSERT(Invariants());
113:       return *this;
114:    }
115:
116: // неконстантный оператор индексирования
117:    char & String::operator[](int offset)
118:  {
119:     ASSERT(Invariants());
120:     if (offset > itsLen)
121:     {
122:       ASSERT(Invariants());
123:       return itsString[itsLen-1];
124:     }
125:      else
126:     {
127:        ASSERT(Invariants());
128:       return itsString[offset];
129:     }
130:  }
131:    // константный оператор индексирования
132:    char String::operator[](int offset) const
133:    {
134:       ASSERT(Invariants());
135:       char retVal;
136:       if (offset > itsLen)
137:          retVal = itsString[itsLen-1];
138:       else
139:          retVal = itsString[offset];
140:       ASSERT(Invariants());
141:       return retVal;
142:    }
143:    bool String::Invariants() const
144:    {
```

```
145:    #ifdef SHOW_INVARIANTS
146:       cout << "Invariants Tested";
147:    #endif
148:        return ( (itsLen && itsString) ||
149:          (!itsLen && !itsString) );
150:     }
151:     class Animal
152:     {
153:     public:
154:        Animal():itsAge(1),itsName("John Q. Animal")
155:           { ASSERT(Invariants());}
156:         Animal(int, const String&);
157:        ~Animal(){ }
158:        int GetAge() {   ASSERT(Invariants()); return itsAge;}
159:        void SetAge(int Age)
160:        {
161:             ASSERT(Invariants());
162:             itsAge = Age;
163:             ASSERT(Invariants());
164:        }
165:         String& GetName()
166:         {
167:               ASSERT(Invariants());
168:               return itsName;
169:         }
170:         void SetName(const String& name)
171:               {
172:               ASSERT(Invariants());
173:               itsName = name;
174:               ASSERT(Invariants());
175:         }
176:         bool Invariants();
177:      private:
178:         int itsAge;
179:         String itsName;
180:      } ;
181:
182:      Animal::Animal(int age, const String& name):
183:      itsAge(age),
184:      itsName(name)
185:      {
186:         ASSERT(Invariants());
187:      }
188:
189:      bool Animal::Invariants()
190:      {
191:      #ifdef SHOW_INVARIANTS
192:         cout << "Invariants Tested";
193:      #endif
194:         return (itsAge > 0 && itsName.GetLen());
```

```
195:     }
196:
197:     int main()
198:     {
199:        Animal sparky(5, "Sparky");
200:        cout << "\ n" << sparky.GetName().GetString() << " is ";
201:        cout << sparky.GetAge() << " years old. ";
202:        sparky.SetAge(8);
203:        cout << "\ n" << sparky.GetName().GetString() << " is ";
204:        cout << sparky.GetAge() << " years old. ";
205:        return 0;
206:     }
```

Результат

```
String OK  String OK  String OK  String OK  String OK String OK String OK
String OK  String OK  Animal OK  String OK  Animal OK
Sparky is  Animal OK 5 years old. Animal OK  Animal OK
Animal OK  Sparky is  Animal OK 8 years old. String OK
```

Анализ

В строках 9–15 определяется макрос `assert()`. Если лексема `DEBUG` определена и макрос `assert()` возвратит в результате операции сравнения значение `FALSE`, будет выведено сообщение об ошибке.

В строке 39 объявляется функция-член `Invariants()` класса `String`, а ее определение занимает строки 143–150. Конструктор объявляется в строках 49–55, а в строке 54, после того как объект полностью построен, вызывается функция-член `Invariants()`, чтобы подтвердить правомочность этой конструкции.

Этот алгоритм повторен для других конструкторов, а для деструктора функция-член `Invariants()` вызывается только перед тем, как удалить объект. Остальные методы класса вызывают функцию `Invariants()` перед выполнением любого действия, а затем еще раз перед возвратом из функции. В этом проявляется отличие функций-членов от конструкторов и деструкторов: функции-члены всегда работают с реальными объектами и должны оставить их таковыми по завершению выполнения функции.

В строке 176 класс `Animal` объявляет собственный метод `Invariants()`, выполняемый в строках 189–195. Обратите внимание на строки 155, 158, 161 и 163: подставляемые функции также могут вызывать метод `Invariants()`.

Печать промежуточных значений

Не исключено, что в дополнение к возможности с помощью макроса `assert()` убедиться в истинности некоторого тестируемого выражения вы захотите вывести на экран текущие значения указателей, переменных и строк. Это может быть полезно для проверки ваших предположений насчет некоторых аспектов работы программы, а также при поиске ошибок в циклах. Реализация этой идеи показана в листинге 21.6.

Листинг 21.6. Вывод значений в режиме отладки

```
1:    // Листинг 21.6. Вывод значений в режиме отладки
2:    #include <iostream.h>
3:    #define DEBUG
4:
```

```
5:    #ifndef DEBUG
6:    #define PRINT(x)
7:    #else
8:    #define PRINT(x) \
9:       cout << #x << ":\ t" << x << endl;
10:   #endif
11:
12:   enum bool {  FALSE, TRUE }  ;
13:
14:   int main()
15:   {
16:      int x = 5;
17:      long y = 738981;
18:      PRINT(x);
19:      for (int i = 0; i < x; i++)
20:      {
21:         PRINT(i);
22:      }
23:
24:      PRINT (y);
25:      PRINT("Hi.");
26:      int *px = &x;
27:      PRINT(px);
28:      PRINT (*px);
29:    return 0;
30:   }
```

```
x:      5
i:      0
i:      1
i:      2
i:      3
i:      4
y:      73898
"Hi.":  Hi.
px:       0x2100
*px:    5
```

Макрос `PRINT(x)` (строки 5—10) реализует вывод текущего значения переданного параметра. Обратите внимание, что сначала объекту `cout` передается сам параметр, взятый в кавычки, т.е., если вы передадите параметр `x`, объект `cout` примет `"x"`.

Затем объект `cout` принимает заключенную в кавычки строку `":\ t "`, которая обеспечивает печать двоеточия и табуляции. После этого объект `cout` принимает значение параметра `(x)`, а объект `endl` выполняет переход на новую строку и очищает буфер.

Обратите внимание, что у вас вместо значения `0x2100` может быть выведено другое число.

Уровни отладки

В больших и сложных проектах вам, возможно, понадобится больше рычагов управления для отлаживания программы, чем просто подключение и отключение режима отладки (путем определения лексемы DEBUG). Вы можете определять уровни отладки и выполнять тестирование для этих уровней, принимая решение о том, какие макрокоманды использовать, а какие – удалить.

Чтобы определить *уровень отладки*, достаточно после выражения #define DEBUG указать номер. Хотя число уровней может быть любым, обычная система должна иметь четыре уровня: HIGH (высокий), MEDIUM (средний), LOW (низкий) и NONE (никакой). В листинге 21.7 показано, как это можно сделать, на примере классов String и Animal из листинга 21.5.

Листинг 21.7. Уровни отладки

```
1:    enum LEVEL {  NONE, LOW, MEDIUM, HIGH } ;
2:    const int FALSE = 0;
3:    const int TRUE = 1;
4:    typedef int bool;
5:
6:     #define DEBUGLEVEL HIGH
7:
8:     #include <iostream.h>
9:     #include <string.h>
10:
11:     #if DEBUGLEVEL < LOW  // должен быть средний или высокий
12:     #define ASSERT(x)
13:     #else
14:     #define ASSERT(x) \
15:         if (! (x)) \
16:         {  \
17:            cout << "ERROR!! Assert " << #x << " failed\ n"; \
18:            cout << " on line " << __LINE__  << "\ n"; \
19:            cout << " in file " << __FILE__ << "\ n";  \
20:         }
21:     #endif
22:
23:     #if DEBUGLEVEL < MEDIUM
24:     #define EVAL(x)
25:     #else
26:     #define EVAL(x) \
27:       cout << #x << ":\ t" << x << endl;
28:     #endif
29:
30:    #if DEBUGLEVEL < HIGH
31:     #define PRINT(x)
32:     #else
33:     #define PRINT(x) \
34:       cout << x << endl;
35:     #endif
```

```
36:
37:
38:     class String
39:     {
40:        public:
41:           // конструкторы
42:           String();
43:           String(const char *const);
44:           String(const String &);
45:           ~String();
46:
47:           char & operator[](int offset);
48:           char operator[](int offset) const;
49:
50:           String & operator= (const String &);
51:           int GetLen()const {  return itsLen; }
52:           const char * GetString() const
53:            {  return itsString; }
54:           bool Invariants() const;
55:
56:        private:
57:           String (int);          // закрытый конструктор
58:           char * itsString;
59:           unsigned short itsLen;
60:     } ;
61:
62:    // стандартный конструктор создает строку нулевой длины
63:    String::String()
64:    {
65:       itsString = new char[1];
66:       itsString[0] = '\ 0';
67:       itsLen=0;
68:       ASSERT(Invariants());
69:    }
70:
71:    // закрытый (вспомогательный) конструктор, используемый
72:    // методами класса только для создания новой строки
73:    // требуемого размера.  Заполняется символом Null.
74:    String::String(int len)
75:    {
76:       itsString = new char[len+1];
77:       for (int i = 0; i<=len; i++)
78:          itsString[i] = '\ 0';
79:       itsLen=len;
80:       ASSERT(Invariants());
81:    }
82:
83:    // Преобразует массив символов к типу String
84:    String::String(const char * const cString)
```

```
85:    {
86:       itsLen = strlen(cString);
87:       itsString = new char[itsLen+1];
88:       for (int i = 0; i<itsLen; i++)
89:          itsString[i] = cString[i];
90:       itsString[itsLen]='\ 0';
91:       ASSERT(Invariants());
92:    }
93:
94:    // конструктор-копировщик
95:    String::String (const String & rhs)
96:    {
97:       itsLen=rhs.GetLen();
98:       itsString = new char[itsLen+1];
99:       for (int i = 0; i<itsLen;i++)
100:          itsString[i] = rhs[i];
101:       itsString[itsLen] = '\ 0';
102:       ASSERT(Invariants());
103:    }
104:
105:    // деструктор освобождает выделенную память
106:    String::~String ()
107:    {
108:       ASSERT(Invariants());
109:       delete [] itsString;
110:       itsLen = 0;
111:    }
112:
113:    // оператор выполняет сравнение, освобождает занятую память
114:    // затем копирует строку и ее размер
115:    String& String::operator=(const String & rhs)
116:    {
117:       ASSERT(Invariants());
118:       if (this == &rhs)
119:          return *this;
120:       delete [] itsString;
121:       itsLen=rhs.GetLen();
122:       itsString = new char[itsLen+1];
123:       for (int i = 0; i<itsLen;i++)
124:          itsString[i] = rhs[i];
125:       itsString[itsLen] = '\ 0';
126:       ASSERT(Invariants());
127:       return *this;
128:    }
129:
130: // неконстантный оператор индексирования
131:    char & String::operator[](int offset)
132:    {
133:       ASSERT(Invariants());
```

```
134:     if (offset > itsLen)
135:     {
136:       ASSERT(Invariants());
137:       return itsString[itsLen-1];
138:     }
139:      else
140:     {
141:        ASSERT(Invariants());
142:       return itsString[offset];
143:     }
144: }
145:    // константный оператор индексирования
146:    char String::operator[](int offset) const
147:    {
148:       ASSERT(Invariants());
149:       char retVal;
150:       if (offset > itsLen)
151:          retVal = itsString[itsLen-1];
152:       else
153:          retVal = itsString[offset];
154:       ASSERT(Invariants());
155:       return retVal;
156:    }
157:
158:     bool String::Invariants() const
159:     {
160:         PRINT("(String Invariants Checked)");
161:         return ( (bool) (itsLen && itsString) ||
162:             (!itsLen && !itsString) );
163:     }
164:
165:     class Animal
166:     {
167:     public:
168:        Animal():itsAge(1),itsName("John Q. Animal")
169:            { ASSERT(Invariants());}
170:
171:        Animal(int, const String&);
172:        ~Animal(){ }
173:
174:        int GetAge()
175:            {
176:                ASSERT(Invariants());
177:                return itsAge;
178:            }
179:
180:        void SetAge(int Age)
181:            {
182:                ASSERT(Invariants());
```

```
183:              itsAge = Age;
184:              ASSERT(Invariants());
185:          }
186:       String& GetName()
187:          {
188:              ASSERT(Invariants());
189:              return itsName;
190:          }
191:
192:       void SetName(const String& name)
193:          {
194:              ASSERT(Invariants());
195:              itsName = name;
196:              ASSERT(Invariants());
197:          }
198:
199:       bool Invariants();
200:    private:
201:       int itsAge;
202:       String itsName;
203:    } ;
204:
205:     Animal::Animal(int age, const String& name):
206:     itsAge(age),
207:     itsName(name)
208:     {
209:        ASSERT(Invariants());
210:     }
211:
212:    bool Animal::Invariants()
213:    {
214:       PRINT("(Animal Invariants Checked)");
215:       return (itsAge > 0 && itsName.GetLen());
216:    }
217:
218:    int main()
219:    {
220:       const int AGE = 5;
221:       EVAL(AGE);
222:       Animal sparky(AGE,"Sparky");
223:       cout << "\ n" << sparky.GetName().GetString();
224:       cout << " is ";
225:       cout << sparky.GetAge() << " years old.";
226:       sparky.SetAge(8);
227:       cout << "\ n" << sparky.GetName().GetString();
228:       cout << " is ";
229:       cout << sparky.GetAge() << " years old.";
230:       return 0;
231:    }
```

РЕЗУЛЬТАТ

```
AGE:     5
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)
 (String Invariants Checked)

Sparky is (Animal Invariants Checked)
5 years old. (Animal Invariants Checked)
 (Animal Invariants Checked)
 (Animal Invariants Checked)

Sparky is (Animal Invariants Checked)
8 years old. (String Invariants Checked)
 (String Invariants Checked)

// run again with DEBUG = MEDIUM

AGE:     5
Sparky is 5 years old.
Sparky is 8 years old.
```

АНАЛИЗ

В строках 11–21 макрос `assert()` определяется таким образом, чтобы вообще не создавался никакой код, если уровень отладки `DEBUGLEVEL` меньше, чем `LOW` (т.е. `DEBUGLEVEL` установлен равным значению `NONE`). Если же отладка разрешена, то и макрос `assert()` будет работать (строки 14–21). В строке 24 макрос `EVAL` отключается, если уровень отладки `DEBUGLEVEL` меньше, чем `MEDIUM`; иными словами, если уровень отладки `DEBUGLEVEL` установлен равным значению `NONE` или `LOW`, макрос `EVAL` не работает.

Наконец, в строках 30–35, макрофункция `PRINT` объявляется "бездействующей", если уровень отладки `DEBUGLEVEL` меньше, чем `HIGH`. Макрофункция `PRINT` используется только в том случае, если уровень отладки `DEBUGLEVEL` установлен равным значению `HIGH`, т.е. этот макрос можно удалить, установив уровень отладки `DEBUGLEVEL` равным значению `MEDIUM`, и при этом поддерживать использование макросов `EVAL` и `assert()`.

Макрос `PRINT` используется внутри методов `Invariants()` для печати информативного сообщения. Макрос `EVAL` используется в строке 221, чтобы отобразить текущее значение целочисленной константы `AGE`.

Рекомендуется

Используйте ПРОПИСНЫЕ буквы для имен макросов. Это широко распространенное соглашение, поэтому его несоблюдение может ввести в заблуждение других программистов.

Заключайте все аргументы макросов в круглые скобки.

Не рекомендуется

Не изменяйте и не присваивайте значения переменных в макросах отладки, поскольку это чревато появлением побочных эффектов.

Операции с битами данных

Иногда, чтобы отслеживать состояние объектов, бывает удобно устанавливать для них флаги. (Например, с помощью флагов можно проверить, был ли объект инициализирован, вызывался ли для него определенный метод и пр., а также связать вывод предупреждающих сообщений с флагом объекта `AlarmState`.)

Для флагов можно использовать переменные типа `Boolean`, но если у вас много признаков и для вас важно экономить ресурсы компьютера, удобнее для установки флагов использовать отдельные биты значения в двоичном формате.

Каждый байт имеет восемь битов, поэтому четырехбайтовая переменная типа `long` может представлять 32 отдельных флага. Если значение бита равно `1`, то говорят, что флаг установлен, а если `0` — то сброшен. Другими словами, чтобы установить флаг, нужно определенному биту переменной присвоить значение `1`, а чтобы сбросить флаг — значение `0`. Устанавливать и сбрасывать флаги можно, изменяя значения переменной типа `long`, но такой подход нерационален и может ввести в заблуждение.

В приложении В содержится ценная дополнительная информация об операциях над двоичными и шестнадцатеричными числами.

В языке C++ предусмотрены побитовые операторы для работы с битами данных. Они схожи, но в то же время отличаются от логических операторов, поэтому многие начинающие программисты их путают. Побитовые операторы представлены в табл. 21.1.

Таблица 21.1. Побитовые операторы

Символ	*Оператор*
&	AND (логическое И)
|	OR (логическое ИЛИ)
^	Исключающее ИЛИ
~	Дополнение до единицы

Оператор И (AND)

Для обозначения оператора побитового И (`&`) используется одиночный амперсант, а оператор логического И обозначается двумя амперсантами. При выполнении операции побитового И с двумя битами результат равен 1, если оба бита равны 1, и 0, если хотя бы один бит (или оба сразу) равен 0.

Оператор ИЛИ (OR)

Вторым побитовым оператором является ИЛИ (`|`). И опять-таки для его обозначения используется одиночный символ вертикальной черты, в отличие от логического ИЛИ, обозначаемого двумя символами вертикальной черты. При выполнении операции побитового ИЛИ с двумя битами результат равен 1, если хотя бы один бит (или оба сразу) равен 1.

Оператор исключающего ИЛИ (OR)

Третий побитовый оператор — исключающее ИЛИ (^). При выполнении операции исключающего ИЛИ с двумя битами результат равен 1, если эти два разряда различны.

Оператор дополнения до единицы

Оператор дополнения до единицы (~) сбрасывает каждый установленный бит и устанавливает каждый сброшенный бит в числе. Если текущее значение равно 1010 0011, то дополнение его до единицы будет иметь вид 0101 1100.

Установка битов

Если вы хотите установить или сбросить конкретный флаг, следует использовать операцию маскирования. Если в программе для установки флагов используется 4-байтовая переменная и нужно установить флаг, связанный с восьмым битом этой переменной, следует выполнить операцию побитового ИЛИ для этой переменной и числа 128. Почему 128? Потому что 128 – это 1000 0000 в двоичной системе счисления, таким образом, можно сказать, что число 128 определяет значение восьмого разряда. Независимо от текущего значения этого разряда в 4-байтовой переменной (установлен он или сброшен), при выполнении операции ИЛИ с числом 128 этот бит будет установлен, а все остальные биты сохранят прежние значения. Предположим, что текущее значение этой 4-байтовой переменной в двоичном формате равно 1010 0110 0010 0110. Применение к ней операции ИЛИ с числом 128 выглядит следующим образом:

```
9 8765 4321
1010 0110 0010 0110   // 8-й бит сброшен
|   0000 0000 1000 0000   // 128

- - - - - - - - - -
1010 0110 1010 0110   // 8-й бит установлен
```

Хочется обратить ваше внимание на некоторые вещи. Во-первых, как правило, биты считаются справа налево. Во-вторых, значение 128 содержит все нули, за исключением восьмого бита, т.е. того разряда, который вы хотите установить. В-третьих, в исходном числе 1010 0110 0010 0110 операцией ИЛИ изменяется только восьмой бит. Если бы он в этом значении был установлен еще до выполнения операции ИЛИ, то значение вообще не изменилось бы.

Сброс битов

Если нужно сбросить восьмой бит, можно использовать побитовую операцию И с дополнением числа 128 до единицы. Дополнение числа 128 до – это называется такое число, которое получается, если взять в двоичном представлении число 128 (1000 0000), а затем установить в нем каждый сброшенный и сбросить каждый установленный бит (0111 1111). При выполнении побитовой операции И с этими числами исходное число не изменяется, за исключением восьмого разряда, который сбрасывается в нуль.

