Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Тамбовский государственный технический университет»

Ю.Ю. Громов, О.Г. Иванова, М.П. Беляев, Ю.В. Минин

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

Рекомендовано федеральным государственным бюджетным образовательным учреждением высшего профессионального образования «Московский государственный технический университет имени Н.Э. Баумана» в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки 230400 «Информационные системы и технологии»

Тамбов
Издательство ФГБОУ ВПО «ТГТУ»
2013

УДК 004.05(075.8)
ББК $_{3}$973-018я73
Т384

Р е ц е н з е н т ы:

Кандидат технических наук, профессор ФГБОУ ВПО «ТГТУ»
Ю.Ф. Мартемьянов

Заслуженный деятель науки РФ,
доктор физико-математических наук, профессор ФГБОУ ВПО «ТГТУ»
В.Ф. Крапивин

Т384 Технология программирования : учебное пособие / Ю.Ю. Громов, О.Г. Иванова, М.П. Беляев, Ю.В. Минин. – Тамбов : Изд-во ФГБОУ ВПО «ТГТУ», 2013. – 172 с. – 100 экз.
ISBN 978-5-8265-1207-4

Содержит необходимые сведения о методах анализа, проектирования, реализации и тестирования программных систем и существующих подходах и технологиях программирования.

Предназначено для студентов вузов, обучающихся по направлению подготовки 230400 «Информационные системы и технологии».

УДК 004.05(075.8)
ББК $_{3}$973-018я73

ISBN 978-5-8265-1207-4

ВВЕДЕНИЕ

Учебное пособие разработано в соответствии с требованиями, предъявляемыми Федеральным государственным образовательным стандартом высшего профессионального образования (3-е поколение), и предназначено для студентов вузов, обучающихся по направлению 230400 «Информационные системы и технологии» и изучающих дисциплину «Технология программирования».

Создание программной системы – весьма трудоёмкая задача, особенно в наше время, когда объём кода программного обеспечения превышает сотни тысяч операторов. Будущий специалист в области разработки программного обеспечения должен иметь представление о методах анализа, проектирования, реализации и тестирования программных систем, а также ориентироваться в существующих подходах и технологиях.

Структурно пособие состоит из 4 глав и списка использованных источников. В первой главе описываются этапы развития технологии программирования и проблемы, возникающие при разработке сложных программных систем, рассматриваются подходы и этапы разработки программного обеспечения. Вторая глава посвящена технологическим приёмам структурного подхода к программированию и его основным концепциям: нисходящей разработке, структурному и модульному программированию, а также сквозному структурному контролю. Третья и четвёртая главы описывают процедурное и объектно-ориентированное программирование на языке высокого уровня C++.

1. ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ. ОСНОВНЫЕ ПОНЯТИЯ И ПОДХОДЫ

Программирование – сравнительно молодая и быстро развивающаяся отрасль науки и техники. Опыт ведения реальных разработок и совершенствования имеющихся программных и технических средств постоянно переосмысливается, в результате чего появляются новые методы, методологии и технологии, которые, в свою очередь, служат основой более современных средств разработки программного обеспечения. Исследовать процессы создания новых технологий и определять их основные тенденции целесообразно, сопоставляя эти технологии с уровнем развития программирования и особенностями имеющихся в распоряжении программистов программных и аппаратных средств.

1.1. ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ И ОСНОВНЫЕ ЭТАПЫ ЕЁ РАЗВИТИЯ

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения [1, 2]. Как любая другая технология, технология программирования представляет собой набор технологических инструкций, включающих:

– указание последовательности выполнения технологических операций;

– перечисление условий, при которых выполняется та или иная операция;

– описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т.п. (рис. 1.1).

Кроме набора операций и их последовательности, технология также определяет способ описания проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Различают технологии, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый метод, позволяющий решить конкретную задачу. В основе вторых обычно лежит базовый метод или подход, определяющий совокупность методов, используемых на разных этапах разработки, или методологию.