```
1010 0110 1010 0110  // 8-й бит установлен
& 1111 1111 0111 1111  // ~128 (дополнение до единицы числа 128)

- - - - - - - - - -
1010 0110 0010 0110  // 8-й бит сброшен
```

Чтобы до конца понять этот метод решения, сделайте самостоятельно все математические операции. Каждый раз, когда оба бита равны 1, запишите результат равным 1. Если какой-нибудь бит равен 0, запишите в ответе 0. Сравните ответ с исходным числом. Оно должно остаться без изменений, за исключением восьмого бита, который в результате этой операции побитового И будет сброшен.

Инверсия битов

Наконец, если вы хотите инвертировать восьмой бит независимо от его предыдущего состояния, используйте операцию исключающего ИЛИ для этого числа и числа 128. Итак:

```
1010 0110 1010 0110  // число
^ 0000 0000 1000 0000  // 128

_ _ _ _ _ _ _ _ _ _ _ _
1010 0110 0010 0110  // 8-й бит инвертирован
^ 0000 0000 1000 0000  // 128

_ _ _ _ _ _ _ _ _ _ _ _
1010 0110 1010 0110  // 8-й бит инвертирован снова
```

Рекомендуется

Используйте маски и оператор ИЛИ для установки битов.

Используйте маски и оператор И для сброса битов.

Рекомендуется

Используйте маски и оператор исключающего ИЛИ для инвертирования битов.

Битовые поля

При некоторых обстоятельствах, когда на счету каждый байт, экономия шести или восьми байтов в классе может иметь существенные последствия. Если в классе или структуре вместо набора логических переменных (типа `Boolean`) или переменных, которые могут иметь только очень небольшое число возможных значений, использовать битовые поля, можно сэкономить некоторый объем памяти.

Среди стандартных типов данных C++ меньше всего памяти требуют переменные типа `char`: длина переменной составляет всего один байт. Часто для создания битовых полей используются переменные типа `int`, для которых требуется два или чаще четыре байта. В битовом поле, основанном на переменной типа `char`, можно хранить восемь двоичных значений, а в переменной типа `long` - 32 таких значения.

Так как же работают битовые поля? Им присваиваются имена и организуется доступ точно таким же способом, как к любому члену класса. Они всегда объявляются с использованием беззнакового типа `int`. После имени битового поля ставится двоеточие и число. Число указывает компилятору, сколько битов будет использовано для установки одного значения. Так, если записать число 1, то с помощью одного бита можно будет присваивать только значения 0 или 1. Если записать число 2, то с помощью двух битов можно будет представлять четыре значения: 0, 1, 2 или 3. Поле из трех битов может представлять восемь значений и т.д. Обзор двоичных чисел приведен в приложении В. Использование битовых полей иллюстрируется в листинге 21.8.

Листинг 21.8. Использование битовых полей

```
1:      #include <iostream.h>
2:      #include <string.h>
3:
4:      enum STATUS {  FullTime, PartTime }  ;
5:      enum GRADLEVEL {  UnderGrad, Grad }  ;
6:      enum HOUSING {  Dorm, OffCampus } ;
7:      enum FOODPLAN {  OneMeal, AllMeals, WeekEnds, NoMeals } ;
8:
9:      class student
10:      {
11:      public:
12:      student():
13:          myStatus(FullTime),
14:          myGradLevel(UnderGrad),
15:          myHousing(Dorm),
16:          myFoodPlan(NoMeals)
17:          { }
18:          ~student(){ }
19:          STATUS GetStatus();
20:          void SetStatus(STATUS);
21:          unsigned GetPlan() {  return myFoodPlan; }
22:
23:      private:
24:         unsigned myStatus : 1;
25:         unsigned myGradLevel: 1;
26:         unsigned myHousing : 1;
27:         unsigned myFoodPlan : 2;
28:      } ;
29:
30:      STATUS student::GetStatus()
31:      {
32:         if (myStatus)
33:            return FullTime;
34:         else
35:            return PartTime;
36:      }
37:      void student::SetStatus(STATUS theStatus)
38:      {
39:         myStatus = theStatus;
40:      }
41:
42:
43:      int main()
44:      {
45:         student Jim;
46:
47:         if (Jim.GetStatus()== PartTime)
48:            cout << "Jim is part time" << endl;
```

```
49:          else
50:             cout << "Jim is full time" << endl;
51:
52:          Jim.SetStatus(PartTime);
53:
54:          if (Jim.GetStatus())
55:             cout << "Jim is part time" << endl;
56:          else
57:             cout << "Jim is full time" << endl;
58:
59:          cout << "Jim is on the " ;
60:
61:          char Plan[80];
62:          switch (Jim.GetPlan())
63:          {
64:             case  OneMeal: strcpy(Plan, "One meal"); break;
65:             case  AllMeals: strcpy(Plan, "All meals"); break;
66:             case WeekEnds: strcpy(Plan, "Weekend meals"); break;
67:             case NoMeals: strcpy(Plan, "No Meals");break;
68:             default : cout << "Something bad went wrong!\ n"; break;
69:          }
70:          cout << Plan << " food plan. " << endl;
71:       return 0;
72:      }
```

РЕЗУЛЬТАТ

```
Jim is part time
Jim is full time
Jim is on the No Meals food plan.
```

АНАЛИЗ

Строки 4–7 содержат определение нескольких перечислений. Они используются для определения значения битовых полей внутри класса `student`.

В строках 9–28 объявляется класс `student`. Несмотря на тривиальность, он интересен тем, что все его данные упакованы в пяти битах. Первый бит определяет, является ли данный студент представителем очной (`full time`) или заочной (`part time`) формы обучения. Второй — получил ли этот студент степень бакалавра (`UnderGrad`). Третий — проживает ли студент в общежитии. И последние два бита определяют, какой из четырех возможных вариантов питания в студенческой столовой выбран студентом.

Методы класса не отличаются ничем особенным от методов любого другого класса, т.е. на них никоим образом не повлиял тот факт, что они написаны для битовых полей, а не для обычных целочисленных значений или перечислений.

Функция-член `GetStatus()` считывает значение бита и возвращает константу перечисления, но это не обязательное решение. С таким же успехом можно было бы написать вариант, непосредственно возвращающий значение битового поля. Компилятор сам сможет преобразовать битовое значение в константу.

Чтобы убедиться в этом, замените выполнение функции `GetStatus()` следующим кодом:

```
STATUS student::GetStatus()
{
return myStatus;
}
```

При этом в работе программы не произойдет никаких изменений. Вариант, приведенный в листинге, — это дань ясности при чтении кода, а на результат работы компилятора это изменение никак не повлияет.

Обратите внимание на то, что строка 47 должна проверить статус студента (`full time` или `part time`), а затем вывести соответствующее сообщение. Попробуем выполнить то же самое по-другому:

```
cout << "Jim is " << Jim.GetStatus() << endl;
```

В результате выполнения этого выражения будет выведено следующее сообщение:

```
Jim is 0
```

Компилятор не сможет перевести константу перечисления `PartTime` в соответствующую строку текста.

В строке 62 программы определяется вариант питания студента и для каждого возможного значения соответствующее сообщение помещается в буфер, а затем выводится в строке 70. Опять-таки заметим, что конструкцию с оператором `switch` можно было бы написать следующим образом:

```
case  0: strcpy(Plan,"One meal"); break;
case  1: strcpy(Plan,"All meals"); break;
case  2: strcpy(Plan,"Weekend meals"); break;
case  3: strcpy(Plan,"No Meals");break;
```

Самое важное в использовании битовых полей то, что клиент класса не должен беспокоиться насчет способа хранения данных. Поскольку битовые поля относятся к скрытым данным, вы можете свободно изменить их впоследствии, при этом никаких изменений интерфейса не потребуется.

Стиль программирования

Как уже упоминалось в этой книге, в программе важно придерживаться одного принятого стиля, хотя в целом не имеет значения, какому именно стилю вы отдаете предпочтение. Соблюдение определенного стиля существенно облегчает чтение и анализ программы.

Следующие рекомендации совершенно ни к чему вас не обязывают. Они основаны на ряд принципов, которых я придерживаюсь при работе над проектами и которые нахожу полезными. Вы можете выработать собственные правила, но приведенные ниже помогут выделить основные моменты, на которые следует обратить внимание.

Хотя в жизни чрезмерная пунктуальность нас раздражает, тем не менее строгое соблюдение стиля при написании программы поможет вам и вашим коллегам эффективнее использовать и модернизировать однажды написанный код. Постарайтесь выработать собственный стиль программирования и затем относитесь к нему так, как к уголовному кодексу, нарушение которого карается законом.

Отступы

Отступ табуляции должен составлять четыре пробела. Убедитесь в том, что ваш редактор преобразует каждую табуляцию в четыре пробела.

Фигурные скобки

Способ выравнивания фигурных скобок вызывает, возможно, самые бурные споры между программистами C++ и C. Я лично придерживаюсь следующих правил:

- пара фигурных скобок должна быть выровнена по вертикали;
- фигурные скобки первого уровня в определении или объявлении должны быть выровнены по левому полю. Все строки блока объявления или определения записываются с отступом. Все вложенные пары фигурных скобок должны быть выровнены по одной линии со строкой программы, за которой начинается этот блок;
- строки блока никогда не должны находиться на одной линии с фигурными скобками, обрамляющими этот блок, например:

```
if (condition==true)
{
    j = k;
    SomeFunction();
}
m++;
```

Длинные строки

Удерживайте ширину строк в таких пределах, чтобы они помещались на экране. Код, который "убегает" вправо, можно легко пропустить, а горизонтальная прокрутка всегда раздражает. При разбиении строки для следующих строк делайте отступы. Старайтесь разбивать строку, следуя логике и здравому смыслу. Оставляйте оператор в конце предыдущей строки (а не в начале следующей), чтобы было понятно, что данная строка является продолжением предыдущей.

В языке C++ функции часто оказываются более короткими, чем в C, но по-прежнему остается в силе старый добрый совет: старайтесь сохранять свои функции достаточно короткими, чтобы всю функцию можно было увидеть на экране.

Конструкции с оператором switch

В конструкциях с оператором `switch` используйте отступы таким образом, чтобы четко выделить различные варианты:

```
switch(переменная)
{
    case Значение_1:
        Операция_1();
        break;
    case Значение_2:
        Операция_2();
        break;
    default:
        assert("Ошибочное действие");
        break;
}
```

Текст программы

Чтобы создавать программы, которые будут простыми для чтения, воспользуйтесь следующими советами. Если код просто читать, его нетрудно будет и поддерживать.

- Используйте пробелы, чтобы сделать текст программы более разборчивым.
- Не используйте пробелы внутри ссылок на объекты и массивы (`.`, `->`, `[]`).
- Унарные операторы логически связаны со своими операндами, поэтому не ставьте между ними пробелов. К унарным операторам относятся следующие: `!`, `~`, `++`, `--`, `-`, `*` (для указателей), `&` (преобразования типа), `sizeof`.
- Бинарные операторы должны иметь пробелы с обеих сторон: `+`, `=`, `*`, `/`, `%`, `>>`, `<<`, `<`, `>`, `==`, `!=`, `&`, `|`, `&&`, `||`, `?:`, `-=`, `+=` и т.д.
- Не используйте отсутствие пробелов для обозначения приоритета (`4+ 3*2`).
- Ставьте пробелы после запятых и точек с запятой, но не перед ними.
- Круглые скобки не должны отделяться пробелами от заключенных в них параметров.
- Ключевые слова, такие как `if`, следует отделять пробелами: `if (a == b)`.
- Текст комментария следует отделять пробелом от символов `//`.
- Размещайте спецификаторы указателей и ссылок рядом с именем типа, а не с именем переменной, т.е.

```
char* foo;
int& theInt;
```

а не:

```
char *foo;
int &theInt;
```

- Не объявляйте больше одной переменной в одной строке.

Имена идентификаторов

Ниже перечислены рекомендации для работы с идентификаторами.

- Имена идентификаторов должны быть такими, чтобы по ним было легко понять назначение идентификатора.
- Избегайте непонятных сокращений.
- Не жалейте времени и энергии для подбора кратких, но метких имен.
- Нет смысла использовать венгерскую систему имен (устанавливать связь между типом переменной и первой буквой ее имени). В языке C++ строго соблюдается контроль за соответствием типов, и нет никакой необходимости придумывать дополнительные средства контроля. Для типов, определяемых пользователем (классов), венгерская система имен вообще теряет смысл. Исключением может быть разве что использование префикса (`p`) для указателей и (`r`) для ссылок, а также префиксов `its` или `my` для переменных-членов класса.
- Короткие имена (`i`, `p`, `x` и т.д.) должны использоваться только там, где их краткость делает код более читабельным, а использование настолько очевидно, что в более описательных именах нет необходимости.

- Длина имени переменной должна быть пропорциональна ее области видимости.
- Во избежание путаницы и конфликтов имен убедитесь в том, что все идентификаторы пишутся по-разному.
- Имена функций (или методов) обычно представляют собой глаголы или отглагольные существительные: `Search()`, `Reset()`, `FindParagraph()`, `ShowCursor()`. В качестве имен переменных обычно используются абстрактные существительные, иногда с дополнительным существительным: `count`, `state`, `windSpeed`, `windowHeight`. Логические переменные должны называться в соответствии с их назначением: `windowIconized`, `fileIsOpen`.

Правописание и использование прописных букв в именах

При создании собственного стиля важное значение имеет проверка правописания и использование в именах прописных букв. Считаю, что вам не стоит пренебрегать приведенными ниже советами.

- Используйте прописные буквы и символ подчеркивания, чтобы отделить слова в имени идентификатора, например `SOURCE_FILE_TEMPLATE`. Однако имена, полностью состоящие из прописных букв, в C++ довольно редки. Они используются разве что для констант и шаблонов.
- В именах других идентификаторов сочетаются строчные и прописные буквы с символами подчеркивания. Имена функций, методов, классов, типов и структур должны начинаться с прописных букв. Переменные-члены или локальные переменные обычно начинаются со строчных букв.
- Константы перечислений должны начаться несколькими строчными буквами, представляющими аббревиатуру имени перечисления, например:

```
enum TextStyle
{
    tsPlain,
    tsBold,
    tsItalic,
    tsUnderscore,
} ;
```

Комментарии

Комментарии значительно облегчают понимание программы. Иногда работа над программой прерывается на несколько дней или даже месяцев. За это время можно забыть, что делается в той или иной части программы либо зачем был написан определенный фрагмент кода. Проблемы могут также возникать в том случае, если ваш код анализирует кто-то другой (а не вы сами). Комментарии, используемые в соответствии с согласованным и хорошо продуманным стилем, оправдывают затраченные на них усилия. Предлагаю несколько советов, которые стоит помнить при использовании комментариев.

- Везде, где возможно, используйте для комментариев стиль C++, т.е. символы //, а не пары символов /* */.
- Комментарии более высокого уровня гораздо важнее, чем описание отдельного метода. Поясняйте смысл происходящего, а не дублируйте словами выполняемые операции, как в этом примере:

```
n++; // n инкрементируется на единицу
```

Этот комментарий не стоит времени, затраченного на его ввод. Уделите внимание семантике функций и блоков кода. Опишите, что делает функция. Укажите побочные эффекты, типы параметров и возвращаемые значения. Опишите все допущения, которые были сделаны (или не сделаны), например "предположим, что n неотрицателен", или "функция возвращает -1, если x имеет недопустимое значение". В случае ветвления программы указывайте, при каких условиях будет выполняться эта часть кода.

- Используйте законченные предложения с соответствующей пунктуацией и прописными буквами в начале предложений. Потом вы скажете себе за это спасибо. Избегайте условных обозначений и сокращений, понятных только вам. То, что сейчас кажется очевидным, через несколько месяцев может показаться абсолютно непонятным.
- Используйте пустые строки для отделения логических блоков программы. Объединяйте строки программы в логические группы.

Организация доступа к данным и методам

Организация доступа к данным и методам также должна подчиняться определенным правилам. Ниже приведен ряд советов, касающихся того, как нагляднее описать в программе различия в доступе к ее разным членам.

- Всегда используйте спецификаторы `public:`, `private:` и `protected:`. Не следует полагаться на установки доступа, делаемые по умолчанию.
- Сначала объявите открытые (`public`) члены, затем защищенные (`protected:`), а за ними закрытые (`private:`). Объявляйте переменные-члены после методов.
- Сначала объявите конструктор (конструкторы), а затем — деструктор. Сгруппируйте вместе перезагружаемые методы и методы доступа.
- Методы и переменные-члены внутри каждой группы желательно расположить по именам в алфавитном порядке. Следует также упорядочить по алфавиту включения файлов с помощью директивы `#include`.
- Несмотря на то что с замещенными функциями использование ключевого слова `virtual` необязательно, лучше им не пренебрегать. Оно напомнит вам, что данная функция является виртуальной, и обеспечит преемственность объявлений.

Определения классов

Старайтесь сохранять порядок определения методов таким же, как и в объявлении. Это ускорит поиск нужного метода.

При определении функции размещайте тип возвращаемого значения и все другие спецификаторы на предыдущей строке, чтобы имя класса и функции располагалось в начале строки. Это значительно облегчит поиск функций.

Включение файлов

Старайтесь избегать этого в файлах заголовков, кроме случая включения файла заголовка базового класса, от которого производится данный класс. Использование директив `#include` также необходимо в тех случаях, когда в объявляемом классе используются объекты другого класса. Для классов, на которые просто делаются ссылки, достаточно будет передать ссылку или указатель.

Если же все-таки необходимо включить некоторый файл в программу, сделайте это в файле заголовка, даже если вы полагаете, что этот файл будет включен и в файле источника.

Во всех файлах заголовков следует использовать систему защиты от повторного объявления, с которой вы познакомились выше при рассмотрении макросов.

Макрос assert()

Используйте макрос `assert()` без всяких ограничений. Он не только помогает находить ошибки, но и облегчает чтение программы. Этот макрос также помогает сконцентрировать мысли автора на том, что является допустимым, а что — нет.

Ключевое слово const

Используйте ключевое слово `const` везде, где считаете нужным: для параметров, переменных и методов. Часто существует потребность в использовании как константных, так и неконстантных версий некоторых методов. Будьте очень осторожны при явном приведении типа с константного к неконстантному и наоборот (хотя иногда такой подход оказывается единственным способом решения проблемы). Убедитесь в целесообразности этих действий и добавьте подробный комментарий.

Сделаем еще один шаг вперед

Долгие три недели вы отдали усердной работе над освоением средств программирования языка C++ и теперь вполне можете считать себя компетентным программистом C++. Но на этом ни в коем случае нельзя ставить точку. Несмотря на то что, прочитав эту книгу, вы узнали много полезных вещей, гораздо больший объем вам еще предстоит узнать, причем сначала имеет смысл познакомиться с источниками ценной информации, благодаря которой из новичка вы превратитесь в опытного программиста C++.

В следующих разделах рекомендуется ряд конкретных источников информации, причем эти рекомендации отражают лишь мой персональный опыт и мое личное мнение. В последнее время ежегодно издается множество книг, посвященных программированию на C++, поэтому, прежде чем делать покупку, постарайтесь проконсультироваться со специалистами в этой области.

Где получить справочную информацию и советы

Первое, что вам стоит сделать, — отыскать в Internet одну из конференций по C++. Эти группы поддерживают непосредственный контакт с сотнями и даже тысячами программистов C++, которые смогут ответить на ваши вопросы, предложить советы и подсказать решения для ваших идей.

Я принимаю участие в группах новостей Internet, посвященных C++ (`comp.lang.c++` и `comp.lang.c++.moderated`), и рекомендую их в качестве превосходных источников информации и поддержки.

Кроме того, стоит поискать локальные группы пользователей. Во многих городах есть так называемые группы по интересам (в том числе и группы C++), в которых можно встретить других программистов и обменяться идеями.

Журналы

Закрепить свои навыки можно, подписавшись на хороший журнал, посвященный программированию на языке C++. По моему мнению, самым лучшим журналом по этой тематике является *C++ Report* издательства SIGS Publications. Каждый выпуск этого журнала содержит полезные статьи, поэтому их стоит сохранять — ведь то, что не волнует вас сегодня, станет чрезвычайно важным уже завтра. (Предостережение: я написал об этом журнале в первом и втором издании книги, но теперь я веду в нем ежемесячную рубрику, и потому налицо конфликт интересов. Тем не менее я по-прежнему считаю, что этот журнал — потрясающее издание.)

Журнал *C++ Report* можно приобрести в издательстве SIGS Publications по адресу: P.O. Box 2031, Langhorne, PA 19047-9700.

Выскажите свое мнение о книге

Если у вас есть комментарии, предложения или замечания относительно этой или других книг, я бы с интересом их выслушал. Пишите мне по адресу: `jliberty@libertyassociates.com` или посетите мой Web-узел: `www.libertyassociates.com`. Я с нетерпением буду ждать ваши отзывы.

Рекомендуется

Обратитесь к другим книгам по C++. Нет такой книги, в которой все темы были бы рассмотрены одинаково полно и которая могла бы научить всему, что нужно знать профессиональному программисту C++.

Подпишитесь на хороший журнал по C++ и подключитесь к хорошей группе новостей пользователей C++.

Не рекомендуется

Не ограничивайтесь при освоении C++ только чтением чужих программ. Лучший способ изучения языка — самому писать программы.

Резюме

Сегодня вы узнали много подробностей о работе с препроцессором. При каждом запуске компилятора сначала запускается препроцессор, который расшифровывает и выполняет такие директивы, как `#define` и `#ifdef`.

Препроцессор осуществляет текстовую подстановку, хотя использование макросов может несколько усложнить чтение программы. С помощью таких директив, как `#ifdef`, `#else` и `#ifndef`, можно выполнять условную компиляцию, которая позволяет компилировать один набор команд при одних условиях, и другой — при других условиях. Благодаря этому можно писать программы сразу для нескольких платформ и успешно выполнять их отладку.

Макросы обеспечивают сложную текстовую подстановку на основе параметров, передаваемых им во время компиляции программы. Чтобы гарантировать правильное выполнение подстановки, каждый параметр макроса нужно заключить в круглые скобки.

В C++ макросы, в частности и препроцессор, вообще несут на себе меньшую нагрузку, чем это было в языке C. В C++ предусмотрен ряд таких средств программиро-

вания, как ключевое слово const и шаблоны, которые предлагают лучшие альтернативы использованию препроцессора.

Кроме того, вы узнали, как устанавливать и возвращать значения отдельных битов и как выделять ограниченное число битов для переменных-членов класса.

Наконец, вы получили информацию о стиле написания программы на языке C++, а также о том, с помощью каких первоисточников лучше продолжить изучение C++.

Вопросы и ответы

Если C++ предлагает лучшие решения, чем препроцессор, почему же эти средства все еще доступны?

Во-первых, язык C++ совместим с языком C, поэтому все существенные компоненты языка C должны поддерживаться в C++. Во-вторых, некоторые возможности препроцессора все еще используются в C++, например защита от повторного включения файла.

Зачем использовать макросы там, где можно использовать обычные функции?

Макросы вставляются компилятором прямо в код программы, что ускоряет их выполнение. Кроме того, с помощью макросов можно динамически менять типы в объявлениях, хотя использовать для этого шаблоны предпочтительнее.

В каких случаях лучше использовать макрос, чем подставляемую функцию?

Часто это не имеет большого значения, поэтому используйте тот вариант, который кажется вам проще. Однако макросы предоставляют такие дополнительные возможности, как замена символов, взятие в кавычки и конкатенация. Ни одна из этих возможностей не реализуется с помощью подставляемой функции.

Какая альтернатива использованию препроцессора для печати промежуточных значений в процессе отладки?

Лучше всего использовать оператор watch внутри отладчика. За информацией о его использовании обращайтесь к документации, прилагаемой к вашему компилятору или отладчику.

Когда лучше использовать макрос assert(), а когда — исключение?

Если ошибка в выполнении программы может возникнуть из-за внешних причин, таких как некорректный ввод данных пользователем или несовершенство компьютерной системы, используйте исключения. Если же сбой программы возникает вследствие синтаксических или логических ошибок в коде программы, используйте макрос assert().

Когда лучше использовать битовые поля вместо обычных переменных?

Когда критическим фактором становится размер объектов. При работе с ограниченным объемом памяти или с программным обеспечением устройств связи вы почувствуете, что экономия на каждом байте существенна для успеха вашей программы.

Почему споры о стилях столь эмоциональны?

Программисты очень привязываются к выработанному ими стилю. Допустим, вы привыкли использовать отступы следующим образом:

```
if (НекотороеУсловие){
    // выражения
}     // закрывающая фигурная скобка
```

Согласитесь, что вам будет трудно отказаться от этой привычки — ведь новые стили кажутся поначалу неправильными и вводящими в заблуждение. Если вы устали от моих советов, попробуйте подключиться к популярной группе новостей и узнайте, какой стиль выравнивания работает лучше всего, какой редактор предпочтительнее для

C++ или какую программу лучше всего использовать в качестве текстового процессора. Затем усаживайтесь поудобнее и приготовьтесь прочитать тысяч десять противоречащих друг другу сообщений.

Пришло время прощаться?

Да! Вы изучили C++, хотя... нет. Еще десять лет назад один человек мог изучить все, что было известно в мире о микроЭВМ, или, по крайней мере, чувствовать себя вполне уверенно в этом вопросе. Сегодня это исключено даже теоретически. Одному человеку невозможно разобраться во всем, и даже за то время, пока вы попытаетесь это сделать, ситуация в индустрии программирования изменится, а значит, вы опять отстанете от нее. Тем не менее обязательно продолжайте читать, постоянно обращайтесь к различным источникам — журналам и группам новостей, которые будут держать вас в курсе самых последних новшеств в этой области.

Контрольные вопросы

1. Для чего нужны средства защиты от повторного включения?
2. Как указать компилятору, что необходимо напечатать содержимое промежуточного файла, полученного в результате работы препроцессора?
3. Какова разница между директивами `#define debug 0` и `#undef debug`?
4. Что делает оператор дополнения до единицы?
5. Чем отличается оператор побитового ИЛИ от оператора исключающего побитового ИЛИ?
6. Какова разница между операторами `&` и `&&`?
7. В чем разница между операторами `|` и `||`?

Упражнения

1. Создайте защиту от повторного включения файла заголовка `STRING.H`.
2. Напишите макрос `assert()`, который
   - будет печатать сообщение об ошибке, а также имя файла и номер строки, если уровень отладки равен 2;
   - будет печатать сообщение (без имени файла и номера строки), если уровень отладки равен 1;
   - не будет ничего делать, если уровень отладки равен 0.
3. Напишите макрос `DPrint`, который проверяет, определена ли лексема `DEBUG`, и, если да, выводит значение, передаваемое как параметр.
4. Напишите программу, которая складывает два числа без использования операции сложения (+). Подсказка: используйте побитовые операторы!

Подведение итогов

В приведенной ниже программе используются многие "продвинутые" методы, с которыми вы познакомились на протяжении трех недель усердных занятий. Программа содержит связанный список, основанный на шаблоне; кроме того, в ней проводится обработка исключительных ситуаций. Тщательно разберитесь в этой программе, и, если полностью ее поймете, значит, вы — программист C++.

Если ваш компилятор не поддерживает шаблоны или блоки `try` и `catch`, вы не сможете скомпилировать эту программу.

Листинг 3.1. Программа, основанная на материалах недели 3

```
1:   // **********************************************
2:   //
3:   // Название:     Обзор недели 3
4:   //
5:   // Файл:       Week3
6:   //
7:   // Описание: Программа с использованием связанного списка
8:   // на основе шаблона с обработкой исключительных ситуаций
9:   //
10:   // Классы:    PART - хранит номера запчастей и потенциально другую
11:   // информацию о запчастях. Это будет
12:   // пример класса для хранения списка.
13:   // Обратите внимание на использование
14:   // оператора << для печати информации о запчасти
15:   // на основе его типа и времени выполнения.
16:   //
17:   //              Node - действует как узел в классе List
18:   //
19:   //              List - список, основанный на шаблоне, который
20:   //  обеспечивает работу связанного списка
21:   //
22:   //
23:   // Автор:    Jesse Liberty (jl)
```

```
24:  //
25:  // Разработан:   Pentium 200 Pro. 128MB RAM MVC 5.0
26:  //
27:  // Требования:    Не зависит от платформы
28:  //
29:  // История создания:  9/94 - Первый выпуск (jl)
30:  //                    4/97 - Обновлено (jl)
31:  // ************************************************
32:
33:  #include <iostream.h>
34:
35:  // классы исключений
36:  class Exception { } ;
37:  class OutOfMemory :   public Exception{ } ;
38:  class NullNode :    public Exception{ } ;
39:  class EmptyList :   public Exception { } ;
40:  class BoundsError :   public Exception { } ;
41:
42:
43:  // *************** Part ************
44:  // Абстрактный базовый класс запчастей
45:  class Part
46:  {
47:  public:
48:   Part():itsObjectNumber(1) { }
49:   Part(int ObjectNumber):itsObjectNumber(ObjectNumber){ }
50:   virtual ~Part(){ } ;
51:   int GetObjectNumber() const {  return itsObjectNumber; }
52:   virtual void Display() const =0;  // функция будет замещена в производном классе
53:
54:  private:
55:   int itsObjectNumber;
56:  } ;
57:
58:  // выполнение чистой виртуальной функции, необходимой
59:  // для связывания объектов производного класса
60:  void Part::Display() const
61:  {
62:    cout << "\nPart Number: " << itsObjectNumber << endl;
63:  }
64:
65:  // Этот оператор << будет вызываться для всех объектов запчастей.
66:  // Его не нужно объявлять другом, поскольку он не обращается к закрытым данным.
67:  // Он вызывает метод Display(), в результате чего реализуется полиморфизм классов.
68:  // Было бы не плохо замещать функцию оператора для разных
69:  // типов thePart, но C++ не поддерживает контравариантность
70:  ostream& operator<<( ostream& theStream,Part& thePart)
71:  {
72:   thePart.Display();  // косвенная реализация полиморфизма оператора вывода!
73:   return theStream;
74:  }
```

```
75:
76:   // *************** Car Part ************
77:   class CarPart : public Part
78:   {
79:   public:
80:    CarPart():itsModelYear(94){ }
81:    CarPart(int year, int partNumber);
82:    int GetModelYear() const {  return itsModelYear; }
83:    virtual void Display() const;
84:   private:
85:    int itsModelYear;
86:   } ;
87:
88:   CarPart::CarPart(int year, int partNumber):
89:    itsModelYear(year),
90:    Part(partNumber)
91:   { }
92:
93:   void CarPart::Display() const
94:   {
95:       Part::Display();
96:       cout << "Model Year: " << itsModelYear << endl;
97:   }
98:
99:   // *************** AirPlane Part ************
100:   class AirPlanePart : public Part
101:   {
102:   public:
103:    AirPlanePart():itsEngineNumber(1){ } ;
104:    AirPlanePart(int EngineNumber, int PartNumber);
105:    virtual void Display() const;
106:    int GetEngineNumber()const {  return itsEngineNumber; }
107:   private:
108:    int itsEngineNumber;
109:   } ;
110:
111:   AirPlanePart::AirPlanePart(int EngineNumber, int PartNumber):
112:    itsEngineNumber(EngineNumber),
113:    Part(PartNumber)
114:   { }
115:
116:   void AirPlanePart::Display() const
117:   {
118:    Part::Display();
119:    cout << "Engine No.: " << itsEngineNumber << endl;
120:   }
121:
122:   // Объявление класса List
123:   template <class T>
124:   class List;
125:
```

```
126:  // ***************  Node ************
127:  // Общий узел, который можно добавить к списку
128:  // ************************************
129:
130:  template <class T>
131:  class Node
132:  {
133:  public:
134:   friend class List<T>;
135:   Node (T*);
136:   ~Node();
137:   void SetNext(Node * node) {  itsNext = node; }
138:   Node * GetNext() const;
139:   T * GetObject() const;
140:  private:
141:   T* itsObject;
142:   Node * itsNext;
143:  } ;
144:
145:  // Выполнение узла...
146:
147:  template <class T>
148:  Node<T>::Node(T* pOjbect):
149:  itsObject(pOjbect),
150:  itsNext(0)
151:  { }
152:
153:  template <class T>
154:  Node<T>::~Node()
155:  {
156:   delete itsObject;
157:   itsObject = 0;
158:   delete itsNext;
159:   itsNext = 0;
160:  }
161:
162:  // Возвращает значение NULL, если нет следующего узла
163:  template <class T>
164:  Node<T> * Node<T>::GetNext() const
165:  {
166:      return itsNext;
167:  }
168:
169:  template <class T>
170:  T * Node<T>::GetObject() const
171:  {
172:   if (itsObject)
173:      return itsObject;
174:   else
175:      throw NullNode();
176:  }
```

```
177:
178:  // ***************  List ************
179:  // Общий шаблон списка
180:  // Работает с любым нумерованным объектом
181:  // ***********************************
182:  template <class T>
183:  class List
184:  {
185:  public:
186:   List();
187:   ~List();
188:
189:   T*         Find(int & position, int ObjectNumber)  const;
190:   T*       GetFirst() const;
191:   void       Insert(T *);
192:   T*       operator[](int) const;
193:   int       GetCount() const {  return itsCount; }
194:  private:
195:   Node<T> * pHead;
196:   int       itsCount;
197:  } ;
198:
199:  // Выполнение списка...
200:  template <class T>
201:  List<T>::List():
202:   pHead(0),
203:   itsCount(0)
204:   { }
205:
206:  template <class T>
207:  List<T>::~List()
208:  {
209:   delete pHead;
210:  }
211:
212:  template <class T>
213:  T*   List<T>::GetFirst() const
214:  {
215:   if (pHead)
216:      return pHead->itsObject;
217:   else
218:      throw EmptyList();
219:  }
220:
221:  template <class T>
222:  T *  List<T>::operator[](int offSet) const
223:  {
224:   Node<T>* pNode = pHead;
225:
226:   if (!pHead)
227:      throw EmptyList();
```

```
228:
229:    if (offSet > itsCount)
230:       throw BoundsError();
231:
232:    for (int i=0;i<offSet; i++)
233:       pNode = pNode->itsNext;
234:
235:   return   pNode->itsObject;
236:   }
237:
238:   // Находим данный объект в списке на основе его идентификационного номера (id)
239:   template <class T>
240:   T*   List<T>::Find(int & position, int ObjectNumber)  const
241:   {
242:    Node<T> * pNode = 0;
243:    for (pNode = pHead, position = 0;
244:          pNode!=NULL;
245:          pNode = pNode->itsNext, position++)
246:    {
247:       if (pNode->itsObject->GetObjectNumber() == ObjectNumber)
248:          break;
249:    }
250:    if (pNode == NULL)
251:       return NULL;
252:    else
253:       return pNode->itsObject;
254:   }
255:
256:   // добавляем в список, если номер объекта уникален
257:   template <class T>
258:   void List<T>::Insert(T* pObject)
259:   {
260:    Node<T> * pNode = new Node<T>(pObject);
261:    Node<T> * pCurrent = pHead;
262:    Node<T> * pNext = 0;
263:
264:    int New =  pObject->GetObjectNumber();
265:    int Next = 0;
266:    itsCount++;
267:
268:    if (!pHead)
269:    {
270:       pHead = pNode;
271:       return;
272:    }
273:
274:    // если номер текущего объекта меньше номера головного,
275:    // то этот объект становится новым головным узлом
276:    if (pHead->itsObject->GetObjectNumber() > New)
277:    {
278:       pNode->itsNext = pHead;
```

```
279:       pHead = pNode;
280:       return;
281:    }
282:
283:    for (;;)
284:    {
285:       // если нет следующего объекта, добавляем в конец текущий объект
286:       if (!pCurrent->itsNext)
287:       {
288:          pCurrent->itsNext = pNode;
289:          return;
290:       }
291:
292:       // если данный объект больше текущего, но меньше следующего,
293:       // то вставляем его между ними, в противном случае переходим к следующему объекту
294:       pNext = pCurrent->itsNext;
295:       Next = pNext->itsObject->GetObjectNumber();
296:       if (Next > New)
297:       {
298:          pCurrent->itsNext = pNode;
299:          pNode->itsNext = pNext;
300:          return;
301:       }
302:       pCurrent = pNext;
303:    }
304: }
305:
306:
307: int main()
308: {
309:  List<Part> theList;
310:  int choice;
311:  int ObjectNumber;
312:  int value;
313:  Part * pPart;
314:  while (1)
315:  {
316:     cout << "(0)Quit (1)Car (2)Plane: ";
317:     cin >> choice;
318:
319:     if (!choice)
320:        break;
321:
322:     cout << " New PartNumber?: ";
323:     cin >>  ObjectNumber;
324:
325:     if (choice == 1)
326:     {
327:        cout << "Model Year?: ";
328:        cin >> value;
329:        try
```

```
330:          {
331:             pPart = new CarPart(value,ObjectNumber);
332:          }
333:          catch (OutOfMemory)
334:          {
335:             cout << "Not enough memory; Exiting..." << endl;
336:             return 1;
337:          }
338:       }
339:       else
340:       {
341:          cout << "Engine Number?: ";
342:          cin >> value;
343:          try
344:          {
345:             pPart = new AirPlanePart(value,ObjectNumber);
346:          }
347:          catch (OutOfMemory)
348:          {
349:             cout << "Not enough memory; Exiting..." << endl;
350:             return 1;
351:          }
352:       }
353:       try
354:       {
355:          theList.Insert(pPart);
356:       }
357:       catch (NullNode)
358:       {
359:          cout << "The list is broken, and the node is null!" << endl;
360:          return 1;
361:       }
362:       catch (EmptyList)
363:       {
364:          cout << "The list is empty!" << endl;
365:          return 1;
366:       }
367:    }
368:    try
369:    {
370:       for (int i = 0; i < theList.GetCount(); i++ )
371:          cout << *(theList[i]);
372:    }
373:       catch (NullNode)
374:       {
375:          cout << "The list is broken, and the node is null!" << endl;
376:          return 1;
377:       }
378:       catch (EmptyList)
379:       {
380:          cout << "The list is empty!" << endl;
```

```
381:          return 1;
382:       }
383:       catch (BoundsError)
384:       {
385:          cout << "Tried to read beyond the end of the list!" << endl;
386:          return 1;
387:       }
388:    return 0;
389:    }
```

РЕЗУЛЬТАТ

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2837
Model Year? 90

 (0)Quit (1)Car (2)Plane: 2
New PartNumber?: 378
Engine Number?: 4938

 (0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4499
Model Year? 94

 (0)Quit (1)Car (2)Plane: 1
New PartNumber?: 3000
Model Year? 93

 (0)Quit (1)Car (2)Plane: 0

Part Number: 378
Engine No. 4938

Part Number: 2837
Model Year: 90

Part Number: 3000
Model Year: 93

Part Number 4499
Model Year: 94
```

Итоговая программа, основанная на материале за неделю 3, — это модификация программы, приведенной в обзорной главе по материалам за неделю 2. Изменения заключались в добавлении шаблона, обработке объекта `ostream` и исключительных ситуаций. Результаты работы обеих программ идентичны.

В строках 36–40 объявляется ряд классов исключений. В этой программе используется несколько примитивная обработка исключительных ситуаций. Классы исключений не содержат никаких данных или методов, они служат флагами для перехвата блоками `catch`, которые выводят простые предупреждения, а затем выполняют выход.

Более надежная программа могла бы передать эти исключения по ссылке, а затем извлечь контекст или другие данные из объектов исключения, чтобы попытаться исправить возникшую проблему.

В строке 45 объявляется абстрактный класс Part, причем точно так же, как это было сделано в листинге, обобщающем материал за неделю 2. Единственное интересное изменение здесь — это использование оператора operator<<(), который не является членом класса (он объявляется в строках 70–74). Обратите внимание, что он не является ни членом класса запчастей Part, ни другом класса Part. Он просто принимает в качестве одного из своих параметров ссылку на класс Part.

Возможно, вы бы хотели иметь замещенный оператор operator<<() для объектов классов CarPart и AirPlanePart с учетом различий в типах объектов. Но поскольку программа передает указатель на объект базового класса Part, а не указатель на указатель производных классов CarPart и AirPlanePart, то выбор правильной версии функции пришлось бы основывать не на типе объекта, а на типе одного из параметров функции. Это явление называется контравариантностью и не поддерживается в C++.

Есть только два пути достижения полиморфизма в C++: использование полиморфизма функций и виртуальных функций. Полиморфизм функций здесь не будет работать, сигнатуры функций, принимающих ссылку на класс Part, одинаковы.

Виртуальные функции также не будут здесь работать, поскольку оператор operator<< не является функцией-членом класса запчастей Part. Вы не можете сделать оператор operator<< функцией-членом класса Part, потому что в программе потребуется выполнить следующий вызов:

```
cout << thePart
```

Это означает, что фактически вызов относится к объекту cout.operator<<(Part&), а объект cout не имеет версии оператора operator<<, который принимает ссылку на класс запчастей Part!

Чтобы обойти это ограничение, в приведенной выше программе используется только один оператор operator<<, принимающий ссылку на класс Part. Затем вызывается метод Display(), который является виртуальной функцией-членом, в результате чего вызывается правильная версия этого метода.

В строках 130–143 класс Node определяется как шаблон. Он играет ту же роль, что и класс Node в программе из обзора за неделю 2, но эта версия класса Node не связана с объектом класса Part. Это значит, что данный класс может создавать узел фактически для любого типа объекта.

Обратите внимание: если вы попытаетесь получить объект из класса Node и окажется, что не существует никакого объекта, то такая ситуация рассматривается как исключительная и исключение генерируется в строке 175.

В строках 182–183 определяется общий шаблон класса List. Этот класс может содержать узлы любых объектов, которые имеют уникальные идентификационные номера, кроме того, он сохраняет их отсортированными в порядке возрастания номеров. Каждая из функций списка проверяет ситуацию на исключительность и при необходимости генерирует соответствующие исключения.

В строках 307–308 управляющая программа создает список двух типов объектов класса Part, а затем печатает значения объектов в списке, используя стандартные потоки вывода.

Если бы в языке C++ поддерживалась контравариантность, можно было бы вызывать замещенные функции, основываясь на типе объекта указателя, на который ссылается указатель базового класса. Программа, представленная в листинге 3.2, демонстрирует суть контравариантности, но, к сожалению, ее нельзя будет скомпилировать в C++.

Вопросы и ответы

В комментарии, содержащемся в строках 65–69, говорится, что C++ не поддерживает контравариантность. Что такое контравариантность?

Контравариантностью называется возможность создания указателя базового класса на указатель производного класс.

ВНИМАНИЕ: Этот листинг не будет скомпилирован!

Листинг 3.2. Пример контравариантности

```
#include<iostream.h>
class Animal
{
public:
virtual void Speak() {  cout << "Animal
                  Speaks\ n"; }
} ;

class Dog : public Animal
{
public:
        void Speak() {  cout << "Dog Speaks\ n"; }
} ;


class Cat : public Animal
{
public:
        void Speak() {  cout << "Cat Speaks\ n"; }
} ;

void DoIt(Cat*);
void DoIt(Dog*);

int main()
{

        Animal * pA = new Dog;
        DoIt(pA);
        return 0;
}

void DoIt(Cat * c)
{
        cout << "They passed a cat!\ n" << endl;
```

```
        c->Speak();
}

void DoIt(Dog * d)
{
        cout << "They passed a dog!\ n" << endl;
        d->Speak();
}
```

Но в C++ эту проблему можно решить с помощью виртуальной функции.

```
#include<iostream.h>

class Animal
{
public:
virtual void Speak() {  cout << "Animal Speaks\ n"; }
} ;

class Dog : public Animal
{
public:
        void Speak() {  cout << "Dog Speaks\ n"; }
} ;


class Cat : public Animal
{
public:
        void Speak() {  cout << "Cat Speaks\ n"; }
} ;

void DoIt(Animal*);

int main()
{

        Animal * pA = new Dog;
        DoIt(pA);
        return 0;
}

void DoIt(Animal * c)
{
        cout << "They passed some kind of
                  animal\ n" << endl;
        c->Speak();
}
```

Приоритеты операторов

Важно понять, что операторы имеют приоритеты, но запоминать их совсем не обязательно.

Приоритет оператора определяет последовательность, в которой программа выполняет операторы в выражении или формуле. Если один оператор имеет приоритет над другим оператором, то он выполняется первым.

Приоритет оператора убывает с увеличением номера категории. Все операторы одной категории имеют равный приоритет. Унарные операторы (категория 3), условный оператор (категория 14) и операторы присваивания (категория 15) ассоциируются справа налево, все остальные — слева направо. В приведенной ниже таблице операторы перечислены по категориям в порядке убывания их приоритетности.

Приоритеты операторов

Категория	Название или действие	Оператор
1 (наивысшего приоритета)	Разрешение области видимости, индексирования	`:: []`
2	Прямое и косвенное обращение к члену класса	`. ->`
	Вызов функции	`()`
	Постфиксные инкремент и декремент	`++ --`
3 (унарные)	Префиксные инкремент и декремент	`++ --`
	Размер	`sizeof, sizeof()`
	Дополнение до единицы и логическое отрицание	`~ !`
	Унарные минус и плюс	`- +`
	Получение адреса и разыменование	`& *`
	Создание и удаление динамического объекта	`new, new[], delete, delete[]`
	Приведение типа	`casting`
4 (мультипликативные)	Умножение, деление, деление по модулю	`* / %`
5 (аддитивные)	Бинарный плюс, бинарный минус	`+ -`
6 (сдвига)	Вывода и ввода	`<< >>`

7 (отношения)	Меньше, меньше или равно, больше, больше или равно	`< <= > >=`
8 (равенства)	Равно, не равно	`== !=`
9	Побитовое И	`&`
10	Побитовое исключающее ИЛИ	`^`
11	Побитовое ИЛИ	`|`
12	Логическое И	`&&`
13	Логическое ИЛИ	`||`
14	Условный	`?:`
15 (присваивания)	Простое присваивание	`=`
	Присваивание с умножением и делением	`*= /=`
	Присваивание с делением по модулю	`%=`
	Присваивание с суммой и разностью	`+= -=`
	Присваивание со сдвигом	`<<= >>=`
	Присваивание с побитовыми И, включающим ИЛИ и исключающим ИЛИ	`&= |= ^=`
17	Генерация исключения	`throw`
18	Запятая	`,`

Ключевые слова C++

Ключевые слова зарезервированы в компиляторе как элементы языка программирования. Эти ключевые слова нельзя использовать в качестве имен при определении классов, переменных или функций. Приведенный список нельзя назвать абсолютно строгим, поскольку некоторые ключевые слова зависят от конкретного компилятора. Поэтому список ключевых слов вашего компилятора может быть немного другим.

```
auto
break
case
catch
char
class
const
continue
default
delete
do
double
else
enum
extern
float
for
friend
goto
if
int
long
mutable
new
operator
private
protected
public
register
return
short
signed
sizeof
static
struct
switch
template
this
throw
typedef
union
unsigned
virtual
void
volatile
while
```

SAMS

Освой самостоятельно

Двоичные и шестнадцатеричные числа

С основами арифметики вы познакомились так давно, что, вероятно, вам трудно представить свою жизнь без этих знаний. Взглянув на число 145, вы без малейших колебаний скажете, что это сто сорок пять.

Понимание двоичных и шестнадцатеричных чисел потребует по-новому взглянуть на число 145 и увидеть в нем не число, а некоторый код для его выражения.

Начнем с малого. Рассмотрим взаимоотношения между числом три и символом "3". Символ числа (цифра) 3 — это некая "закорючка" на листе бумаги, число три — это некоторая идея или понятие. Определенная цифра используется для представления определенного числа.

Отличие между идеей и символом становится яснее, если осознавать, что для представления одного и того же понятия могут использоваться совершенно разные символы: `три`, `3`, `|||`, `III` или `***`.

В десятичной системе счисления для представления чисел используются цифры 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Как же представляется число десять?

Здесь возможны разные варианты. Можно было бы для представления этого понятия использовать букву `A` или "сороконожку" `IIIIIIIIII`. Римляне использовали символ `X`. В арабской системе (которой придерживаемся и мы) для представления числовых значений играет роль комбинация базовых десяти цифр. Первая (самая крайняя) позиция, или *порядок*, используется для единиц, а расположенная слева от нее — для десятков. Следовательно, число пятнадцать представляется как 15 (читается как "один, пять"), т.е. 1 десяток и 5 единиц.

Итак, вырисовываются некоторые правила, позволяющие сделать ряд обобщений.

1. Для представления чисел по основанию 10 используются цифры 0—9.
2. Порядок представляет собой степень числа десять: единицы (1), десятки (10), сотни (100) и т.д.
3. Поскольку третья позиция в числе представляет сотни, то самым большим двузначным числом может быть 99. В общем случае, используя `n` позиций, можно представить числа от `0` до `(10n-1)`. Следовательно, с помощью трех позиций можно представить числа от `0` до `(103-1)`, или `0-999`.

Другие системы счисления

Отнюдь не случайно мы используем основание 10 — вспомните, ведь у нас на руках 10 пальцев. Однако вполне можно представить арифметику с использованием другого основания. Применяя правила, сформулированные для основания 10, можно описать представление чисел в системе счисления с основанием 8.

1. Для представления чисел по основанию 8 используются цифры 0–7.
2. Позиции разных порядков представляют собой степени числа восемь: единицы (1), восьмерки (8), 64-ки и т.д.
3. Используя n позиций, можно представить числа от 0 до (8^n-1).

Чтобы различать числа, написанные с использованием разных оснований, это основание записывают рядом с числом как нижний индекс. Тогда число пятнадцать по основанию 10 следует записать как 15_{10} и читать как "один, пять по основанию десять".

Таким образом, для представления числа 15_{10} по основанию 8 следует записать 17_8. Это читается как "один, семь по основанию восемь". Обратите внимание, что это число также можно прочитать как "пятнадцать", поскольку именно его мы и имеем в виду, просто используем другое обозначение.

Откуда взялось число 17? Цифра 1 означает одну восьмерку, а цифра 7 означает 7 единиц. Одна восьмерка плюс семь единиц равно пятнадцати. Рассмотрим пятнадцать звездочек:

```
*****      *****
*****
```

Наше естественное желание — создать две группы: одна содержит десять звездочек, а другая — пять. В десятичной системе эта "композиция" представляется числом 15 (1 десяток и 5 единиц). Но те же звездочки можно сгруппировать и по-другому:

```
****       *******
****
```

т.е. имеем две группы: с восемью и семью звездочками. Такое распределение звездочек может служить иллюстрацией представления числа 17_8 с использованием основания восемь (одна восьмерка и семь единиц).