Рис. 1.1. Структура описания технологической операции

1.1.1. ПЕРВЫЙ ЭТАП – «СТИХИЙНОЕ» ПРОГРАММИРОВАНИЕ

Первый этап охватывает период от момента появления первых вычислительных машин до середины 60-х гг. XX в. В этот период практически отсутствовали сформулированные технологии, и программирование фактически было искусством. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных (рис. 1.2). Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

Появление ассемблеров позволило вместо двоичных или шестнадцатеричных кодов использовать символические имена данных и мнемоники кодов операций. В результате программы стали более «читаемыми».

Рис. 1.2. Структура первых программ

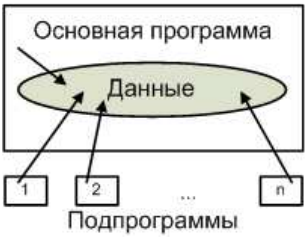


Рис. 1.3. Архитектура программы с глобальной областью данных

Создание языков программирования высокого уровня, таких как FORTRAN и ALGOL, существенно упростило программирование вычислений, снизив уровень детализации операций. Это, в свою очередь, позволило увеличить сложность программ.

Революционным было появление в языках средств, позволяющих оперировать подпрограммами. Идея написания подпрограмм появилась гораздо раньше, но отсутствие средств поддержки в первых языковых средствах существенно снижало эффективность их применения. Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы огромные библиотеки расчётных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рис. 1.3).

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок, было предложено в подпрограммах размещать локальные данные (рис. 1.4).

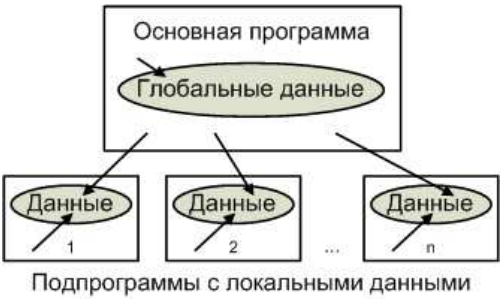


Рис. 1.4. Архитектура программы, использующей подпрограммы с локальными данными

Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне. Однако появление средств поддержки подпрограмм позволило осуществлять разработку программного обеспечения нескольким программистам параллельно.

В начале 60-х гг. XX в. разразился «кризис программирования». Он выражался в том, что фирмы, взявшиеся за разработку сложного программного обеспечения, такого как операционные системы, срывали все сроки завершения проектов. Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены.

Объективно все это было вызвано несовершенством технологии программирования. В отсутствии чётких моделей описания подпрограмм и методов их проектирования создание каждой подпрограммы превращалось в непростую задачу, интерфейсы подпрограмм получались сложными, и при сборке программного продукта выявлялось большое количество ошибок согласования. Исправление таких ошибок, как правило, требовало серьёзного изменения уже разработанных подпрограмм, что ещё более осложняло ситуацию, так как при этом в программу часто вносились новые ошибки, которые также необходимо было исправлять. Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван «структурным».

1.1.2. ВТОРОЙ ЭТАП – СТРУКТУРНЫЙ ПОДХОД К ПРОГРАММИРОВАНИЮ (60 – 70-е гг. XX в.)

Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приёмов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40 – 50 операторов) подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т.д.) данный способ получил название процедурной декомпозиции.

В отличие от используемого ранее процедурного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, ре-

комендовались формальные модели их описания, а также специальный метод проектирования алгоритмов – метод пошаговой детализации.

Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования. Как правило, они включали основные «структурные» операторы передачи управления, поддерживали вложение подпрограмм, локализацию и ограничение области «видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, C.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных. Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые модули (библиотеки подпрограмм), например, модуль графических ресурсов, модуль подпрограмм вывода на принтер (рис. 1.5).