Еще об основаниях

Число пятнадцать по основанию десять представляется как 15, по основанию девять — как 16_9, по основанию восемь — как 17_8, а по основанию семь — как 21_7. В системе счисления по основанию 7 нет цифры 8, поэтому для представления числа пятнадцать нужно использовать две семерки и одну единицу.

Как же прийти к какому-нибудь общему принципу? Чтобы преобразовать десятичное число в число с основанием 7, вспомните о значении каждой порядковой позиции. В семеричной системе счисления переход к следующему порядку будет происходить на значениях, соответствующих десятичным числам: единица, семь, сорок девять, триста сорок три и т.д. Откуда взялись эти числа? Так ведь это же степени числа семь: 7^0, 7^1, 7^2, 7^3 и т.д. Построим следующую таблицу:

4	3	2	1
7^3	7^2	7^1	7^0
343	49	7	1

В первой строке представлен порядок числа. Во второй — степень числа семь, а в третьей — десятичное представление соответствующей степени числа семь.

Чтобы получить представление некоторого десятичного числа в системе счисления с основанием 7, выполните следующую процедуру. Проанализируйте, к числам какого порядка может относиться это значение. Возьмем, к примеру, число 200. Вы уже

знаете, что числа четвертого порядка в семеричной системе счисления начинаются с 343, а потому это может быть только число третьего порядка.

Чтобы узнать, сколько раз число 49 (граничное значение третьего порядка) "поместится" в нашем числе, разделите его на 49. В ответе получается число 4, поэтому поставьте 4 в третью позицию и рассмотрите остаток, который в данном случае тоже равен 4. Поскольку в этом остатке не укладывается ни одной целой семерки, то во второй разряд (второй порядок) помещаем цифру 0. Нетрудно догадаться, что в остатке 4 содержится 4 единицы, поэтому и ставим цифру 4 в первую позицию (порядок единиц). В итоге получаем число 404_7.

Для преобразования числа 968 в систему счисления по основанию 6 используем следующую таблицу:

5	4	3	2	1
6^4	6^3	6^2	6^1	6^0
1296	216	36	6	1

В числе 968 число 1296 (граничное значение пятого порядка) не умещается ни разу, поэтому мы имеем дело с числом четвертого порядка. При делении числа 968 на число 216 (граничное значение четвертого порядка) получается число 4 с остатком, равным 104. В четвертую позицию ставим цифру 4. Делим остаток 104 на число 36 (граничное значение третьего порядка). Получаем в результате деления число 2 и остаток 32. Поэтому третья позиция будет содержать цифру 2. При делении остатка 32 на число 6 (граничное значение второго порядка) получаем 5 и остаток 2. Итак, в ответе имеем число 4252_6, что наглядно показано в следующей таблице:

5	4	3	2	1
6^4	6^3	6^2	6^1	6^0
1296	216	36	6	1
0	4	2	5	2

Для обратного преобразования, т.е. из системы счисления с недесятичным основанием (например, с основанием 6) в десятичную систему, достаточно умножить каждую цифру числа на граничное значение соответствующего порядка, а затем сложить полученные произведения:

4 * 216		864
	=	
2 * 36	=	72
5 * 6	=	30
2 * 1	=	2
		$\Sigma = 968$

Двоичная система счисления

Минимальным допустимым основанием является 2. В этом случае используются только две цифры: 0 и 1. Вот как выглядят порядки двоичного числа:

Порядок	8	7	6	5	4	3	2	1
Степень	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Значение	128	64	32	16	8	4	2	1

Для преобразования числа 88 в двоичное число (с основанием 2) выполните описанную выше процедуру. В числе 88 число 128 не укладывается ни разу, поэтому в восьмой позиции ставим 0.

В числе 88 число 64 укладывается только один раз, поэтому в седьмую позицию ставим 1, а остаток равен 24. В числе 24 число 32 не укладывается ни разу, поэтому шестая позиция содержит 0.

В числе 24 число 16 укладывается один раз, поэтому пятой цифрой двоичного числа будет 1. Остаток при этом равен 8. В остатке 8 число 8 (граничное значение четвертого порядка) укладывается один раз, следовательно, в четвертой позиции ставим 1. Новый остаток равен нулю, поэтому в оставшихся позициях будут стоять нули.

```
0   1   0   1   1   0   0   0
```

Чтобы протестировать полученный ответ, выполним обратное преобразование:

```
1 * 64 = 64
0 * 32 =  0
1 * 16 = 16
1 *  8 =  8
0 *  4 =  0
0 *  2 =  0
0 *  1 =  0
    Σ = 88
```

Почему именно основание 2

Система счисления с основанием 2 более всего соответствует способу представления информации в компьютере. На самом деле компьютеры "понятия не имеют" ни о каких буквах, цифрах, командах или программах, поскольку представляют собой обычные электронные схемы, для которых важны только такие понятия, как сила тока и напряжение в сети.

Чтобы не усложнять себе жизнь измерениями относительной силы тока в сети (малая, меньше средней, средняя, больше средней и т.д.), разработчики первых компьютерных систем сошлись на том, что проще и надежнее отслеживать только два состояния: есть ток — нет тока. Эти состояния можно выразить словами "да" и "нет", или "истинно" и "ложно", или цифрами 1 и 0. По соглашению 1 означает истинно или "да", но это всего лишь соглашение. С таким же успехом единица могла бы означать ложно или "нет".

Теперь легко понять, почему двоичная система счисления так пришлась по душе разработчикам компьютерных систем. Последовательностями нулей и единиц, соответствующих отсутствию и наличию импульса тока в сети, можно кодировать и передавать любую информацию, подобно тому как точками и тире кодируются буквы в азбуке Морзе.

Биты, байты и полубайты

Если мы приняли решение кодировать данные последовательностями единиц и нулей, то минимальной единицей информации будет двоичный разряд (или бит). На заре компьютерной эры информация передавалась порциями по 8 битов, поэтому минимальной смысловой единицей (словом) в программировании было 8-разрядное число, называемое байтом.

ПРИМЕЧАНИЕ

Половина байта (4 бита) называется полубайтом (nybble).

С помощью восьми двоичных разрядов можно представить до 256 различных значений. Почему? Рассмотрим разрядные позиции. Если все восемь разрядов установлены (равны 1), то полученное двоичное число будет соответствовать десятичному 255. Если не установлен ни один из разрядов, значение равно 0, т.е. в диапазоне 0–255 укладываются 256 возможных значений.

Что такое килобайт

Оказывается, что 2^{10} (1 024) приблизительно равно 10^3 (1 000). Это совпадение грешно было бы не использовать, поэтому ученые компьютерщики 2^{10} байтов начали называть 1 килобайтом (1 Кбайт), используя префикс "кило", который в переводе с латинского означает тысяча.

Аналогично и число 1024 * 1024 (1 048 576) не намного отличается от миллиона, в результате в компьютерной среде широко используется обозначение 1 Мбайт (или 1 мегабайт), а 1 024 мегабайта называют 1 гигабайтом ("гига" означает тысячу миллионов, или миллиард).

Двоичные числа

В компьютерах используются наборы из единиц и нулей для кодирования любой информации. Программы на машинном языке также кодируются как наборы из единиц и нулей и интерпретируются центральным процессором. Специалист по компьютерным системам мог бы вручную перекодировать последовательность единиц и нулей в строку десятичных чисел, но код от этого не станет понятнее с точки зрения человеческой логики.

Например, микросхема Intel 80.6 интерпретирует битовый набор 1001 0101 как команду. В десятичном представлении это значение соответствует числу 149, что для человека, не сведущего в механизмах работы процессора, также ни о чем не говорит.

Иногда числа представляют собой команды, иногда — значения, а иногда — структурные элементы кода. Одним из стандартизованных наборов кодов является ASCII. В нем каждая буква или знак препинания имеет 7-разрядное двоичное представление. Например, строчная буква "а" представляется двоичным числом 0110 0001. Хотя это значение можно преобразовать в десятичное число 97 (64 + 32 + 1), следует понимать, что это не цифра, а буква. Поэтому иногда говорят, что буква "а" в ASCII представлена числом 97, хотя на самом деле двоичное представление десятичного числа 97 (0110 0001) является кодом буквы "а".

Шестнадцатеричная система счисления

Поскольку двоичная система громоздка и трудна для понимания, для упрощения манипулирования с данными было бы полезно иметь возможность быстро и динамично приводить двоичные значения к числам с большим основанием. Оказалось, что преобразовывать двоичные значения к числам шестнадцатеричной системы счисления намного проще и быстрее, чем к десятичным числам. Почему? Давайте сначала рассмотрим, что представляют собой шестнадцатеричные числа.

Для представления шестнадцатеричных чисел используется 16 символов: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E и F. Как видите, последние шесть символов — не цифры, а буквы. Буквы A–F выбраны произвольно, просто как первые буквы латинского алфавита. Вот чему равны граничные значения в шестнадцатеричной системе счисления:

4	3	2	1
16^3	16^2	16^1	16^0
4096	256	16	1

При переводе шестнадцатеричного числа в десятичное можно использовать описанную выше схему (вычислить сумму произведений цифр числа на граничные значения соответствующих порядков). Возьмем, например, число F8C:

```
F * 256 = 15 * 256 =     3840
8 * 16 =                          128
C * 1 = 12 * 1 =           12
3840 + 128 + 1 =             3980
```

Перевод числа FC в двоичное число лучше всего делать путем первоначального перевода в десятичное, а затем уже в двоичное:

```
F * 16 = 15 * 16 =     240
C * 1 = 12 * 1 =         12
240 + 12 =               252
```

Преобразование числа 252_{10} в двоичное представление показано в следующей таблице:

Разряд	9	8	7	6	5	4	3	2	1
Степень	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Значение	256	128	64	32	16	8	4	2	1

```
256 не укладывается ни разу.
1    раз 128     остаток 124
1    раз 64        остаток 60
1    раз 32        остаток 28
1    раз 16        остаток 12
1    раз 8            остаток 4
1    раз 4            остаток 0
0    раз 2
0    раз 1
1    1    1    1    1    1    0    0
```

Таким образом, мы получили двоичное число 1111 1100.

Теперь оказывается, что, представив это число как два набора, состоящих из четырех цифр, мы можем сделать одно магическое превращение.

Правый набор представляет собой число 1100. В десятичном выражении это число 12, а в шестнадцатеричном — число C.

Левый набор имеет вид 1111, который по основанию 10 представляется как число 15, а по основанию 16 — как число F.

Итак, мы получили следующее:

```
1111      0000
F           C
```

Расположив два шестнадцатеричных числа вместе, получаем число FC, которое равно настоящему значению 1111 1100. Этот быстрый метод преобразования работает всегда безотказно. Вы можете взять любое двоичное число любой длины, разбить его на группы по четыре разряда, перевести каждую группу в шестнадцатеричную цифру и расположить эти цифры вместе, чтобы получить шестнадцатеричное число. Вот другой пример:

```
1011 0001 1101 0111
```

Напомню, что в двоичной системе используются следующие граничные значения порядков: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 и 32768.

```
1 × 1 =          1
1 × 2 =          2
1 × 4 =          4
0 × 8 =          0

1 × 16 =         16
1 × 32 =          0
1 × 64 =         64
1 × 128 =         128

1 × 256 =        256
1 × 512 =         0
1 × 1024 =          0
1 × 2048 =            0

1 × 4096 =       4 096
1 × 8192 =        8 192
1 × 16384 =           0
1 × 32768 =    32 768
Итого:         45 527
```

Для преобразования этого числа в шестнадцатеричное вспомним граничные значения порядков в этой системе счислений:

```
65536        4096        256        16        1
```

Число 65 536 (значение пятого порядка) не укладывается в числе 45 527, в пятой позиции ставим 0. Число 4096 (значение четвертого порядка) укладывается в числе 45 527 одиннадцать раз с остатком 471. В остатке 471 число 256 (значение третьего порядка) укладывается один раз с остатком 215. В новом остатке 215 число 16 (значение второго порядка) укладывается 13 раз с остатком 7. Таким образом, получаем шестнадцатеричное число B1D7.

Проверим наши математические выкладки:

```
B     (11) * 4096 =        45 056
1          *  256 =           256
D    (13) * 16    =           208
7          * 1     =           7
Всего          45 527
```

Для проверки ускоренного метода перевода возьмем двоичное число 1011000111010111, разделим его на группы по четыре знака: 1011 0001 1101 0111. Каждая из четырех групп затем преобразуется в шестнадцатеричное число:

```
1011 =
1 × 1 = 1
1 × 2 = 2
0 × 4 = 0
1 × 8 = 8
Всего:                    11
Шестнадцатеричное:    B

0001 =
1 × 1 = 1
0 × 2 = 0
0 × 4 = 0
0 × 8 = 0
Всего:                    1
Шестнадцатеричное:    1

1101 =
1 × 1 = 1
0 × 2 = 0
1 × 4 = 4
1 × 8 = 8
Всего:                    13
Шестнадцатеричное:    D

0111 =
1 × 1 = 1
1 × 2 = 2
1 × 4 = 4
0 × 8 = 0
Всего:                    7
Шестнадцатеричное:    7
```

Итак, получаем шестнадцатеричное число B1D7.

День 1

Контрольные вопросы

1. В чем разница между интерпретаторами и компиляторами?

   Интерпретаторы считывают исходный код программы строка за строкой и преобразуют его непосредственно в машинные команды. Компиляторы преобразуют код источника в исполняемую программу, которая может быть выполнена в более позднее время.

2. Как происходит компиляция исходного кода программы?

   Каждый компилятор имеет свои особенности. Обязательно ознакомьтесь с документацией, которая прилагается к вашему компилятору.

3. В чем состоит назначение компоновщика?

   Задача компоновщика — связать скомпилированный код программы с элементами, взятыми из стандартных библиотек, предоставленных фирмой — изготовителем компилятора, и другими источниками. Компоновщик позволяет формировать отдельные модули программы, а затем соединять эти части в одну большую программу.

4. Какова обычная последовательность действий в цикле разработки?

   Редактирование исходного кода, компиляция, компоновка, тестирование, повторение перечисленных выше действий.

Упражнения

1. Инициализирует две целочисленные переменные, а затем выводит их сумму и произведение.
2. Ознакомьтесь с документацией, которая прилагается к вашему компилятору.
3. В первой строке перед словом `include` нужно поместить символ `#`.
4. Эта программа выводит на экран слова `Hello World`, которые завершаются символом разрыва строки.

День 2

Контрольные вопросы

1. В чем разница между компилятором и препроцессором?

   При каждом запуске компилятора первым запускается препроцессор. Он читает исходный код и включает указанные вами файлы, а также выполняет другие вспомогательные операции. Подробно функции препроцессора рассматриваются на занятии 18.

2. В чем состоит особенность функции `main()`?

   Функция `main()` вызывается автоматически при каждом выполнении программы.

3. Какие два типа комментариев вы знаете и чем они отличаются друг от друга?

   Строки комментариев в стиле C++ задаются двумя символами слеша (//), которые блокируют любой текст до конца текущей строки. Комментарии в стиле языка C заключаются в пары символов (/* */), и все, что находится между этими символами, блокируется от выполнения компилятором. Следует внимательно отнестись к использованию пар символов комментариев, чтобы не заблокировать целый блок программы.

4. Могут ли комментарии быть вложенными?

   Да, комментарии в стиле C++ могут быть вложены внутрь комментариев в стиле C. Можно также комментарии в стиле C вкладывать внутрь комментариев в стиле C++, но при этом следует помнить, что комментарии в стиле C++ заканчиваются в конце текущей строки.

5. Могут ли комментарии занимать несколько строк?

   Это позволено лишь комментариям в стиле C. Если же вы хотите продолжить на следующей строке комментарии в стиле C++, необходимо поставить в начале второй строки еще одну пару символов (//).

Упражнения

1. Напишите программу, которая выводит на экран сообщение `I love C++`.

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:    cout << "I love C++\n";
6:    return 0;
7: }
```

2. Напишите самую маленькую программу, которую можно скомпилировать, скомпоновать и выполнить.

```
int main() { returna 0; }
```

3. **Жучки:** введите эту программу и скомпилируйте ее. Почему она дает сбой? Как ее можно исправить?

```
1: #include <iostream.h>
2: int main()
3: {
4:   cout << Is there a bug here?";
5:   return 0;
6: }
```

В строке 4 пропущена открывающая кавычка для строкового выражения.

4. Исправьте ошибку в упражнении 3, после чего перекомпилируйте ее, скомпонуйте и запустите на выполнение.

```
1: #include <iostream.h>
2: int main()
3: {
4:   cout << " Is there a bug here?";
5:   return 0;
6: }
```

День 3

Контрольные вопросы

1. В чем разница между целочисленной и вещественной (с плавающей точкой) переменными?

   Целочисленные переменные предназначены для работы с целыми числами, а вещественные — с вещественными числами, содержащими плавающую десятичную точку. Числа с плавающей точкой могут быть представлены с использованием мантиссы и экспоненты.

2. Каково различие между типами `unsigned short int` и `long int`?

   Ключевое слово `unsigned` означает, что данная целочисленная переменная будет содержать только положительные числа. В большинстве компьютеров для коротких (`short`) целых чисел отводится 2 байта, а для длинных (`long`) — 4 байта.

3. Каковы преимущества использования символьной константы вместо литерала?

   Символьная константа "говорит сама за себя", т.е. само имя константы указывает на ее назначение. Кроме того, при внесении изменений символьную константу достаточно переопределить в одной строке исходного кода, в то время как при использовании литералов программисту придется редактировать код всюду, где встречается этот литерал.

4. Каковы преимущества использования ключевого слова `const` вместо директивы `#define`?

   Константы, объявленные с помощью ключевого слова `const`, используются с контролем за соответствием типа, поэтому компилятор сможет заметить ошибку в случае неправильного определения или применения такой константы. Кроме того, поскольку эти константы остаются в программе после ее обработки препроцессором, они доступны отладчику.

5. Как влияет на работу программы "хорошее" или "плохое" имя переменной?

   Хорошее имя переменной говорит о назначении этой переменной; плохое не несет никакой информации. Например, `MyAge` (МойВозраст) и `PeopleOnTheBus` (ПассажировВАвтобусе) — это хорошие имена переменных, а в таких именах, как `xjk` и `prndl`, вероятно, мало пользы.

6. Если перечисление (`enum`) заданно таким образом, то каково значение его члена `Blue`?

```
enum COLOR { WHITE, BLACK = 100, RED, BLUE, GREEN = 300 };
BLUE = 102
```

7. Какие из следующих имен переменных являются хорошими, плохими и вообще недопустимыми?

   а) `Age` — хорошее имя;

   б) `!ex` — недопустимое имя;

   в) `R79J` — допустимое, но неудачное имя;

   г) `TotalIncome` — хорошее имя;

   д) `_Invalid` — допустимое, но неудачное имя.

Упражнения

1. Какой тип переменной был бы правильным для хранения следующей информации?

   - Ваш возраст.
     `Unsigned short integer`
   - Площадь вашего заднего двора.
     `Unsigned long integer` или `unsigned float`
   - Количество звезд в галактике.
     `Unsigned double`
   - Средний уровень выпадения осадков за январь месяц.
     `Unsigned short integer`

2. Создайте подходящие имена переменных для хранения этой информации.

   а) `myAge` (МойВозраст)

   б) `backYardArea` (ПлощадьЗаднегоДвора)

   в) `StarsInGalaxy` (ЗвездВГалактике)

   г) `averageRainFall` (СреднемесячныйУровеньОсадков)

3. Объявите константу для числа pi, равного 3.14159.

```
const float pi = 3.14159;
```

4. Объявите переменную типа `float` и инициализируйте ее, используя константу `pi`.

```
float myPi = PI;
```

День 4

Контрольные вопросы

1. Что такое выражение?

   Это любой набор операторов, возвращающий значение.

2. Является ли запись `x = 5 + 7` выражением? Каково его значение?

   Да, является и возвращает значение 12.

3. Каково значение выражения `201 / 4`?

   50

4. Каково значение выражения `201 % 4`?

   1

5. Если переменные `myAge`, `a` и `b` имеют тип `int`, то каковы будут их значения после выполнения выражения:

```
myAge = 39;
a = myAge++;
b = ++myAge;
niyAge: 41, a: 39, b: 41
```

6. Каково значение выражения `8+2*3`?

   14

7. Какая разница между выражениями `if(x = 3)` и `if(x == 3)`?

   Первое выражение присваивает переменной `x` значение `3` и возвращает `TRUE`. Второе выражение проверяет, равно ли значение переменной `x` числу 3, и возвращает `TRUE`, если значение переменной `x` равно 3, и `FALSE` в противном случае.

8. Будут ли следующие выражения возвращать `true` или `false`?

   а) `0`

```
FALSE
```

   б) `1`

```
TRUE
```

   в) `-1`

```
TRUE
```

   г) `x = 0`

```
FALSE
```

   д) `x == 0 // предположим, что x имеет значение 0`

```
TRUE
```

Упражнения

1. Напишите один оператор if, который проверяет две целочисленные переменные и присваивает переменной с бо́льшим значением меньшее значение, используя только один дополнительный оператор else.

```
if (x > y)
    x = y;
else          // y > x || y == x
    y = x;
```

2. Проанализируйте следующую программу. Представьте, что вы ввели три значения. Какой результат вы ожидаете получить?

```
1: #include <iostream.h>
2: int main()
3: {
4:  int a, b, c;
5:  cout << "Please enter three numbers\ n";
6:  cout << "a:  ";
7:  cin >> a;
8:  cout << "\ nb:  ";
9:  cin >> b;
10:  cout << "\ nc:  ";
11:  cin >> c;
12:
13:  if (c = (a-b))
14:  { cout << "a:  ";
15:  cout << a;
16:  cout << "minus b:  ";
17:  cout << b;
18:  cout << "equals c:  ";
19:  cout << c << endl;}
20:  else
21:  cout << "a-b does not equal c:  " << endl;
22:  return 0;
23: }
```

3. Введите программу из упражнения 2; скомпилируйте, скомпонуйте и выполните ее. Введите числа 20, 10 и 50. Вы получали результат, который и ожидали? Почему нет?

   Введите числа 20, 10, 50. А вот результат:

   a: 20 minus b: 10 equals c: 10

   Неожиданный результат? Дело в том, что в строке 13 выполняется присваивание, а не проверка равенства.

4. Проанализируйте эту программу и спрогнозируйте результат:

```
1:   #include <iostream.h>
2:   int main()
3:   {
4:        int a = 1, b = 1, c;
5:        if (c = (a-b))
```

```
6:            cout << "The value of c is: " << c;
7:    return 0;
8:    }
```

5. Введите, скомпилируйте, скомпонуйте и выполните программу из упражнения 4. Каков был результат? Почему?
6. Поскольку в строке 5 переменной `c` присваивается значение `a - b`, то значение присваивания выражения `a` (`1`) минус `b` (`1`) равно `0`. Поскольку `0` означает `false` (ложь), то выходит, что условие проверки не выполняется и поэтому ничего не выводится.

День 5

Контрольные вопросы

1. В чем разница между объявлением прототипа функции и определением функции?

   В прототип функции объявляются список формальных параметров и тип возврата функции, а выполнение функции задается ее определением. Символ точки с запятой завершает прототип функции, но не ее определение. Объявление может включать ключевое слово `inline` и установки значений параметров по умолчанию. В объявлении функции достаточно указать типы параметров, а определение должно содержать их имена.
2. Должны ли имена параметров, указанные в прототипе, определении и вызове функции соответствовать друг другу?

   Нет. Все параметры идентифицируются позицией, а не по имени.
3. Если функция не возвращает значение, как следует объявить такую функцию?

   Для возврата функции следует установить тип `void`.
4. Если не объявить тип возврата, то какой тип будет принят по умолчанию для возвращаемого значения?

   Любая функция, в которой явно не объявляется тип возвращаемого значения, возвращает значение типа `int`.
5. Что такое локальная переменная?

   Это переменная, передаваемая или объявленная внутри некоторого блока (обычно функции). Она видима только в пределах этого блока.
6. Что такое область видимости?

   Область видимости определяет "продолжительность жизни" локальных и глобальных переменных и обычно устанавливается набором фигурных скобок.
7. Что такое рекурсия?

   В общем случае это способность функции вызывать самое себя.
8. Когда следует использовать глобальные переменные?

   Глобальные переменные обычно используются, когда многим функциям нужен доступ к одним и тем же данным. В C++ глобальные переменные используются очень редко. Как только вы научитесь создавать статические переменные класса, вы практически не будете обращаться к глобальным переменным.

9. Что такое перегрузка функции?

Это способность записать несколько функций с одним и тем же именем, но с различным числом или типом параметров.

10. Что такое полиморфизм?

Это возможность вызова одноименных методов для объектов разных, но взаимосвязанных типов с учетом различий в выполнении функции для разных типов. В C++ полиморфизм реализуется путем создания производных классов и виртуальных функций.

Упражнения

1. Запишите прототип для функции с именем `Perimeter`, которая возвращает значение типа `unsigned long int` и принимает два параметра, имеющих тип `unsigned short int`.

```
unsigned long int Perimeter(unsigned short int, unsigned short int);
```

2. Запишите определение функции `Perimeter` согласно объявлению в упражнении 1. Два принимаемых ею параметра представляют длину и ширину прямоугольника, а функция возвращает его периметр (удвоенная длина плюс удвоенная ширина).

```
unsigned long int Perimeter(unsigned short int length, unsigned short int width)
{
    return 2*length + 2*width;
}
```

3. **Жучки:** что неправильно в этой функции?

```
#include <iostream.h>
void myFunc(unsigned short int x);
int main()
{
unsigned short int x, y;
y = myFunc(int);
cout << "x: " << x << " y: " << y << "\n";
}

void myFunc(unsigned short int x)
{
return (4*x);
}
```

Функция объявлена с использованием ключевого слова `void`, и поэтому она не может возвращать какое-либо значение. Кроме того, при вызове функции `myFunc` ей следует передать параметр `x`, а не `int`.

4. **Жучки:** что неправильно в этой функции?

```
#include <iostream.h>
int myFunc(unsigned short int x);
int main()
{
unsigned short int x, y;
y = myFunc(x);
```

```
cout << "x: " << x << " y: " << y << "\n";
}

int myFunc(unsigned short int x);
{
return (4*x);
}
```

Эта функция была бы идеальной, если бы не точка с запятой, поставленная в конце заголовка ее определения.

5. Напишите функцию, которая принимает два параметра типа `unsigned short int` и возвращает результат деления первого параметра на второй. Функция не должна выполнять операцию деления, если второе число равно нулю, но в этом случае она должна возвратить значение -1.

```
short int Divider(unsigned short int valOne, unsigned short int valTwo)
{
    if (valTwo == 0)
        return -1;
    else
        return valOne / valTwo;
}
```

6. Напишите программу, которая запрашивает у пользователя два числа и вызывает функцию, записанную при выполнении упражнения 5. Выведите результат или сообщение об ошибке, если функция возвратит значение, равное -1.

```
#include <iostream.h>
typedef unsigned short int USHORT;
typedef unsigned long int ULONG;
short int Divider(
unsigned short int valone,
unsigned short int valtwo);
int main()
{
    USHORT one, two;
    short int answer;
    cout << "Enter two numbers.\n Number one: ";
    cin >> one;
    cout << "Number two: ";
    cin >> two;
    answer = Divider(one, two);
    if (answer > -1)
        cout << "Answer: " << answer;
    else
    cout << "Error, can't divide by zero!";
return 0;
}
```

7. Напишите программу, которая запрашивает число и показатель степени. Напишите рекурсивную функцию, которая возводит число в степень путем многократного умножения числа на самое себя, т.е. если число равно 2, а показатель степени равен 4, то эта функция должна возвратить число 16.

```
#include <iostream.h>
typedef unsigned short USHORT;
typedef unsigned long ULONG;
ULONG GetPower(USHORT n, USHORT power);
int main()
{
    USHORT number, power;
    ULONG answer;
    cout << "Enter a number: ";
    cin >> number;
    cout << "To what power? ";
    cin >> power;
    answer = GetPower(number,power);
    cout << number << " to the " << power << "th power is " << answer << endl;
return 0;
}
ULONG GetPower(USHORT n, USHORT power)
{
    if(power == 1)
        return n;
    else
        return (n * GetPower(n,power-1));
}
```

День 6

Контрольные вопросы

1. Что представляет собой оператор прямого доступа и для чего он используется?

   Оператор точки прямого доступа представляет собой символ точки (.). Он используется для обращения к членам класса.

2. Что резервирует память — объявление или определение?

   Память резервируется определениями переменных. Объявления классов не резервируют память.

3. Объявление класса является его интерфейсом или выполнением?

   Объявление класса является его интерфейсом, который сообщает клиентам класса, как с ним взаимодействовать. Выполнение класса — это набор функций-членов, сохраняемых обычно в файле с расширением CPP.

4. Какова разница между открытыми (public) и закрытыми (private) данными-членами?

   К открытым переменным-членам могут обращаться клиенты класса, а к закрытым могут получить доступ только функции-члены класса.

5. Могут ли функции-члены быть закрытыми?

   Да. Как функции-члены, так и переменные-члены могут быть закрытыми.

6. Могут ли переменные-члены быть открытыми?

Хотя переменные-члены могут быть открытыми, но считается хорошей практикой программирования, когда переменные-члены объявляются все же закрытыми, а доступ к этим данным обеспечивается за счет открытых методов доступа.

7. Если объявить два объекта класса `Cat`, могут ли они иметь различные значения своих переменных-членов `itsAge`?

   Да. Каждый объект класса имеет свои собственные переменные-члены.

8. *Нужно ли объявления класса завершать точкой с запятой? А определения методов класса?*

   Объявления класса заканчиваются точкой с запятой после закрывающей фигурной скобки, а определения функций-членов — нет.

9. Как бы выглядел заголовок функции-члена `Meow` класса `Cat`, которая не принимает никаких параметров и возвращает значение типа `void`?

   Заголовок функции-члена `Meow()` класса `Cat`, которая не принимает параметров и возвращает значение типа `void`, должен иметь следующий вид:

   ```
   void Cat::Meow()
   ```

10. Какая функция вызывается для выполнения инициализации класса?

    Для инициализации класса вызывается конструктор.

Упражнения

1. Напишите программу, которая объявляет класс с именем `Employee` (Служащие) с такими переменными-членами: `age` (возраст), `yearsOfService` (стаж работы) и `Salary` (зарплата).

```
class Employee
{
    int Age;
    int YearsOfService;
    int Salary;
};
```

2. Перепишите класс `Employee`, чтобы сделать данные-члены закрытыми и обеспечить открытые методы доступа для чтения и установки всех данных-членов.

```
class Employee
{
public:
    int GetAge() const;
    void SetAge(int age);
    int GetYearsOfService() const;
    void SetYearsOfService(int years);
    int GetSalary() const;
    void SetSalary(int salary);

private:
    int Age;
    int YearsOfService;
    int Salary;
};
```

3. Напишите программу с использованием класса Employee, которая создает два объекта класса Employee; устанавливает данные-члены age, YearsOfService и Salary, а затем выводит их значения.

```
int main()
{
    Employee John;
    Employee Sally;
    John.SetAge(30);
    John.SetYearsOfService(5);
    John.SetSalary(50000);
    Sally.SetAge(32);
    Sally.SetYearsOfService(8);
    Sally.SetSalary(40000);
    cout << "At AcmeSexist company, John and Sally have the same job.\n";
    cout << "John is " << John.GetAge() << " years old and he has been with";
    cout << "the firm for " << John.GetYearsOfService << " years.\n";
    cout << "John earns $" << John.GetSalary << " dollars per year.\n\n";
    cout << "Sally, on the other hand is " << Sally.GetAge() << " years old and has";
    cout << "been with the company " << Sally.GetYearsOfService;
    cout << " years. Yet Sally only makes $" << Sally.GetSalary();
    cout << " dollars per year! Something here is unfair.";
    return 0;
}
```

4. На основе программы из упражнения 3 создайте метод класса Employee, который сообщает, сколько тысяч долларов зарабатывает служащий, округляя ответ до 1 000 долларов.

```
float Employee:GetRoundedThousands() const
{
    return (Salary+500) / 1000;
}
```

5. Измените класс Employee так, чтобы можно было инициализировать данные-члены age, YearsOfService и Salary в процессе "создания" служащего.

```
class Employee
{
public:

    Employee(int Age, int yearsOfService, int salary);
    int GetAge() const;
    void SetAge(int Age);
    int GetYearsOfService() const;
    void SetYearsOfService(int years);
    int GetSalary() const;
    void SetSalary(int salary);

private:
```

```
    int Age;
    int YearsOfService;
    int Salary;
};
```

6. **Жучки:** что неправильно в следующем объявлении?

```
class Square
{
public:
    int Side;
}
```

Объявления классов должны завершаться точкой с запятой.

7. **Жучки:** что весьма полезное отсутствует в следующем объявлении класса?

```
class Cat
{
    int GetAge() const;
private:
    int itsAge;
};
```

Метод доступа к данным `GetAge()` *является закрытым по умолчанию. Помните:* все члены класса считаются закрытыми, если не оговорено иначе.

8. **Жучки:** какие три ошибки обнаружит компилятор в этом коде?

```
class TV
{
public:
    void SetStation(int Station);
    int GetStation() const;
private:
    int itsStation;
};
main()
{
    TV myTV;
    myTV.itsStation = 9;
    TV.SetStation(10);
    TV myOtherTv(2);
}
```

Нельзя обращаться к переменной `itsStation` непосредственно. Это закрытая переменная-член.

Нельзя вызывать функцию-член `SetStation()`прямо в классе. Метод `SetStation()` можно вызывать только для объекта.

Нельзя инициализировать переменную-член `itsStation`, поскольку в программе не определен нужный для этого конструктор.

День 7

Контрольные вопросы

1. Можно ли в цикле `for` инициализировать сразу несколько переменных-счетчиков?

   Можно, для этого в строке инициализации их следует разделить запятыми, например:

   ```
   for (x = 0, y = 10; x < 100; x++, y++)
   ```

2. Почему следует избегать использование оператора `goto`?

   Оператор `Goto` выполняет переход в любом направлении к любой произвольной строке программы. Это делает исходный код слишком запутанным, а следовательно, и трудным для понимания и дальнейшего обслуживания.

3. Можно ли с помощью оператора `for` организовать цикл, тело которого не будет выполняться?

   Да. Если условие после инициализации оказывается ложным (`FALSE`), то тело цикла `for` никогда не будет выполняться. Вот пример:

   ```
   for (int x = 100; x < 100; x++)
   ```

4. Можно ли организовать цикл `while` внутри цикла `for`?

   Да. Любой цикл может быть вложен внутрь любого другого цикла.

5. Можно ли организовать цикл, который никогда не завершится? Приведите пример.

   Да. Ниже приведены примеры как для цикла `for`, так и для цикла `while`:

   ```
   for(;;)
   {
       // Этот цикл никогда не заканчивается!
   }
   while(1)
   {
       // Этот цикл никогда не заканчивается!
   }
   ```

6. Что происходит при запуске бесконечного цикла?

   Программа зависнет и придется перезагрузить компьютер.

Упражнения

1. Каким будет значение переменной `x` после завершения цикла `for (int x = 0; x < 100; x++)`?

   100

2. Создайте вложенный цикл `for`, заполняющий нулями массив размером 10×10.

   ```
   for (int  = 0; i< 10; i++)
   {
       for ( int j = 0; j< 10; j++)
   ```

```
        cout << "0";
    cout << "\n";
}
```

3. Организуйте цикл `for`, счетчик которого изменяется от 100 до 200 с шагом 2.

```
for (int x = 100; x<=200; x+=2)
```

4. Организуйте цикл `while`, счетчик которого изменяется от 100 до 200 с шагом 2.

```
int x = 100;
while (x <= 200)
    x+= 2;
```

5. Организуйте цикл `do...while`, счетчик которого изменяется от 100 до 200 с шагом 2.

```
int x = 100;
do
{
    x+=2;
} while (x <= 200);
```

6. **Жучки:** найдите ошибку в приведенном фрагменте программы:

```
int counter = 0;
while (counter < 10)
{
    cout << "counter: " << counter;
}
```

Нет выражения, в котором выполнялось бы приращение счетчика `counter`, поэтому цикл `while` никогда не закончится.

7. **Жучки:** найдите ошибку в приведенном фрагменте программы:

```
for (int counter = 0; counter < 10; counter++);
    cout << counter << " ";
```

В конце строки задания цикла стоит точка с запятой, поэтому цикл выполняет только приращение счетчика. Программист, возможно, именно это и имел в виду, но если предполагался еще и вывод каждого значения переменной `counter`, то этого не произойдет.

8. **Жучки:** найдите ошибку в приведенном фрагменте программы:

```
int counter = 100;
while (counter < 10)
{
    cout << "counter now: " << counter;
    counter--;
}
```

Счетчик `counter` инициализирован числом 100, но проверяемое условие таково, что, если значение переменной `counter` больше 10, выражение условия возвратит `FALSE` и тело цикла никогда не будет выполнено. Если первую строку заменить вариантом `int counter = 5;`, то этот цикл не закончится до тех пор, пока не выполнится обратный отсчет до минимально возможного значения счетчика. Поскольку *тип счетчика* `int` *по умолчанию определяется как* `signed`, то мы получим бесконечный цикл.

9. **Жучки:** найдите ошибку в приведенном фрагменте программы:

```
cout << "Enter a number between 0 and 5: ";
cin >> theNumber;
switch (theNumber)
{
     case 0:
          doZero();
     case 1:      // идем дальше
     case 2:      // идем дальше
     case 3:      // идем дальше
     case 4:      // идем дальше
     case 5:
          doOneToFive();
          break;
     default:
          doDefault();
          break;
}
```

После оператора `case 0`, видимо, должен быть оператор `break`. Если это не так, то ситуацию следовало бы разъяснить с помощью комментария.

День 8

Контрольные вопросы

1. Какой оператор используется для получения адреса переменной?

   Для возвращения адреса любой переменной используется оператор получения адреса (&).

2. Какой оператор позволяет получить значение, записанное по адресу, содержащемуся в указателе?

   Для доступа к значению, сохраненному по адресу, содержащемуся в указателе, используется оператор разыменования (*).

3. Что такое указатель?

   Это переменная, которая содержит адрес другой переменной.

4. В чем различие между адресом, который хранится в указателе, и значением, записанным по этому адресу?

   Адрес, сохраненный в указателе, — это адрес другой переменной. Значение, сохраненное по этому адресу, — это любое значение, сохраняемое в переменной, на которую ссылается указатель. Оператор разыменования (*) возвращает значение, сохраненное по адресу, который хранится в указателе.

5. В чем различие между оператором разыменования и оператором получения адреса?

   Оператор разыменования (*) возвращает значение, хранящееся по адресу, на который ссылается указатель. А оператор получения адреса (&) возвращает адрес переменной в памяти.

6. В чем различие между следующими объявлениями: `const int * ptrOne` и `int * const ptrTwo`?

   Выражение `const int * ptrOne` объявляет, что переменная `ptrOne` представляет собой указатель на постоянное целое число. Само это целое число не может быть изменено с помощью данного указателя.

   Выражение `int * const ptrTwo` объявляет, что переменная `ptrTwo` является постоянным указателем на некоторое целое число. После такой инициализации этот указатель не может быть переназначен.

Упражнения

1. Объясните смысл следующих объявлений переменных.
   - `int * pOne;`
   - `int vTwo;`
   - `int * pThree = &vTwo;`

   Ответы:

   а) `int * pOne;` — объявляет указатель на целое значение;

   б) `int vTwo;` — объявляет целочисленную переменную;

   в) `int * pThree = &vTwo;` — объявляет указатель на целое значение и инициализирует его адресом переменной.

2. Допустим, в программе объявлена переменная `yourAge` типа `unsigned short`. Как объявить указатель, позволяющий манипулировать этой переменной?

```
unsigned short *pAge = &yourAge;
```

3. С помощью указателя присвойте переменной `yourAge` значение 50.

```
*pAge =50;
```

4. Напишите небольшую программу и объявите в ней переменную типа `int` и указатель на этот тип. Сохраните адрес переменной в указателе. Используя указатель, присвойте переменной какое-либо значение.

```
int theInteger;
int *pInteger = &theInteger;
*pInteger = 5;
```

5. **Жучки:** *найдите ошибку в следующем фрагменте программы:*

```
#include <iostream.h>
int main()
{
     int *pInt;
     *pInt = 9;
     cout << " The value at pInt: " << *pInt;
     return 0;
}
```

   Указатель `pInt` должен быть инициализирован. Поскольку он не был инициализирован и ему не присвоен адрес какой-либо ячейки памяти, то он указывает на случайное место в памяти. Присвоение этому случайному месту числа 9 является опасной ошибкой.

6. **Жучки:** найдите ошибку в следующем фрагменте программы:

```
int main()
{
    int SomeVariable = 5;
    cout << "SomeVariable: " << SomeVariable << "\ n";
    int *pVar = & SomeVariable;
    pVar = 9;
    cout << "SomeVariable: " << *pVar << "\ n";
return 0;
}
```

Возможно, программист хотел присвоить число 9 переменной, на которую указывает указатель pVar. К сожалению, число 9 было присвоено самому указателю pVar, поскольку был опущен оператор косвенного доступа (*). Если указатель pVar используется для присвоения ему значения, такое программирование неминуемо приведет к тяжелым последствиям.

День 9

Контрольные вопросы

1. В чем разница между ссылкой и указателем?

   Ссылка — это условное название (псевдоним), а указатель — это переменная, которая содержит адрес. Ссылки не могут быть нулевыми и не могут переназначаться.

2. Когда нужно использовать именно указатель, а не ссылку?

   Если в программе нужно назначить указателю новую переменную или если указатель нужно сделать нулевым.

3. Что возвращает оператор new, если для создания нового объекта недостаточно памяти?

   Нулевой указатель.

4. Что представляет собой константная ссылка?

   Это сокращенный вариант определения ссылки на константный объект.

5. В чем разница между передачей объекта как ссылки и передачей ссылки в функцию?

   Передача объекта как ссылки означает, что локальная копия для этого объекта создаваться не будет. Этого можно достичь путем передачи в качестве параметра ссылки или указателя.

Упражнения

1. Напишите программу, которая объявляет переменную типа int, ссылку на значение типа int и указатель на значение типа int. Используйте указатель и ссылку для управления значением переменной типа int.

```
int main()
{
int varOne;
int& rVar = varOne;
```

```
int* pVar = &varOne;
rVar = 5;
*pVar = 7;
return 0;
}
```

2. Напишите программу, которая объявляет константный указатель на постоянное целое значение. Инициализируйте его, чтобы он указывал на целочисленную переменную varOne. Присвойте переменной varOne значение 6. Используйте указатель, чтобы присвоить переменной varOne значение 7. Создайте вторую целочисленную переменную varTwo. Переназначьте указатель, чтобы он указывал на переменную varTwo. Пока не компилируйте это упражнение.

```
int main()
{
    int varOne;
    const int * const pVar = &varOne;
    *pVar = 7;
    int varTwo;
    pVar = &varTwo;
return 0;
}
```

3. Скомпилируйте программу, написанную в упражнении 2. Какие действия компилятор считает ошибочными? Какие строки генерируют предупреждения?

   Нельзя присваивать значение константному объекту и нельзя переназначать константный указатель.

4. Напишите программу, которая создает блуждающий указатель.

```
int main()
{
int * pVar;
*pVar = 9;
return 0;
}
```

5. Исправьте программу из упражнения 4, чтобы блуждающий указатель стал нулевым.

```
int main()
{
int varOne;
int * pVar = &varOne;
*pVar = 9;
return 0;
}
```

6. Напишите программу, которая приводит к утечке памяти.

```
#include <iostream.h>

int * FuncOne();
int main()
```

```
{
    int * pInt = FuncOne();
    cout << "the value of pInt in main is: " << *pInt << endl;
    return 0;
}

int * FuncOne()
{
    int * pInt = new int (5);
    cout << "the value of pInt in FuncOne is: " << *pInt << endl;
    return pInt;
}
```

7. Исправьте программу из упражнения 6.

```
#include <iostream.h>

int FuncOne();
int main()
{
    int theInt = FuncOne();
    cout << "the value of pInt in main is: " << theInt << endl;
    return 0;
}

int FuncOne()
{
    int * pInt = new int (5);
    cout << "the value of pInt in FuncOne is: " << *pInt << endl;
    delete pInt;
    return temp;
}
```

8. **Жучки:** что неправильно в этой программе?

```
1:    #include <iostream.h>
2:
3:    class CAT
4:    {
5:       public:
6:          CAT(int age) {  itsAge = age; }
7:          ~CAT(){ }
8:          int GetAge() const {  return itsAge;}
9:       private:
10:         int itsAge;
11:   };
12:
```

```
13:    CAT & MakeCat(int age);
14:    int main()
15:    {
16:       int age = 7;
17:       CAT Boots = MakeCat(age);
18:       cout << "Boots is " << Boots.GetAge() << " years old\ n";
19:       return 0;
20:    }
21:
22:    CAT & MakeCat(int age)
23:    {
24:       CAT * pCat = new CAT(age);
25:       return *pCat;
26:    }
```

Функция `MakeCat` возвращает ссылку на объект класса `CAT`, созданный в свободной памяти. Но поскольку здесь не предусмотрена операция по освобождению этой памяти, создание нового объекта приводит к ее утечке.

9. Исправьте программу из упражнения 8.

```
1:     #include <iostream.h>
2:
3:     class CAT
4:     {
5:        public:
6:           CAT(int age) {  itsAge = age; }
7:           ~CAT(){ }
8:           int GetAge() const {  return itsAge;}
9:        private:
10:          int itsAge;
11:    };
12:
13:    CAT * MakeCat(int age);
14:    int main()
15:    {
16:       int age = 7;
17:       CAT * Boots = MakeCat(age);
18:       cout << "Boots is " << Boots.GetAge() << " years old\ n";
19:       delete Boots;
20:    return 0;
21:    }
22:
23:    CAT * MakeCat(int age)
24:    {
25:       return new CAT(age);
26:    }
```

День 10

Контрольные вопросы

1. Если вы перегрузили функцию-член, как потом можно будет различить разные варианты функции?

   Перегруженными называются функции-члены, которые имеют одно и то же имя, но отличаются по количеству или типу параметров.

2. Какая разница между определением и объявлением?

   Определение резервирует память, а объявление — нет. Объявления часто являются и определениями, за исключением объявлений классов, прототипов функций и новых типов с помощью `typedef`.

3. Когда вызывается конструктор-копировщик?

   Всегда, когда создается временная копия объекта. Это случается каждый раз, когда объект передается как значение.

4. Когда вызывается деструктор?

   Деструктор вызывается при удалении объекта либо по причине выхода за пределы области видимости, либо при вызове оператора `delete` для указателя, указывающего на данный объект.

5. Чем отличается конструктор-копировщик от оператора присваивания (`=`)?

   Оператор присваивания работает с существующим объектом, а конструктор-копировщик создает новый временный объект.

6. Что представляет собой указатель `this`?

   Это скрытый параметр в каждой функции-члене, который указывает на сам объект.

7. Как различить перегрузку префиксных и постфиксных операторов приращения?

   Префиксный оператор не принимает никаких параметров. Постфиксный оператор принимает один параметр типа `int`, который используется в качестве флага для компилятора, сообщающего о том, что это постфиксный оператор.

8. Можно ли перегрузить `operator+` для переменных типа `short int`?

   Нет, для встроенных типов нельзя перегружать никаких операторов.

9. Допускается ли в C++ перегрузка `operator++` таким образом, чтобы он выполнял в классе операцию декремента?

   Правомочно, но этого делать не стоит. Операторы следует перегружать таким способом, который должен быть понятен любому читателю вашей программы.

10. Как устанавливается тип возврата в объявлениях функций операторов преобразования типов?

    Никак. Подобно конструкторам и деструкторам, они не имеют никаких возвращаемых значений.

Упражнения

1. Представьте объявление класса SimpleCircle (простая окружность) с единственной переменой-членом itsRadius (радиус). В классе должны использоваться конструктор и деструктор, заданные по умолчанию, а также метод установки радиуса.