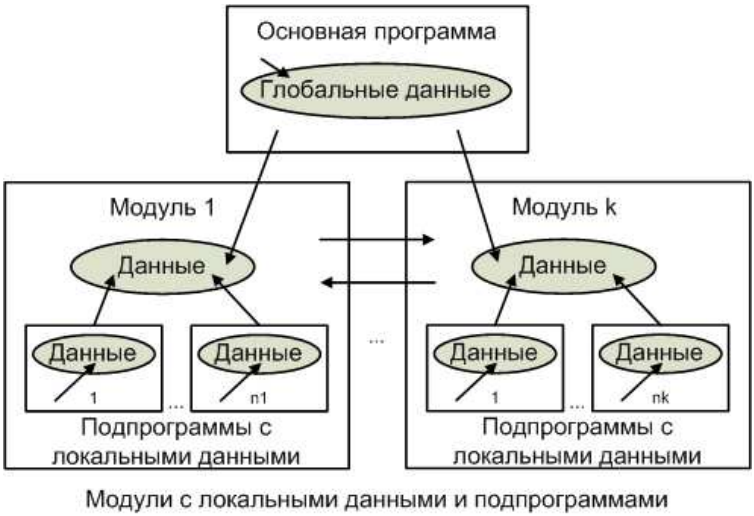


Рис. 1.5. Архитектура программы, состоящей из модулей

Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещён. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula.

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объёма было предложено использовать объектный подход.

1.1.3. ТРЕТИЙ ЭТАП – ОБЪЕКТНЫЙ ПОДХОД К ПРОГРАММИРОВАНИЮ (с середины 80-х до конца 90-х гг. XX в.)

Объектно-ориентированное программирование определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого типа (класса), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путём передачи сообщений (рис. 1.6).

Объектная структура программы впервые была использована в языке имитационного моделирования сложных систем Simula, появившемся ещё в 60-х гг. XX в. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования – языке Smalltalk (70-е гг. XX в.), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, C++, Modula, Java.

Основным достоинством объектно-ориентированного программирования по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, кото-

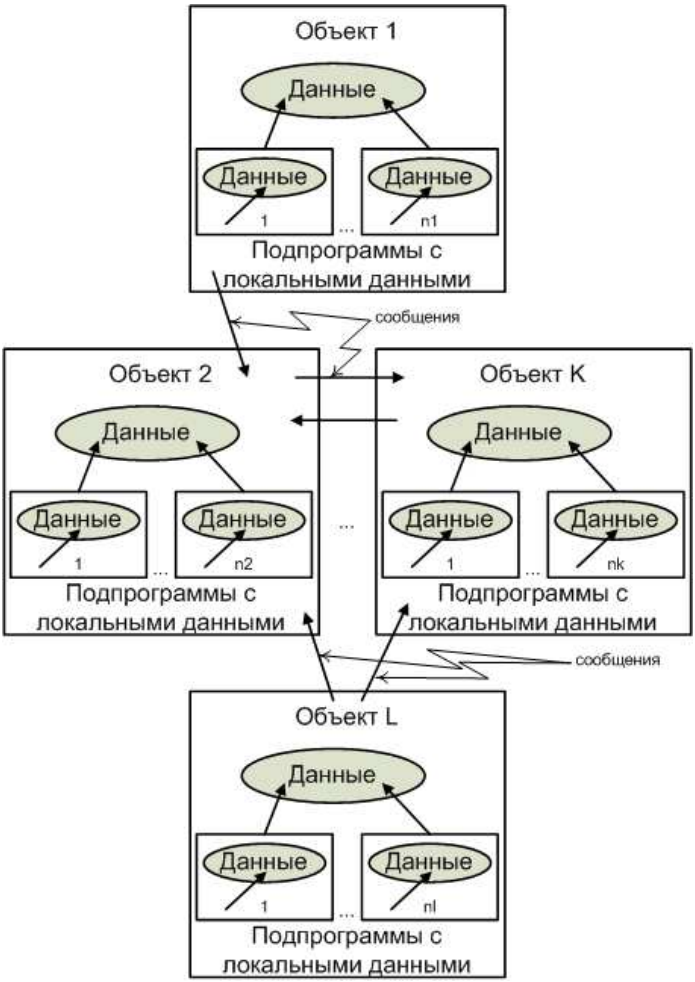


Рис. 1.6. Архитектура программ при объектно-ориентированном программировании

рая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) программы. Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполне-