```
class SimpleCircle
{
public:
    SimpleCircle();
    ~SimpleCircle();
    void SetRadius(int);
    int GetRadius();
private:
    int itsRadius;
};
```

2. Используя класс, созданный в упражнении 1, с помощью конструктора, заданного по умолчанию, инициализируйте переменную itsRadius значением 5.

```
SimpleCircle::SimpleCircle():
itsRadius(5);
{}
```

3. Добавьте в класс новый конструктор, который присваивает значение своего параметра переменной itsRadius.

```
SimpleCircle::SimpleCircle(int radius):
itsRadius(radius)
{}
```

4. Перегрузите операторы преинкремента и постинкремента для использования в вашем классе SimpleCircle с переменной itsRadius.

```
const SimpleCircle& SimpleCircle::operator++()
{
    ++(itsRadius);
    return *this;
}
// постфиксный оператор Operator ++(int),
// Выборка, затем инкрементирование
const SimpleCircle SimpleCircle::operator++ (int)
// объявляем локальный объект класса SimpleCircle и инициализируем его значением
*this
    SimpleCircle temp(*this);
    ++(itsRadius);
    return temp;
}
```

5. Измените SimpleCircle таким образом, чтобы сохранять itsRadius в динамической области памяти и фиксировать существующие методы.

```
class SimpleCircle
{
public:
    SimpleCircle();
    SimpleCircle(int);
    ~SimpleCircle();
    void SetRadius(int);
    int GetRadius();
    const SimpleCircle& operator++();
    const SimpleCircle operator++(int);
private:
    int *itsRadius;
};

SimpleCircle::SimpleCircle()
{itsRadius = new int(5);}

SimpleCircle::SimpleCircle(int radius)
{itsRadius = new int(radius);}

SimpleCircle::~SimpleCircle()
{
    delete itsRadius;
}

const SimpleCircle& SimpleCircle::operator++()
{
    ++(*itsRadius);
    return *this;
}

// Постфиксный оператор Operator ++(int).
// Выборка, затем инкрементирование
const SimpleCircle SimpleCircle::operator++ (int)
{
// объявляем локальный объект класса SimpleCircle и инициализируем его значением
*this
    SimpleCircle temp(*this);
    ++(*itsRadius);
    return temp;
}
```

6. Создайте в классе `SimpleCircle` **конструктор-копировщик.**

```
SimpleCircle::SimpleCircle(const SimpleCircle & rhs)
{
    int val = rhs.GetRadius();
    itsRadius = new int(val);
}
```

7. Перегрузите в классе SimpleCircle оператор присваивания.

```
SimpleCircle& SimpleCircle::operator=(const SimpleCircle & rhs)
{
    if (this == &rhs)
        return *this;
    delete itsRadius;
    itsRadius = new int;
    *itsRadius = rhs.GetRadius();
    return *this;
}
```

8. Напишите программу, которая создает два объекта класса SimpleCircle. Для создания одного объекта используйте конструктор, заданный по умолчанию, а второму экземпляру при объявлении присвойте значение 9. С каждым из объектов используйте оператор инкремента и выведите полученные значения на печать. Наконец, присвойте значение одного объекта другому объекту и выведите результат на печать.

```
#include <iostream.h>

class SimpleCircle
{
public:
    // конструкторы
    SimpleCircle();
    SimpleCircle(int);
    SimpleCircle(const SimpleCircle &);
    ~SimpleCircle() {}

// методы доступа к данным
    void SetRadius(int);
    int GetRadius() const;

// операторы
    const SimpleCircle& operator++();
    const SimpleCircle operator++(int);
    SimpleCircle& operator=(const SimpleCircle &);

private:
    int *itsRadius;
};

SimpleCircle::SimpleCircle()
{itsRadius = new int(5);}

SimpleCircle::SimpleCircle(int radius)
{itsRadius = new int(radius);}
SimpleCircle::SimpleCircle(const SimpleCircle & rhs)
{
    int val = rhs.GetRadius();
```

```
    itsRadius = new int(val);
}
SimpleCircle::~SimpleCircle()
{
    delete itsRadius;
}
SimpleCircle& SimpleCircle::operator=(const SimpleCircle & rhs)
{
    if (this == &rhs)
        return *this;
    *itsRadius = rhs.GetRadius();
    return *this;
}
          •
const SimpleCircle& SimpleCircle::operator++()
{
    ++(*itsRadius);
    return *this;
}

// Постфиксный оператор Operator ++(int).
// Выборка, затем инкрементирование
const SimpleCircle SimpleCircle::operator++ (int)
{
// объявляем локальный объект класса SimpleCircle и инициализируем его значением
*this
    SimpleCircle temp(*this);
    ++(*itsRadius);
    return temp;
}
int SimpleCircle::GetRadius() const
{
    return *itsRadius;
}
int main()
{
    SimpleCircle CircleOne, CircleTwo(9);
    CircleOne++;
    ++CircleTwo;
    cout << "CircleOne: " << CircleOne.GetRadius() << endl;
    cout << "CircleTwo: " << CircleTwo.GetRadius() << endl;
    CircleOne = CircleTwo;
    cout << "CircleOne: " << CircleOne.GetRadius() << endl;
    cout << "CircleTwo: " << CircleTwo.GetRadius() << endl;
return 0;
}
```

9. **Жучки:** что неправильно в следующем примере использования оператора присваивания?

```
SQUARE SQUARE ::operator=(const SQUARE & rhs)
{
      itsSide = new int;
      *itsSide = rhs.GetSide();
      return *this;
}
```

Нужно выполнить проверку на равенство объектов `rhs` и `this`, в противном случае обращение к оператору `a = a` приведет к аварийному отказу вашей программы.

10. **Жучки:** что неправильно в следующем примере использования оператора суммирования?

```
VeryShort  VeryShort::operator+ (const VeryShort& rhs)
{
   itsVal += rhs.GetItsVal();
   return *this;
}
```

Этот оператор `operator+` изменяет значение в одном из операндов, а не создает с помощью суммирования новый объект `VeryShort`. Правильно написать следующее:

```
VeryShort VeryShort::operator+ (const VeryShort& rhs)
{
    return VeryShort(itsVal + rhs.GetItsVal());
}
```

День 11

Контрольные вопросы

1. Что такое v-таблица?

   V-таблица, *или таблица виртуальных функций, является обычным средством* управления виртуальными функциями в C++, используемым компилятором. Эта таблица хранит список адресов всех виртуальных функций и обеспечивает вызов правильной функции в зависимости от указываемого типа объекта во время выполнения программы.

2. Что представляет собой виртуальный деструктор?

   Деструктор любого класса, который может быть объявлен виртуальным. Во время выполнения программы при применении `delete` к указателю на определенный тип объекта будет вызван деструктор соответствующего типа.

3. Можно ли объявить виртуальный конструктор?

   Виртуальных конструкторов не существует.

4. Как создать виртуальный конструктор-копировщик?

Путем создания в классе виртуального метода, который вызывает конструктор-копировщик.

5. Как вызвать функцию базового класса из объекта производного класса, если в производном классе эта функция была замещена?

```
Base::FunctionName();
```

6. Как вызвать функцию базового класса из объекта производного класса, если в производном классе эта функция не была замещена?

```
FunctionName();
```

7. Если в базовом классе функция объявлена как виртуальная, а в производном классе виртуальность функции указана не была, сохранится ли функция как виртуальная в следующем произведенном классе?

   Да, виртуальность наследуется и не может быть отменена.

8. С какой целью используется ключевое слово `protected`?

   Защищенные члены (которые объявлены с использованием ключевого слова `protected`) доступны для функций-членов производных объектов.

Упражнения

1. Объявите виртуальную функцию, которая принимает одно целочисленное значение и возвращает `void`.

```
virtual void SomeFunction(int);
```

2. Запишите объявление класса `Square` (квадрат), произведенного от класса `Rectangle` (прямоугольник), который, в свою очередь, произведен от класса `Shape` (форма).

```
class Square : public Rectangle
{};
```

3. Предположим, что в предыдущем примере объект класса `Shape` не использует параметры, объект класса `Rectangle` принимает два параметра (`length` и `width`), а объект класса `Square` — один параметр (`length`); запишите конструктор для класса `Square`.

```
Square::Square(int length):
    Rectangle(length, length){}
```

4. Запишите виртуальный конструктор-копировщик для класса `Square`, взятого из упражнения 3.

```
class Square
    {
        public:
        // ...
        virtual Square * clone() const { return new Square(*this); }
        // ...
    };
```

5. **Жучки:** что неправильно в следующем программном коде?

```
void SomeFunction (Shape);
Shape * pRect = new Rectangle;
SomeFunction(*pRect);
```

Возможно, здесь все правильно. Функция SomeFunction ожидает получения объекта класса Shape. Вы передали ей объект класса Rectangle, произведенный от класса Shape. До тех пор пока вам не нужны никакие составные части класса Rectangle, такой подход будет нормально работать. Если же вам понадобятся члены класса Rectangle, придется изменить объявление функции SomeFunction, чтобы она принимала указатель или ссылку на объект класса Rectangle.

6. **Жучки:** что неправильно в следующем программном коде?

```
class Shape()
{
public:
     Shape();
     virtual ~Shape();
     virtual Shape(const Shape&);
} ;
```

Нельзя объявить виртуальным конструктор-копировщик.

День 12

Контрольные вопросы

1. Как обратиться к первому и последнему элементам массива SomeArray[25]?

   SomeArray[0], SomeArray[24]

2. Как объявить многомерный массив?

   Напишите набор индексов для каждого измерения. Например, SomeArray[2][3][2] — это трехмерный массив. Первое измерение содержит два элемента, второе — три, а третье — два.

3. Выполните инициализацию элементов многомерного массива, созданного при ответе на вопрос 2.

```
SomeArray[2][3][2] = { { {1,2},{3,4},{5,6} } , { {7,8},{9,10},{11,12} } };
```

4. Сколько элементов содержит массив SomeArray[10][5][20]?

   10×5×20=1 000

5. Каково максимальное число элементов, которые можно добавить в связанный список?

   Не существует никакого фиксированного максимума. Это зависит от объема доступной памяти.

6. Можно ли в связанном списке использовать индексы?

   Индексы для обозначения элементов связанного списка можно использовать только при написании собственного класса, который будет содержать связанный список и перегруженный оператор индексирования.

7. Каким является последний символ в строке “Сергей — хороший парень”?
8. Концевой нулевой символ.

Упражнения

1. Объявите двухмерный массив, который представляет поле для игры в крестики и нолики.

```
int GameBoard[3][3];
```

2. Запишите программный код, инициализирующий значением 0 все элементы созданного перед этим массива.

```
int GameBoard[3][3] = { {0,0,0},{0,0,0},{0,0,0} }
```

3. Объявите класс узла Node, поддерживающего целые числа.

```
class Node
{
public:
    Node ();
    Node (int);
    ~Node();
    void SetNext(Node * node) { itsNext = node; }
    Node * GetNext() const { return itsNext; }
    int GetVal() const { return itsVal; }
    void Insert(Node *);
    void Display();
private:
    int itsVal;
    Node * itsNext;
};
```

4. **Жучки:** что неправильно в следующей программе?

```
unsigned short SomeArray[5][4];
for (int i = 0; i<4; i++)
     for (int j = 0; j<5; j++)
          SomeArray[i][j] = i+j;
```

Массив SomeArray предназначен для хранения 5×4 элементов, но код инициализирует матрицу 4×5 элементов.

5. **Жучки:** что неправильно в следующей программе?

```
unsigned short SomeArray[5][4];
for (int i = 0; i<=5; i++)
     for (int j = 0; j<=4; j++)
          SomeArray[i][j] = 0;
```

Вероятно, программист хотел написать `i < 5`, но написал вместо этого `i <= 5`. Программа будет работать, когда `i == 5` и `j == 4`, но в массиве SomeArray нет такого элемента, как SomeArray[5][4].

День 13

Контрольные вопросы

1. Что такое приведение типа объекта вниз?

   Под приведением типа объекта вниз понимается такое объявление, когда указатель на базовый класс приводится во время выполнения программы к указателю на производный класс.

2. Что такое v-ptr?

   Указатель на виртуальную функцию v-ptr является элементом выполнения виртуальных функций. Каждый объект в классе, содержащем виртуальные функции, имеет указатель v-ptr, который ссылается на таблицу виртуальных функций для этого класса.

3. Предположим, для создания прямоугольника с закругленными углами используется класс RoundRect, произведенный от двух базовых классов — Rectangle и Circle, которые, в свою очередь, производятся от общего класса Shape. Как много объектов класса Shape создается при создании одного объекта класса RoundRect?

   Если никакой класс не наследует использование ключевого слова virtual, то создаются два объекта класса Shape: один для класса RoundRect и один для класса Circle. Если же ключевое слово virtual используется для обоих классов, то создается только один общий объект класса Shape.

4. Если классы Horse (Лошадь) и Bird (Птица) виртуально наследуются от класса Animal (Животное) как открытые, будут ли конструкторы этих классов инициализировать конструктор класса Animal? Если класс Pegasus (Пегас) наследуется сразу от двух классов, Horse и Bird, как в нем будет инициализироваться конструктор класса Animal?

   Оба класса Horse и Bird инициализируют в своих конструкторах базовый класс Animal. Класс Pegasus делает то же самое, но когда создается объект класса Pegasus, инициализации класса Animal в производных классах Horse и Bird игнорируются.

5. Объявите класс Vehicle (Машина) как абстрактный тип данных.

```
class Vehicle
{
    virtual void Move() = 0;
}
```

6. Если в программе объявлен класс ADT с тремя чистыми виртуальными функциями, сколько из них нужно заместить в производных классах, чтобы получить возможность создания объектов этих классов?

   Если нужно произвести еще один абстрактный класс, то можно заместить одну или две чистые виртуальные функции базового класса, либо не замещать их вообще. В случае наследования обычного неабстрактного класса необходимо заместить все три функции.

Упражнения

1. Опишите класс JetPlane (Реактивный самолет), наследуя его от двух базовых классов — Rocket (Ракета) и Airplane (Самолет).

```
class JetPlane : public Rocket, public Airplane
```

2. Произведите от класса JetPlane, объявленного в первом упражнении, новый класс 747.

```
class 747 : public JetPlane
```

3. Напишите программу, производящую классы Car (Легковой автомобиль) и Bus (Автобус) от класса Vehicle (Машина). Опишите класс Vehicle как абстрактный тип данных с двумя чистыми виртуальными функциями. Классы Car и Bus не должны быть абстрактными.

```
class Vehicle
{
    virtual void Move() = 0;
    virtual void Haul() = 0;
};

class Car : public Vehicle
{
    virtual void Move();
    virtual void Haul();
};

class Bus : public Vehicle
{
    virtual void Move();
    virtual void Haul();
};
```

4. Измените программу из предыдущего упражнения таким образом, чтобы класс Car тоже стал ADT, и произведите от него три новых класса: SportsCar (Спортивный автомобиль), Wagon (Фургон) и Coupe (Двухместный автомобиль-купе). В классе Car должна замещаться одна из виртуальных функций, объявленных в классе Vehicle, с вызовом функции базового класса.

```
class Vehicle
{
    virtual void Move() = 0;
    virtual void Haul() = 0;
};

class Car : public Vehicle
{
    virtual void Move();
};

class Bus : public Vehicle
{
    virtual void Move();
```

```
    virtual void Haul();
};

class SportsCar : public Car
{
    virtual void Haul();
};

class Coupe : public Car
{
    virtual void Haul();
};
```

День 14

Контрольные вопросы

1. Могут ли статические переменные-члены быть закрытыми?

   Да. Поскольку они являются переменными-членами, то доступ к ним может управляться подобно доступу к любым другим переменным-членам. Если статическая переменная-член объявлена как закрытая, то доступ к ней можно получить только с помощью открытого статического метода класса.

2. Объявите статическую переменную-член.

   ```
   static int itsStatic;
   ```

3. Объявите статическую функцию.

   ```
   static int SomeFunction();
   ```

4. Объявите указатель на функцию, принимающую параметр типа `int` и возвращающую значение типа `long`.

   ```
   long (* function)(int);
   ```

5. Измените указатель, созданный в задании 4, на указатель на функцию-член класса `Car`.

   ```
   long ( Car::*function)(int);
   ```

6. Объявите массив из десяти указателей, созданных в задании 5.

   ```
   long ( Car::*function)(int) theArray [10];
   ```

Упражнения

1. Напишите короткую программу, объявляющую класс с одной обычной переменной-членом и одной статической переменной-членом. Создайте конструктор, выполняющий инициализацию переменной-члена и приращение статической переменной-члена. Затем объявите деструктор, который уменьшает на единицу значение статической переменной.

```
1:    class myClass
2:    {
3:    public:
4:        myClass();
5:        ~myClass();
6:    private:
7:        int itsMember;
8:        static int itsStatic;
9:    };
10:
11:    myClass::myClass():
12:      itsMember(1)
13:    {
14:        itsStatic++;
15:    }
16:
17:    myClass::~myClass()
18:    {
19:        itsStatic--;
20:    }
21:
22:    int myClass::itsStatic = 0;
23:
24:    int main()
25:    {}
```

2. Используя программный блок из упражнения 1, напишите короткую выполняемую программу, которая создает три объекта, а затем выводит значения их переменных-членов и статической переменной-члена класса. Затем последовательно удаляйте объекты и выводите на экран значение статической переменной-члена.

```
1:    #include <iostream.h>
2:
3:    class myClass
4:    {
5:    public:
6:        myClass();
7:        ~myClass();
8:        void ShowMember()
9:        void ShowStatic();
10:    private:
11:        int itsMember;
12:        static int itsStatic;
13:    };
14:
15:    myClass::myClass():
16:      itsMember(1)
17:    {
18:        itsStatic++;
```

```
19:    }
20:
21:    myClass::~myClass()
22:    {
23:        itsStatic--;
24:        cout << "In destructor. ItsStatic: " << itsStatic << endl;
25:    }
26:
27:    void myClass::ShowMember()
28:    {
29:        cout << "itsMember: " << itsMember << endl;
30:    }
31:
32:    void myClass::ShowStatic()
33:    {
34:        cout << "itsStatic: " << itsStatic << endl;
35:    }
36:    int myClass::itsStatic = 0;
37:
38:    int main()
39:    {
40:        myClass obj1;
41:        obj1.ShowMember();
42:        obj1.ShowStatic();
43:
44:        myClass obj2;
45:        obj2.ShowMember();
46:        obj2.ShowStatic();
47:
48:        myClass obj3;
49:        obj3.ShowMember();
50:        obj3.ShowStatic();
51:      return 0;
52:    }
```

3. Измените программу из упражнения 2 таким образом, чтобы доступ к статической переменной-члену осуществлялся с помощью статической функции-члена. Сделайте статическую переменную-член закрытой.

```
1:    #include <iostream.h>
2:
3:    class myClass
4:    {
5:    public:
6:        myClass();
7:        ~myClass();
8:        void ShowMember();
9:        static int GetStatic();
```

```
10:  private:
11:      int itsMember;
12:      static int itsStatic;
13:  };
14:
15:  myClass::myClass():
16:    itsMember(1)
17:  {
18:      itsStatic++;
19:  }
20:
21:  myClass::~myClass()
22:  {
23:      itsStatic--;
24:  cout << "In destructor. ItsStatic: " << itsStatic << endl;
25:  }
26:
27:  void myClass::ShowMember()
28:  {
29:      cout << "itsMember: " << itsMember << endl;
30:  }
31:
32:  int myClass::itsStatic = 0;
33:
34:  void myClass::GetStatic()
35:  {
36:      return itsStatic;
37:  }
38:
39:  int main()
40:  {
41:      myClass obj1;
42:      obj1.ShowMember();
43:      cout << "Static: " << myClass::GetStatic() << endl;
44:
45:      myClass obj2;
46:      obj2.ShowMember();
47:      cout << "Static: " << myClass::GetStatic() << endl;
48:
49:      myClass obj3;
50:      obj3.ShowMember();
51:      cout << "Static: " << myClass::GetStatic() << endl;
52:  return 0;
53:  }
```

4. Создайте в программе из упражнения 3 указатель на функцию-член для доступа к значению нестатической переменной-члена и воспользуйтесь им для вывода этих значений на печать.

```
1:    #include <iostream.h>
2:
3:    class myClass
4:    {
5:    public:
6:        myClass();
7:        ~myClass();
8:        void ShowMember();
9:        static int GetStatic();
10:   private:
11:       int itsMember;
12:       static int itsStatic;
13:   }
14:
15:   myClass::myClass():
16:     itsMember(1)
17:   {
18:       itsStatic++;
19:   }
20:
21:   myClass::~myClass()
22:   {
23:       itsStatic--;
24:       cout << "In destructor. ItsStatic: " << itsStatic << endl;
25:   }
26:
27:   void myClass::ShowMember()
28:   {
29:       cout << "itsMember: " << itsMember << endl;
30:   }
31:
32:   int myClass::itsStatic = 0;
33:
34:   int myClass::GetStatic()
35:   {
36:       return itsStatic;
37:   }
38:
39:   int main()
40:   {
41:       void (myClass::*PMF) ();
42:
43:       PMF=myClass::ShowMember;
44:
45:       myClass obj1;
46:       (obj1.*PMF)();
47:       cout << "Static: " << myClass::GetStatic() << endl;
```

```
48:
49:         myClass obj2;
50:         (obj2.*PMF)();
51:         cout << "Static; " << myClass::GetStatic() << endl;
52:
53:         myClass obj3;
54:         (obj3.*PMF)();
55:         cout << "Static: " << myClass::GetStatic() << endl;
56:     return 0;
57:     }
```

5. Добавьте две дополнительные переменные-члена к классу из предыдущих упражнений. Добавьте методы доступа, возвращающие значения всех этих переменных. Все функции-члены должны возвращать значения одинакового типа и иметь одинаковую сигнатуру. Для доступа к этим методам используйте указатель на функцию-член.

```
1:      #include <iostream.h>
2:
3:      class myClass
4:      {
5:      public:
6:          myClass();
7:          ~myClass();
8:          void ShowMember();
9:          void ShowSecond();
10:         void ShowThird();
11:         static int GetStatic();
12:     private:
13:         int itsMember;
14:         int itsSecond;
15:         int itsThird;
16:         static int itsStatic;
17:     }
18:
19:     myClass::myClass()
20:       itsMember(1),
21:       itsSecond(2),
22:       itsThird(3)
23:     {
24:         itsStatic++;
25:     }
26:
27:     myClass::~myClass()
28:     {
29:         itsStatic--;
30:         cout << "In destructor. ItsStatic: " << itsStatic << endl;
31:     }
32:
```

```
33:    void myClass::ShowMember()
34:    {
35:        cout << "itsMember: " << itsMember << endl;
36:    }
37:
38:    void myClass:: ShowSecond()
39:    {
40:        cout << "itsSecond: " << itsSecond << endl;
41:    }
42:
43:    void myClass::ShowThird()
44:    {
45:        cout << "itsThird: " << itsThird << endl;
46:    }
47:    int myClass::itsStatic = 0;
48:
49:    int myClass::GetStatic()
50:    {
51:        return itsStatic;
52:    }
53:
54:    int main()
55:    {
56:        void (myClass::*PMF) ();
57:
58:        myClass obj1;
59:        PMF=myClass::ShowMember;
60:        (obj1.*PMF)();
61:        PMF=myClass::ShowSecond;
62:        (obj1.*PMF)();
63:        PMF=myClass::ShowThird;
64:        (obj1.*PMF)();
65:        cout << "Static: " << myClass::GetStatic() << endl;
66:
67:        myClass obj2;
68:        PMF=myClass::ShowMember;
69:        (obj2.*PMF)();
70:        PMF=myClass::ShowSecond;
71:        (obj2.*PMF)();
72:        PMF=myClass::ShowThird;
73:        (obj2.*PMF)();
74:        cout << "Static: " << myClass::GetStatic() << endl;
75:
76:        myClass obj3;
77:        PMF=myClass::ShowMember;
78:        (obj3.*PMF)();
79:        PMF=myClass::ShowSecond;
```

```
80:         (obj3.*PMF)();
81:         PMF=myClass::ShowThird;
82:         (obj3.*PMF)();
83:         cout << "Static: " << myClass::GetStatic() << endl;
84:     return 0;
85:     }
```

День 15

Контрольные вопросы

1. Как объявить класс, являющийся частным проявлением другого класса?

   С помощью открытого наследования.

2. Как объявить класс, объекты которого должны использоваться в качестве переменных-членов другого класса?

   Необходимо использовать вложение классов.

3. В чем состоят различия между вложением и делегированием?

   Под вложением понимают использование объектов одного класса в качестве переменных-членов другого класса. Под делегированием — передачу одним классом другому классу выполнения некоторых специфических функций. В то же время делегирование часто реализуется за счет вложения классов.

4. В чем состоят различия между делегированием и выполнением класса в пределах другого класса?

   Под делегированием понимают передачу одним классом другому классу выполнения некоторых специфических функций. Под реализацией в пределах другого класса — наследование выполнения специфических функций от другого класса.

5. Что такое функция-друг?

   Это функция, объявленная с целью получения доступа к защищенным и закрытым членам другого класса.

6. Что такое класс-друг?

   Это класс, объявленный таким образом, чтобы все его функции-члены были дружественными по отношению к другому классу.

7. Если класс `Dog` является другом `Boy`, то можно ли сказать, что `Boy` — друг `Dog`?

   Нет, дружественность классов не взаимна.

8. Если класс `Dog` является другом `Boy`, а `Terrier` произведен от `Dog`, является ли `Terrier` другом `Boy`?

   Нет, дружественность классов не наследуется.

9. Если класс `Dog` является другом `Boy`, а `Boy` — другом `House`, можно ли считать `Dog` другом `House`?

   Нет, дружественность классов не ассоциативна.

10. Где необходимо размещать объявление функции-друга?

В любом месте внутри объявления класса. Не имеет никакого значения, в каком именно разделе будет помещено это объявление — в `public:`, `protected:` или `private:`.

Упражнения

1. Объявите класс `Animal` (Животное), который содержит переменную-член, являющуюся объектом класса `String`.

```
class Animal:
{
private:
    String itsName;
};
```

2. Опишите класс `BoundedArray`, являющийся массивом.

```
class boundedArray : public Array
{
//...
}
```

3. Опишите класс `Set`, выполняемый в пределах массива `BoundedArray`.

```
class Set : private Array
{
// ...
}
```

4. Измените листинг 15.1 таким образом, чтобы класс `String` включал перегруженный оператор вывода (>>).

```
1:     #include <iostream.h>
2:     #include <string.h>
3:
4:     class String
5:     {
6:        public:
7:           // конструкторы
8:           String();
9:            String(const char *const);
10:           String(const String &);
11:          ~String();
12:
13:          // перегруженные операторы
14:          char & operator[](int offset);
15:          char operator[](int offset) const;
16:          String operator+(const String&);
17:          void operator+=(const String&);
18:          String & operator= (const String &);
19:          friend ostream& operator<<
```

```
20:           ( ostream& _theStream,String& theString);
21:         friend istream& operator>>
22:           ( istream& _theStream,String& theString);
23:         // Общие функции доступа
24:         int GetLen()const { return itsLen; }
25:         const char * GetString() const { return itsString; }
26:         // static int ConstructorCount;
27:
28:      private:
29:         String (int);       // закрытый конструктор
30:         char * itsString;
31:         unsigned short itsLen;
32:
33:   };
34:
35:   ostream& operator<<( ostream& theStream,String& theString)
36:   {
37:       theStream << theString.GetString();
38:       return theStream;
39:   }
40:
41:   istream& operator>>( istream& theStream,String& theString)
42:   {
43:       theStream >> theString.GetString();
44:       return theStream;
45:   }
46:
47:   int main()
48:   {
49:       String theString("Привет, мир.");
50:       cout << theString;
51:   return 0;
52:   }
```

5. Жучки: что неправильно в этой программе?

```
1:    #include <iostream.h>
2:
3:    class Animal;
4:
5:    void setValue(Animal& , int);
6:
7:
8:    class Animal
9:    {
10:   public:
11:      int GetWeight()const {  return itsWeight; }
12:      int GetAge() const {  return itsAge; }
```

```
13:    private:
14:       int itsWeight;
15:       int itsAge;
16:    } ;
17:
18:    void setValue(Animal& theAnimal, int theWeight)
19:    {
20:       friend class Animal;
21:       theAnimal.itsWeight = theWeight;
22:    }
23:
24:    int main()
25:    {
26:       Animal peppy;
27:       setValue(peppy,5);
28:    return 0;
29:    }
```

Нельзя помещать объявление `friend` в функцию. Нужно объявить функцию другом в объявлении класса.

6. *Исправьте листинг, приведенный в упражнении 5, и откомпилируйте его.*

```
1:     #include <iostream.h>
2:
3:     class Animal;
4:
5:     void setValue(Animal& , int);
6:
7:
8:     class Animal
9:     {
10:    public:
11:        friend void setValue(Animal&, int);
12:        int GetWeight()const { return itsWeight; }
13:        int GetAge() const { return itsAge; }
14:    private:
15:        int itsWeight;
16:        int itsAge;
17:    };
18:
19:    void setValue(Animal& theAnimal, int theWeight)
20:    {
21:        theAnimal.itsWeight = theWeight;
22:    }
23:
24:    int main()
25:    {
26:        Animal peppy;
```

```
27:         setValue(peppy,5);
28:     return 0;
29:     }
```

7. Жучки: что неправильно в этой программе?

```
1:      #include <iostream.h>
2:
3:      class Animal;
4:
5:      void setValue(Animal& , int);
6:      void setValue(Animal& ,int,int);
7:
8:      class Animal
9:      {
10:     friend void setValue(Animal& ,int);
11:     private:
12:        int itsWeight;
13:        int itsAge;
14:     } ;
15:
16:     void setValue(Animal& theAnimal, int theWeight)
17:     {
18:         theAnimal.itsWeight = theWeight;
19:     }
20:
21:
22:     void setValue(Animal& theAnimal, int theWeight, int theAge)
23:     {
24:        theAnimal.itsWeight = theWeight;
25:        theAnimal.itsAge = theAge;
26:     }
27:
28:     int main()
29:     {
30:        Animal peppy;
31:        setValue(peppy,5);
32:        setValue(peppy,7,9);
33:     return 0;
34:     }
```

Функция `setValue(Animal& ,int)` была объявлена дружественной, но перегруженная функция `setValue(Animal& ,int,int)` не была объявлена дружественной.

8. Исправьте листинг, приведенный в упражнении 7, и откомпилируйте его.

```
1:      #include <iostream.h>
2:
3:      class Animal;
4:
```

```
5:    void setValue(Animal& , int);
6:    void setValue(Animal& ,int,int);
7:
8:    class Animal
9:    {
10:   friend void setValue(Animal& ,int);
11:   friend void setValue(Animal& ,int,int); // изменение!
12:   private:
13:      int itsWeight;
14:      int itsAge;
15:   } ;
16:
17:   void setValue(Animal& theAnimal, int theWeight)
18:   {
19:       theAnimal.itsWeight = theWeight;
20:   }
21:
22:
23:   void setValue(Animal& theAnimal, int theWeight, int theAge)
24:   {
25:      theAnimal.itsWeight = theWeight;
26:      theAnimal.itsAge = theAge;
27:   }
28:
29:   int main()
30:   {
31:      Animal peppy;
32:      setValue(peppy,5);
33:      setValue(peppy,7,9);
34:   return 0;
35:   }
```

День 16

Контрольные вопросы

1. Что такое оператор ввода и как он работает?

   Оператор ввода (>>) является членом объекта `istream` и используется для записи данных в переменные программы.

2. Что такое оператор вывода и как он работает?

   Оператор вывода (<<)является членом объекта `ostream` и используется для записи данных в устройство вывода.

3. Перечислите три варианта перегруженной функции `cin.get()` и укажите основные их отличия.

Первый вариант функции-члена get() используется без параметров. Она возвращает значение считанного символа. При достижении конца файла она возвратит EOF (*end of file*, т.е. конец файла).

Второй вариант функции-члена cin.get() принимает в качестве параметра ссылку на символьную переменную. Этой переменной присваивается следующий символ в потоке ввода. Возвращаемым значением этой функции является объект iostream.

В третьей, последней версии в функции get() устанавливаются массив символов, количество считываемых символов и символ разделения (которым по умолчанию является разрыв строки). Эта версия функции get() возвращает символы в массив либо до тех пор, пока не будет введено максимально возможное количество символов, либо до первого символа разрыва строки. Если функция get() встречает символ разрыва строки, ввод прерывается, а символ разрыва строки остается в буфере ввода.

4. Чем cin.read() отличается от cin.getline()?

Функция cin.read() используется для чтения структур двоичных данных.

Функция cin.getline() предназначена для чтения из буфера istream.

5. Какая ширина устанавливается по умолчанию для вывода длинных целых чисел с помощью оператора вывода?

Автоматически устанавливается ширина, достаточная для отображения всего числа.

6. Какое значение возвращает оператор вывода?

Ссылку на объект istream.

7. Какой параметр принимается конструктором объекта ofstream?

Имя открываемого файла.

8. Что устанавливает аргумент ios::ate?

Аргумент ios::ate помещает точку ввода в конец файла, но вы можете записывать данные в любом месте файла.

Упражнения

1. Напишите программу, использующую четыре стандартных объекта класса iostream — cin, cout, cerr и clog.

```
1:    #include <iostream.h>
2:    int main()
3:    {
4:        int x;
5:        cout << "Enter a number: ";
6:        cin >> x;
7:        cout << "You entered: " << x << endl;
8:        cerr << "Uh oh, this to cerr!" << endl;
9:        clog << "Ouh oh, this to clog!" << endl;
10:    return 0;
11:    }
```

2. Напишите программу, предлагающую пользователю ввести свое полное имя с последующим выводом этого имени на экран.

```
1:    #include <iostream.h>
2:    int main()
```

```
3:    {
4:        char name[80];
5:        cout << "Enter your full name: ";
6:        cin.getline(name,80);
7:        cout << "\nYou entered: " << name << endl;
8:    return 0;
9:    }
```

3. Перепишите листинг 16.9, отказавшись от использования методов `putback()` и `ignore()`.

```
1:    // Листинг 16.9. Измененный
2:    #include <iostream.h>
3:
4:    int main()
5:    {
6:       char ch;
7:       cout << "enter a phrase: ";
8:       while ( cin.get(ch) );
9:       {
10:           switch (ch)
11:           {
12:             case '!':
13:                 cout << '$';
14:                 break;
15:             case '#':
16:                 break;
17:             default:
18:                 cout << ch;
19:                 break;
20:           }
21:       }
22:   return 0;
23:   }
```

4. Напишите программу, считывающую имя файла в качестве аргумента командной строки и открывающую файл для чтения. Разработайте алгоритм анализа всех символов, хранящихся в файле, и выведите на экран только текстовые символы и знаки препинания (пропускайте все непечатаемые символы). Закройте файл перед завершением работы программы.

```
1:    #include <fstream.h>
2:    enum BOOL { FALSE, TRUE };
3:
4:    int main(int argc, char**argv)   // возвращает 1 в случае ошибки
5:    {
6:
7:        if (argc != 2)
8:        {
```

```
9:              cout << "Usage: argv[0] <infile>\n";
10:             return(1);
11:         }
12:
13:     // открываем поток ввода
14:         ifstream fin (argv[1],ios::binary);
15:         if (!fin)
16:         {
17:             cout << "Unable to open " << argv[1] << " for reading.\n";
18:             return(1);
19:         }
20:
21:         char ch;
22:         while ( fin.get(ch))
23:             if ((ch > 32 && ch < 127) || ch == '\n' || ch  = '\t')
24:                 cout << ch;
25:         fin.close();
26:     }
```

5. Напишите программу, которая выведет заданные аргументы командной строки в обратном порядке, отбросив имя программы.

```
1:      #include <fstream.h>
2:
3:      int main(int argc, char**argv)   // возвращает 1 в случае ошибки
4:      {
5:          for (int ctr = argc-1; ctr ; ctr--)
6:            cout << argv[ctr] << " ";
7:      return 0;
8:      }
```

День 17

Контрольные вопросы

1. Можно ли использовать идентификаторы, объявленные в пространстве имен, без применения ключевого слова `using`?

   Да, имена, определенные в пространстве имен, можно свободно использовать в программе, если указывать перед ними идентификатор пространства имен.

2. Назовите основные отличия между именованными и неименованными пространствами имен.

   Неименованные пространства имен компилятор рассматривает так, как если бы к ним по умолчанию был применен оператор `using`. Следовательно, имена в них можно использовать без идентификатора пространства имен. Чтобы сделать доступными имена обычных пространств имен, необходимо либо обращаться к ним с

помощью идентификатора пространства имен, либо использовать оператор using или ключевое слово using в объявлении пространства имен.

Имена, определенные в обычном пространстве имен, можно использовать вне модуля трансляции, в котором объявлено данное пространство имен. Имена, определенные в неименованном пространстве имен, можно использовать только внутри того модуля трансляции, в котором объявлено данное пространство имен.

3. Что такое стандартное пространство имен std?

   Данное пространство определено в стандартной библиотеке C++ (C++ Standard Library) и содержит объявления всех классов и функций стандартной библиотеки.

Упражнения

1. **Жучки:** найдите ошибку в следующем коде:

```
#include <iostream>

int main()
{
    cout << "Hello world!" << end;
    return 0;
}
```

   Стандартный файл заголовка C++ iostream объявляет объекты cout и endl в пространстве имен std. Их нельзя использовать вне стандартного пространства имен std без соответствующего идентификатора.

2. Перечислите три способа устранения ошибки, найденной в коде упражнения 1.

   - `using namespace std;`
   - `using std::cout;`
     `using std::endl;`
   - `std::cout << "Hello world!" << std::endl;`

День 18

Контрольные вопросы

1. Какая разница между объектно-ориентированным и процедурным программированием?

   Процедурное программирование опирается на функции, отделенные от обрабатываемых ими данных. Объектно-ориентированное программирование объединяет данные и функции в таком понятии, как объект, и фокусирует внимание на взаимодействии между объектами.

2. Каковы этапы объектно-ориентированного анализа и проектирования?

   а) разработка концепции;

   б) анализ;

в) проектирование;

г) реализация;

д) тестирование;

е) возвращение.

3. Как связанны диаграммы последовательности и сотрудничества?

Это два вида диаграмм взаимодействий классов. Диаграмма последовательности определяет последовательность событий за некоторое время, а диаграмма сотрудничества — принципы взаимодействия классов. Диаграмму сотрудничества можно создать прямо из диаграммы последовательности с помощью такого средства, как Rational Rose.

Упражнения

1. Предположим, что есть две пересекающиеся улицы с двусторонним движением, светофорами и пешеходными переходами. Нужно создать виртуальную модель, чтобы определить, позволит ли изменение частоты подачи сигнала светофора сделать дорожное движение более равномерным.

Какие объекты и какие классы потребуются для имитации этой ситуации?

Автомобили, мотоциклы, грузовики, велосипеды, пешеходы и спецмашины — все *используют этот перекресток. Кроме того, существует еще светофор, регулирующий* движение по перекрестку.

Нужно ли включать в модель покрытие дороги? Безусловно, качество дороги может оказывать существенное влияние на движение транспорта, однако для упрощения начального варианта модели лучше пока исключить из рассмотрения этот фактор.

Первым объектом, вероятно, будет сам перекресток. Возможно, объект перекрестка будет управлять списками автомобилей, ожидающих зеленого сигнала светофора в каждом направлении, а также списками пешеходов, ожидающих возможности пройти по переходу. Для этого объекта потребуются методы, позволяющие выбирать, какие автомобили и пешеходы пересекут этот перекресток и в каком количестве.

Поскольку рассматривается только один перекресток, нужно позаботиться о том, чтобы в программе допускалось создание только одного экземпляра этого объекта (подсказка: вспомните о статических методах и защищенном доступе к членам).

Как пешеходы, так и автомобили являются клиентами перекрестка. Они обладают некоторыми общими характеристиками. Например, могут появляться в любое время, могут отсутствовать вообще и могут ожидать сигнала светофора (хотя и на различных линиях). Эта "общность" является предпосылкой того, что стоит рассмотреть общий базовый класс для пешеходов и автомобилей.

Следовательно, в модель перекрестка необходимо включить такие классы:

```
class Entity;  // клиент перекрестка


// базовый класс для всех автомобилей, грузовиков, велосипедов и спецмашин
class Vehicle : Entity ...;
```

```
// базовый класс для пешеходов
class Pedestrian : Entity ...;


class Car : public Vehicle...;
class Truck : public Vehicle...;
class Motorcycle : public Vehicle...;
class Bicycle : public Vehicle...;
class Emergency_Vehicle : public Vehicle...;


// класс списка автомобилей и людей, ожидающих движения
class Intersection;
```

2. Усложним ситуацию из упражнения 1. Предположим, что есть три вида водителей: таксисты, переезжающие переход на красный свет; иногородние, которые едут медленно и осторожно; и частники, которые ведут машины по-разному, в зависимости от представлений о своей "крутизне".

   Также есть два вида пешеходов: местные, которые переходят улицу, где им заблагорассудится, и туристы, которые переходят улицу только на зеленый свет.

   А кроме того, есть еще велосипедисты, которые ведут себя то как пешеходы, то как водители.

   Как эти соображения изменят модель?

   Вероятно, целесообразно начать с создания производных объектов, которые моделируют разновидности рассмотренных выше объектов:

```
class Local_Car : public Car...;
        class Tourist_Car : public Car...;
        class Taxi : public Car...;
class Local_Pedestrian : public Pedestrian...;
        class Tourist_Pedestrian : public Pedestrian...;
class Local_Bicycle : public Bicycle...;
```

   Используя виртуальные методы, для объектов разных классов можно модифицировать общее поведение в соответствии с особенностями этих объектов. Например, местный водитель может реагировать на красный сигнал светофора не так, как турист, но в остальном наследовать общее поведение своего класса.

3. Вам заказали программу планирования времени конференций и встреч, а также бронирования мест в гостинице для визитеров компании и для участников конференций. Определите главные подсистемы.

   Для этого проекта нужно написать две отдельные программы: программу-клиент, которую будут запускать пользователи, и программу-сервер, которая будет работать на отдельном компьютере. Кроме того, компьютер клиента должен иметь административный компонент, позволяющий системному администратору добавлять новых людей и новые помещения.

Если вы решили реализовать этот проект в виде модели типа клиент/сервер, то программа-клиент должна принимать данные, вводимые пользователем, и генерировать запрос к программе-серверу. Сервер должен обслужить запрос и отправить результаты назад клиенту. С помощью этой модели многие участники конференции смогут одновременно планировать свои встречи.

На стороне клиента (помимо административного модуля) существует две основные подсистемы: интерфейс пользователя и система связей. На стороне сервера — три основные подсистемы: связей, планирования и почтового интерфейса, который объявляет пользователю об изменениях в расписании.

4. Спроектируйте интерфейсы к классам той части программы, обсуждаемой в упражнении 3, которая относится к резервированию гостиничных номеров.

Для организации конференции или деловой встречи необходимо зарезервировать помещение на определенное время. Этим занимается организационный комитет, который должен знать конкретное время проведения встречи и список участников.

В качестве объектов, вероятно, следует определить всех возможных пользователей системы, а также все имеющиеся залы заседаний и номера гостиниц. Не забудьте также включить для календаря и класс `Meeting`, который инкапсулирует все, что известно о конкретной встрече или конференции.

Приведем прототипы перечисленных выше классов.