ния. Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

1.1.4. ЧЕТВЁРТЫЙ ЭТАП – КОМПОНЕНТНЫЙ ПОДХОД И CASE-ТЕХНОЛОГИИ (с середины 90-х гг. XX в. до нашего времени)

Компонентный подход предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов, объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. Это позволяет программистам создавать продукты, хотя бы частично состоящие из повторно использованных частей, т.е. использовать технологию, хорошо зарекомендовавшую себя в области проектирования аппаратуры.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов), и технологии создания распределённых приложений CORBA (Common Object Request Broker Architecture – общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология COM фирмы Microsoft является развитием технологии OLE I (Object Linking and Embedding – связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология COM определяет общую парадигму взаимодействия программ любых типов: библиотек, приложений, операционной системы, т.е. позволяет одной части программного обеспечения использовать функции (службы), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рис. 1.7). Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM – распределённая COM).

На базе технологии COM и её распределённой версии DCOM были разработаны компонентные технологии, решающие различные задачи разработки программного обеспечения.

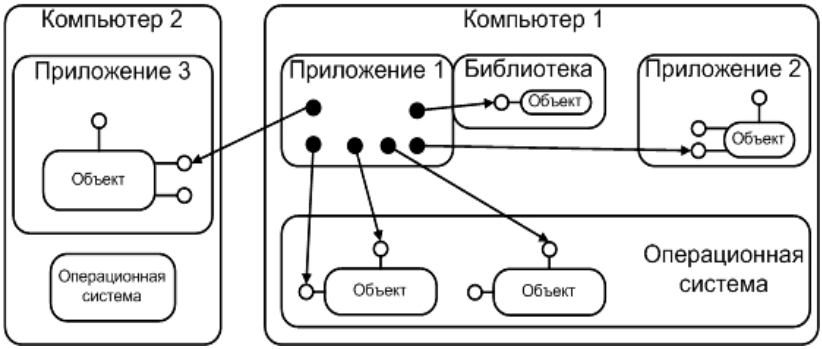


Рис. 1.7. Взаимодействие программных компонентов различных типов

OLE-automation или просто Automation (автоматизация) – технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений.

ActiveX – технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения как сосредоточенного на одном компьютере, так и распределённого в сети. Предполагает использование визуального программирования для создания компонентов – элементов управления ActiveX. Полученные таким образом элементы управления можно устанавливать на компьютер дистанционно с удалённого сервера, причём устанавливаемый код зависит от используемой операционной системы. Это позволяет применять элементы управления ActiveX в клиентских частях приложений Интернет.

Технология CORBA, разработанная группой компаний OMC (Object Management Group – группа внедрения объектной технологии программирования), реализует подход, аналогичный COM, на базе объектов и интерфейсов CORBA. Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ и потому эту технологию можно использовать для создания распределённого программного обеспечения в гетерогенной (разнородной) вычислительной среде. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью специального посредника, названного VisiBroker, и другого специализированного программного обеспечения.

Отличительной особенностью современного этапа развития технологии программирования, кроме изменения подхода, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые были названы

CASE-технологиями (Computer-Aided Software/System Engineering – разработка программного обеспечения/программных систем с использованием компьютерной поддержки). Без средств автоматизации разработка достаточно сложного программного обеспечения на настоящий момент становится трудно осуществимой. На сегодня существуют CASE-технологии, поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию.

Появление нового подхода не означает, что отныне всё программное обеспечение будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного программного обеспечения показывает, что он будет применяться достаточно широко.