```
class Calendar_Class;           // ссылка на класс
class Meeting;                 // ссылка на класс
class Configuration
{
public:
    Configuration();
    ~Configuration();
    Meeting Schedule( ListOfPerson&, Delta Time duration );
    Meeting Schedule( ListOfPerson&, Delta Time duration, Time );
    Meeting Schedule( ListOfPerson&, Delta Time duration, Room );
    ListOfPerson&     People();      // открытые методы доступа
    ListOfRoom&     Rooms();      // открытые методы доступа

protected:
    ListOfRoom     rooms;
    ListOfPerson     people;
};
typedef long      Room_ID;

class Room
{
public:
    Room( String name, Room_ID id, int capacity, String directions = "", String
description = "" );
    ~Room();
    Calendar_Class Calendar();
```

```
protected:
    Calendar_Class     calendar;
    int          capacity;
    Room_ID     id;
    String          name;
    String          directions;     // где этот гостиничный номер?
    String          description;
};
typedef long Person_ID;

class Person
{
public:
    Person ( String name, Person_ID id );
    ~Person();
    Calendar_Class Calendar();       // место доступа для добавления встреч

protected:
    Calendar_Class     calendar;
    Person_ID     id;
    String          name;
};

class Calendar_Class
{
public:
    Calendar_Class();
    ~Calendar_Class();

    void Add( const Meeting& );      // добавляем встречу в календарь
    void Delete( const Meeting& );
    Meeting* Lookup( Time );     // проверяем, не назначена ли уже встреча на это чис-
ло
    Block( Time, Duration, String reason = "" );
// резервируем время...

protected:
    OrderedListOfMeeting meetings;
};
class Meeting
{
public:
    Meeting( ListOfPerson&, Room room, Time when, Duration duration, String purpose =
"" );
    ~Meeting();
protected:
    ListOfPerson     people;
```

```
    Room          room;
    Time          when;
    Duration      duration;
    String          purpose;
};
```

День 19

Контрольные вопросы

1. Какова разница между шаблоном и макросом?

   Шаблоны являются средствами программирования языка C++, поддерживающими контроль за соответствием типов данных. Макросы выполняются препроцессором и не обеспечивают безопасности работы с типами.

2. В чем состоит отличие параметра шаблона от параметра функции?

   Параметр шаблона используется для создания экземпляра шаблона для каждого типа. Если создать шесть экземпляров шаблонов, то будут созданы шесть различных классов или функций. Параметры функций определяют, какие данные передаются в функцию при ее вызове, но не могут использоваться для создания разных экземпляров одной функции.

3. Чем отличается обычный дружественный шаблонный класс от дружественного шаблонного класса, специализированного по типу?

   Обычный дружественный шаблонный класс создает одну функцию для всех экземпляров параметризованного класса, а специализированный по типу дружественный шаблонный класс создает специализированные по типу экземпляры функции для каждого экземпляра параметризованного класса.

4. Можно ли обеспечить особое выполнение для определенного экземпляра шаблона?

   Да. Создайте для конкретного экземпляра шаблона функцию, специализированную по типу. Чтобы изменить выполнение, например, для массивов целых чисел, помимо функции `Array<t>::SomeFunction()`, создайте также функцию `Array<int>::SomeFunction()`.

5. Сколько создается статических переменных-членов, если поместить один статический член в определение класса шаблона?

   По одной для каждого экземпляра класса.

6. Что представляют собой итераторы?

   Это обобщенные указатели. Итератор можно инкрементировать, чтобы он указывал на следующий узел в последовательности. К нему также можно применить операцию разыменования, чтобы возвратить узел, на который он указывает.

7. Что такое объект функции?

   Это экземпляр класса, в котором определен перегруженный оператор вызова функции `()`. Объект функции можно также использовать как обычную функцию.

Упражнения

1. Создайте шаблон на основе данного класса `List`:

```
class List
{
private:

public:
     List():head(0),tail(0),theCount(0) { }
     virtual ~List();
     void insert( int value );
     void append( int value );
     int is_present( int value ) const;
     int is_empty() const {  return head == 0; }
     int count() const {  return theCount; }
private:
     class ListCell
     {
     public:
          ListCell(int value, ListCell *cell =
):val(value),next(cell){ }
          int val;
          ListCell *next;
     } ;
     ListCell *head;
     ListCell *tail;
     int theCount;
} ;
```

Вот один из способов выполнения этого шаблона:

```
template <class Type>
class List
{

public:
    List():head(0),tail(0),theCount(0) { }
    virtual ~List();

    void insert( Type value );
    void append( Type value );
    int is_present( Type value ) const;
    int is_empty() const { return head == 0; }
    int count() const { return theCount; }

private:
    class ListCell
```

```
    {
    public:
        ListCell(Type value, ListCell *cell = 0):val(value),next(cell){}
        Type val;
        ListCell *next;
    };

    ListCell *head;
    ListCell *tail;
    int theCount;
};
```

2. Напишите выполнение обычной (не шаблонной) версии класса `List`.

```
void List::insert(int value)
{
    ListCell *pt = new ListCell( value, head );
    assert (pt != 0);

    // эта строка добавляется для обработки хвостового узла
    if ( head == 0 ) tail = pt;

    head = pt;
    theCount++;
}
void List::append( int value )
{
    ListCell *pt = new ListCell( value );
    if ( head == 0 )
        head = pt;
    else
        tail->next = pt;

    tail = pt;
    theCount++;
}

int List::is_present( int value ) const
{
    if ( head == 0 ) return 0;
    if ( head->val == value || tail->val == value )
        return 1;

    ListCell *pt = head->next;
    for (; pt != tail; pt = pt->next)
        if ( pt->val == value )
            return 1 ;

    return 0;
}
```

3. Напишите шаблонный вариант выполнения.

```
template <class Type>
List<Type>::~List()
{
    ListCell *pt = head;

    while ( pt )
    {
        ListCell *tmp = pt;
        pt = pt->next;
        delete tmp;
    }
    head = tail = 0;
}

template <class Type>
void List<Type>::insert(Type value)
{
    ListCell *pt = new ListCell( value, head );
    assert (pt != 0);

    // эта строка добавляется для обработки хвостового узла
    if ( head == 0 ) tail = pt;

    head = pt;
    theCount++;
}

template <class Type>
void List<Type>::append( Type value )
{
    ListCell *pt = new ListCell( value );
    if ( head == 0 )
        head = pt;
    else
        tail->next = pt;

    tail = pt;
    theCount++;
}

template <class Type>
int List<Type>::is_present( Type value ) const
{
    if ( head == 0 ) return 0;
    if ( head->val == value || tail->val == value )
        return 1;
```

```
    ListCell *pt = head->next;
    for (; pt != tail; pt = pt->next)
        if ( pt->val == value )
            return 1;

    return 0;
}
```

4. Объявите три списка объектов: типа `String`s, типа `Cat` и типа `int`.

```
List<String> string_list;
List<Cat> Cat_List;
List<int> int_List;
```

5. Жучки: что неправильно в приведенном ниже программном коде? (Предположите, что определяется шаблон класса `List`, а `Cat` — это класс, определенный выше в данной книге.)

```
List<Cat> Cat_List;
Cat Felix;
CatList.append( Felix );
cout << "Felix is " <<
    ( Cat_List.is_present( Felix ) ) ? "" : "not " << "present\ n";
```

ПОДСКАЗКА (поскольку задание не из самых легких): подумайте, чем тип `Cat` отличается от типа `int`.

В классе `Cat` не определен оператор `operator==`. Все операции, в которых сравниваются значения членов класса `List`, таких как `is_present`, будут вызывать ошибку компиляции. Для уменьшения вероятности возникновения таких ошибок перед объявлением шаблона поместите обширный комментарий, в котором должно быть указано, какие операторы следует определить в классе для успешного выполнения всех его методов.

6. Объявите дружественный оператор `operator==` для класса `List`.

```
friend int operator==( const Type& lhs, const Type& rhs );
```

7. Напишите выполнение дружественного оператора `operator==` для класса `List`.

```
template <class Type>
int List<Type>::operator==( const Type& lhs, const Type& rhs )
{
    // сначала сравниваем размеры списков
    if ( lhs.theCount != rhs.theCount )
        return 0;      // списки различны

    ListCell *lh = lhs.head;
    ListCell *rh = rhs.head;

    for(; lh != 0; lh = lh.next, rh = rh.next )
        if ( lh.value != rh.value )
```

```
            return 0;

    return 1;          // если они не различны, то совпадают
}
```

8. Грешит ли оператор `operator==` той же проблемой, которая существует в упражнении 5?

Да. Поскольку сравнение массива включает сравнение элементов, то для элементов также должен быть определен оператор `operator!=`.

9. Напишите выполнение функции шаблона, осуществляющей операцию обмена данными, в результате чего две переменные должны обменяться содержимым.

```
// шаблон swap:
// должен иметь оператор присваивания и конструктор-копировщик, определенные для
класса Type.
template <class Type>
void swap( Type& lhs, Type& rhs)
{
    Type temp( lhs );
    lhs = rhs;
    rhs = temp;
}
```

10. Напишите выполнение класса `SchoolClass`, показанного в листинге 19.8, как списка. Для добавления в список четырех студентов используйте функцию `push_back()`. Затем пройдитесь по полученному списку и увеличьте возраст каждого студента на один год.

```
#include <list>

template<class T, class A>
void ShowList(const list<T, A>& aList);     // отображаем свойства вектора

typedef list<Student> SchoolClass;
int main()
{
    Student Harry("Harry", 18);
    Student Sally("Sally", 15);
    Student Bill( "Bill", 17);
    Student Peter("Peter", 16);

    SchoolClass GrowingClass;
    GrowingClass.push_back(Harry);
    GrowingClass.push_back(Sally);
    GrowingClass.push_back(Bill);
    GrowingClass.push_back(Peter);
    ShowList(GrowingClass);

    cout << "Один год спустя:\n";
```

```
    for (SchoolClass::iterator i = GrowingClass.begin(); i != GrowingClass.end();
++i)
        i->SetAge(i->GetAge() + 1);
    ShowList(GrowingClass);

    return 0;
}
//
// Отображаем свойства списка
//
template<class T, class A>
void ShowList(const list<T, A>& aList)
{
    for (list<T, A>::const_iterator ci = aList.begin(); ci != aList.end(); ++ci)
        cout << *ci << "\n";

    cout << endl;
}
```

11. Измените код из упражнение 10 таким образом, чтобы для отображения данных о каждом студенте использовался объект функции.

```
#include <algorithm>

template<class T>
class Print
{
public:
void operator()(const T& t)
{
cout << t << "\n";
}
};

template<class T, class A>
void ShowList(const list<T, A>& aList)
{
    Print<Student>      PrintStudent;
    for_each(aList.begin(), aList.end(), PrintStudent);
    cout << endl;
}
```

День 20

Контрольные вопросы

1. Что такое исключение?

   Это объект, который создается в результате использования ключевого слова `throw`. Этот объект является признаком возникновения исключительной ситуации и передается в стек вызовов первого оператора `catch`, который выполняет обработку этого исключения.

2. Для чего нужен блок `try`?

   Блок `try` — это набор выражений программы, которые могут создавать исключительные ситуации.

3. Для чего используется оператор `catch`?

   Оператор `catch` содержит сигнатуру типа исключения, которое он способен обработать. Оператор `catch` располагается сразу за блоком `try` и выполняет роль приемника исключения, сгенерированного внутри блока `try`.

4. Какую информацию может содержать исключение?

   Исключение — это объект, способный содержать любую информацию, которую можно определить внутри класса, созданного пользователем.

5. Когда создается объект исключения?

   Объекты исключений создаются при вызове ключевого слова `throw`.

6. Следует ли передавать исключения как значения или как ссылки?

   Вообще исключения нужно передавать как ссылки. Если вы не собираетесь модифицировать содержимое объекта исключения, вам следует передать ссылку, определенную с помощью ключевого слова `const`.

7. Будет ли оператор `catch` перехватывать производные исключения, если он настроен на базовый класс исключения?

   Да, если исключение будет передано как ссылка.

8. Если используются два оператора `catch`, один из которых настроен на базовое сообщение, а второй — на производное, то в каком порядке их следует расположить?

   Операторы `catch` проверяются в порядке их расположения в исходном коде. Причем если первый оператор `catch` перехватит исключение, то другие операторы `catch` уже вызываться не будут. Поэтому операторы `catch` следует располагать в порядке от специфичных (производных) к общим (базовым).

9. Что означает оператор `catch(...)`?

   Оператор `catch(...)` будет перехватывать все исключения любого типа.

10. Что такое точка останова?

    Это позиция в коде, в которой отладчик остановит выполнение программы.

Упражнения

1. Запишите блок try и оператор catch для отслеживания и обработки простого исключения.

```
#include <iostream.h>
class OutOfMemory {};
int main()
{
    try
    {
        int *myInt = new int;
        if (myInt == 0)
            throw OutOfMemory();
    }
    catch (OutOfMemory)
    {
        cout << "Unable to allocate memory!\n";
    }
return 0;
}
```

2. Добавьте в исключение, полученное в упражнении 1, переменную-член и метод доступа и используйте их в блоке оператора catch.

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
class OutOfMemory;
{
public:
    OutOfMemory(char *);
    char* GetString() { return itsString; }
private:
    char* itsString;
};

OutOfMemory::OutOfMemory(char * theType)
{
    itsString = new char[80];
    char warning[] = "Out Of Memory! Can't allocate room for: ";
    strncpy(itsString,warning,60);
    strncat(itsString,theType,19);
}

int main()
{
    try
```

```
    {
        int *myInt = new int;
        if (myInt == 0)
            throw OutOfMemory("int");
    }
    catch (OutOfMemory& theException)
    {
        cout << theException.GetString();
    }
return 0;
}
```

3. Унаследуйте новое исключение от исключения, полученного в упражнении 2. Измените блок оператора `catch` таким образом, чтобы в нем происходила обработка как производного, так и базового исключений.

```
1:    #include <iostream.h>
2:
3:    // Абстрактный тип исключений
4:    class Exception
5:    {
6:    public:
7:        Exception(){}
8:        virtual ~Exception(){}
9:        virtual void PrintError() = 0;
10:   };
11:
12:   // Производный класс для обработки проблем памяти
13:   // Обратите внимание: в этом классе не производится выделение памяти
14:   class OutOfMemory : public Exception
15:   {
16:   public:
17:       OutOfMemory(){}
18:       ~OutOfMemory(){}
19:       virtual void PrintError();
20:   private:
21:   };
22:
23:   void OutOfMemory::PrintError()
24:   {
25:   cout << "Нет памяти !!\n";
26:   }
27:
28:   // Производный класс для обработки ввода неверных чисел
29:   class RangeError : public Exception
30:   {
31:   public:
32:       RangeError(unsigned long number){badNumber = number;}
```

```
33:         ~RangeError(){}
34:         virtual void PrintError();
35:         virtual unsigned long GetNumber() { return badNumber; }
36:         virtual void SetNumber( unsigned long number) {badNumber = number;}
37:     private:
38:         unsigned long badNumber;
39:     };
40:
41:     void RangeError::PrintError()
42:     {
43:         cout << "Number out of range. You used " << GetNumber() << "!!\n";
44:     }
45:
46:     void MyFunction();  // прототип функции
47:
48:     int main()
49:     {
50:         try
51:         {
52:             MyFunction();
53:         }
54:         // Чтобы использовать только один оператор catch,
55:         // примените для этого виртуальные функции
56:         catch (Exceptions theException)
57:         {
58:             theException.PrintError();
59:         }
60:     return 0;
61:     }
62:
63:     void MyFunction()
64:     {
65:     unsigned int *myInt = new unsigned int;
66:     long testNumber;
67:     if (myInt == 0)
68:         throw OutOfMemory();
69:     cout << "Enter an int: ";
70:     cin >> testNumber;
71:     // эту проверку лучше заменить серией
72:     // проверок, чтобы выявить неверные данные, введенные пользователем
73:     if (testNumber > 3768 || testNumber < 0)
74:         throw RangeError(testNumber);
75:
76:         *myInt = testNumber;
77:         cout << "Ok. myInt: " << *myInt;
78:         delete myInt;
79:     }
```

4. Измените код из упражнения 3, чтобы получить трехуровневый вызов функции.

```
1:    #include <iostream.h>
2:
3:    // Абстрактный тип исключений
4:    class Exception
5:    {
6:    public:;
7:        Exception(){}
8:        virtual ~Exception(){}
9:        virtual void PrintError() = 0;
10:   };
11:
12:   // Производный класс для обработки проблем памяти
13:   // Обратите внимание: в этом классе не производится выделение памяти!
14:   class OutOfMemory : public Exception
15:   {
16:   public:
17:       OutOfMemory(){}
18:       ~OutOfMemory(){}
19:       virtual void PrintError();
20:   private:
21:   };
22:
23:   void OutOfMemory::PrintError()
24:   {
25:       cout << "Нет памяти!!\n";
26:   }
27
28:   // Производный класс для обработки ввода неверных чисел
29:   class RangeError : public Exception
30:   {
31:   public:
32:       RangeError(unsigned long number){badNumber = number;}
33:       ~RangeError(){}
34:       virtual void PrintError();
35:       virtual unsigned long GetNumber() { return badNumber; }
36:       virtual void SetNumber(unsigned long number) {badNumber = number;}
37:   private:
38:       unsigned long badNumber;
39:   };
40:
41:   void RangeError::PrintError()
42:   {
43:       cout << " Number out of range. You used " << GetNumber() << "!!\n";
44:   }
45:
46:   // прототипы функций
```

```
47:    void MyFunction();
48:    unsigned int * FunctionTwo();
49:    void FunctionThree(unsigned int *);
50:
51:    int main()
52:    {
53:        try
54:        {
55:            MyFunction();
56:        }
57:        // Чтобы использовать только один оператор catch,
58:        // примените для этого виртуальные функции.
59:        catch (Exception& theException)
60:        {
61:            theException.PrintError();
62:        }
63:        return 0;
64:    }
65:
66:    unsigned int * FunctionTwo()
67:    {
68:        unsigned int *myInt = new unsigned int;
69:        if (myInt == 0)
70:            throw OutOfMemory();
71:        return myInt;
72:    }
73:
74:    void MyFunction()
75:    {
76:        unsigned int *myInt = FunctionTwo();
77:
78:        FunctionThree(myInt);
79:        cout << "Ok. myInt: " << *myInt;
80:        delete myInt;
81:    }
82:
83:    void FunctionThree(unsigned int *ptr)
84:    {
85:        long testNumber;
86:    cout << "Enter an int: ";
87:    cin >> testNumber;
88:    // эту проверку лучше заменить серией
89:    // проверок, чтобы выявить неверные данные, введенные пользователем
90:        if (testNumber > 3768 || testNumber < 0)
91:            throw RangeError(testNumber);
92:        *ptr = testNumber;
93:    }
```

5. **Жучки**: что неправильно в следующем коде?

```
#include "string.h"          // класс строк

class xOutOfMemory
{
public:
    xOutOfMemory( const String& where ) : location( where ){}
    ~xOutOfMemory(){}
    virtual String where(){ return location };
private:
    String location;
}

main()
{
    try {
        char *var = new char;
        if ( var == 0 )
            throw xOutOfMemory();
    }
    catch( xOutOfMemory& theException )
    {
        cout << "Out of memory at " << theException.location() << "\n";
    }
}
```

В процессе обработки ситуации нехватки памяти конструктором класса `xOutOfMemory` в области свободной памяти создается объект типа `string`. Это исключение может возникнуть только в том случае, когда программе не хватает памяти, поэтому попытка нового выделения памяти будет тем более неудачной.

Возможно, что попытка создать эту строку послужит причиной возникновения такого же исключения, что приведет к образованию бесконечного цикла, который будет выполняться до тех пор, пока компьютер не зависнет. Если эта строка все же нужна, можно выделить для нее память в статическом буфере до начала работы программы, а затем использовать ее по необходимости, т.е. при возникновении исключения.

День 21

Контрольные вопросы

1. Для чего нужны средства защиты от повторного включения?

   Эти средства используются для того, чтобы не допустить включение в программу одного и того же файла заголовка более одного раза.

2. Как указать компилятору, что необходимо напечатать содержимое промежуточного файла, полученного в результате работы препроцессора?

На разных компиляторах эта операция выполняется по-разному. Внимательно ознакомьтесь с документацией компилятора.

3. Какова разница между директивами `#define debug 0` и `#undef debug`?

   Директива `#define debug 0` определяет лексему `debug` и присваивает ей `0` (нуль). Поэтому везде, где встретится лексема `debug`, она будет заменена символом `0`. Директива `#undef debug` удаляет любое определение лексемы `debug`, в результате чего любой экземпляр лексемы `debug`, обнаруженный в файле, будет оставаться неизменным.

4. Что делает оператор дополнения до единицы?

   Инвертирует значение каждого бита переменной.

5. Чем отличается оператор побитового ИЛИ от оператора исключающего побитового ИЛИ?

   Оператор побитового ИЛИ возвращает значение `TRUE` (ИСТИНА), если установлен один из битов (или оба сразу). Оператор исключающего ИЛИ возвращает `TRUE` только в том случае, если данный бит установлен лишь в одном операнде, но не в обоих сразу.

6. Какова разница между операторами `&` и `&&`?

   Оператор `&` — это побитовое И, а `&&` — это логическое И.

7. Какова разница между операторами `|` и `||`?

   Оператор `|` — это побитовое ИЛИ, а `||` — это логическое ИЛИ.

Упражнения

1. Создайте защиту от повторного включения файла заголовка `STRING.H`.

```
#ifndef STRING_H
#define STRING_H
...
#endif
```

2. Напишите макрос `assert()`, который
   - будет печатать сообщение об ошибке, а также имя файла и номер строки, если уровень отладки равен `2`;
   - будет печатать сообщение (без имени файла и номера строки), если уровень отладки равен `1`;
   - не будет ничего делать, если уровень отладки равен `0`.

```
1:    #include <iostream.h>
2:
3:    #ifndef DEBUG
4:    #define ASSERT(x)
5:    #elif DEBUG == 1
6:    #define ASSERT(x) \
7:      if (! (x))\
8:      { \
9:          cout << "ERROR!! Assert " << #x << " failed\n"; \
10:               }
```

```
11:    #elif DEBUG == 2
12:    #define ASSERT(x) \
13:      if (! (x) ) \
14:      { \
15:           cout << " ERROR!! Assert " << #x << " failed\n"; \
16:        cout << " on line " << __LINE__ << "\n"; \
17:         cout << " in file " << __LINE__ << "\n"; \
18:      }
19:    #endif
```

3. Напишите макрос `DPrint`, который проверяет, определена ли лексема `DEBUG`, и, если да, выводит значение, передаваемое как параметр.

```
#ifndef DEBUG:
#define DPRINT(string)
#else
#define DPRINT(STRING) cout << #STRING ;
#endif
```

4. Напишите программу, которая складывает два числа без использования операции сложения (+). Подсказка: используйте побитовые операторы!

   Если рассмотреть сложение двух битов, то можно заметить, что ответ будет содержать два бита: бит результата и бит переноса. Таким образом, при сложении двух единиц в двоичной системе бит результата будет равен нулю, а бит переноса — единице. Если сложить два двоичных числа 101 и 001, получим следующие результаты:

```
101   // 5
001   // 1
110   // 6
```

   Следовательно, если сложить два соответствующих бита (каждый из них равен единице), то бит результата будет равен 0, а бит переноса — 1. Если же сложить два сброшенных бита, то и бит результата, и бит переноса будут равны 0. Если сложить два бита, один из которых установлен, а другой сброшен, бит результата будет равен 1, а бит переноса — 0. Перед вами таблица, которая обобщает эти правила сложения

Левый бит `lhs`	Правый бит `rhs`	Перенос	Результат
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

   Рассмотрим логику бита переноса. Если оба суммируемых бита (`lhs` и `rhs`) равны 0 или хотя бы один из них равен 0, бит переноса будет равен 0. И только если оба бита равны 1, бит переноса будет равен 1. Такая ситуация в точности совпадает с определением побитового оператора И (`&`).

   Если подобным образом рассмотреть логику бита результата, то окажется, что она совпадает с выполнением оператора побитового исключающего ИЛИ (`^`): если любой из суммируемых битов (но не оба сразу) равен 1, бит результата равен 1, в противном случае — 0.

Полученный бит переноса добавляется к следующему значимому биту. Это можно реализовать либо итеративным проходом через каждый бит, либо использованием рекурсии.

```
#include <iostream.h>

unsigned int add( unsigned int lhs, unsigned int rhs )
{
    unsigned int result, carry;
    while ( 1 )
    {
        result = lhs ^ rhs;
        carry = lhs & rhs;

        if ( carry == 0 )
            break;

        lhs = carry << 1;
        rhs = result;
    };
    return result;
}

int main()
{
    unsigned long a, b;
    for (;;)
    {
        cout << "Enter two numbers. (0 0 to stop): ";
        cin << a << b;
        if (!a && !b)
            break;
        cout << a << " + " << b << " = " << add(a,b) << endl;
    }
    return 0;
}
```

В качестве альтернативного варианта эту проблему можно решить с помощью рекурсии:

```
#include <iostream.h>

unsigned int add( unsigned int lhs, unsigned int rhs )
{
    unsigned int carry = lhs & rhs;
    unsigned int result = lhs ^ rhs;

    if ( carry )
        return add( result, carry << 1 );
```

```
    else
        return result;
}
int main()
{
    unsigned long a, b;
    for (;;)
    {
        cout << "Enter two numbers. (0 0 to stop): ";
        cin << a << b;
        if (!a && !b)
            break;
        cout << a << " + " << b << " = " << add(a,b) << endl;
    }
    return 0;
}
```

Предметный указатель

P

C

Т

У

Ф

Ц

Ш

Э