1.2. ПРОБЛЕМЫ РАЗРАБОТКИ СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

Большинство современных программных систем объективно очень сложны. Эта сложность обуславливается многими причинами, главной из которых является логическая сложность решаемых ими задач.

Пока вычислительных установок было мало и их возможности были ограничены, ЭВМ применяли в очень узких областях науки и техники, причём, в первую очередь, там, где решаемые задачи были хорошо детерминированы и требовали значительных вычислений. В наше время, когда созданы мощные компьютерные сети, появилась возможность переложить на них решение сложных ресурсоёмких задач, о компьютеризации которых раньше никто и не думал. Сейчас в процесс компьютеризации вовлекаются совершенно новые предметные области, а для уже освоенных областей усложняются уже сложившиеся постановки задач.

Дополнительными факторами, увеличивающими сложность разработки программных систем, являются:

– сложность формального определения требований к программным системам;

– отсутствие удовлетворительных средств описания поведения дискретных систем с большим числом состояний при недетерминированной последовательности входных воздействий;

– коллективная разработка;

– необходимость увеличения степени повторяемости кодов.

Сложность определения требований к программным системам обусловливается двумя факторами. Во-первых, при определении требований необходимо учесть большое количество различных факторов. Во-вторых, разработчики программных систем не являются специали-

стами в автоматизируемых предметных областях, а специалисты в предметной области, как правило, не могут сформулировать проблему в нужном ракурсе.

Отсутствие удовлетворительных средств формального описания поведения дискретных систем. В процессе создания программных систем используют языки сравнительно низкого уровня. Это приводит к ранней детализации операций в процессе создания программного обеспечения и увеличивает объём описаний разрабатываемых продуктов, который, как правило, превышает сотни тысяч операторов языка программирования. Средств же, позволяющих детально описывать поведение сложных дискретных систем на более высоком уровне, чем универсальный язык программирования, не существует.

Коллективная разработка. Из-за больших объёмов проектов разработка программного обеспечения ведётся коллективом специалистов. Работая в коллективе, отдельные специалисты должны взаимодействовать друг с другом, обеспечивая целостность проекта, что при отсутствии удовлетворительных средств описания поведения сложных систем, упоминавшемся выше, достаточно сложно. Причём, чем больше коллектив разработчиков, тем сложнее организовать процесс работы.

Необходимость увеличения степени повторяемости кодов. На сложность разрабатываемого программного продукта влияет и то, что для увеличения производительности труда компании стремятся к созданию библиотек компонентов, которые можно было бы использовать в дальнейших разработках. Однако в этом случае компоненты приходится делать более универсальными, что в конечном итоге увеличивает сложность разработки.

Вместе взятые, эти факторы существенно увеличивают сложность процесса разработки. Однако очевидно, что все они напрямую связаны со сложностью объекта разработки – программной системы.

1.3. БЛОЧНО-ИЕРАРХИЧЕСКИЙ ПОДХОД К СОЗДАНИЮ СЛОЖНЫХ СИСТЕМ

Подавляющее большинство сложных систем как в природе, так и в технике имеет иерархическую внутреннюю структуру. Это связано с тем, что обычно связи элементов сложных систем различны как по типу, так и по силе, что и позволяет рассматривать эти системы как некоторую совокупность взаимозависимых подсистем. Внутренние связи элементов таких подсистем сильнее, чем связи между подсистемами. Например, компьютер состоит из процессора, памяти и внешних устройств, а Солнечная система включает Солнце и планеты, вращающиеся вокруг него.

В свою очередь, используя то же различие связей, можно каждую подсистему разделить на подсистемы и т.д. до самого нижнего «элементарного» уровня, причём выбор уровня, компоненты которого следует считать элементарными, остаётся за исследователем. На элементарном уровне система, как правило, состоит из немногих типов подсистем, по-разному скомбинированных и организованных. Иерархии такого типа получили название «целое-часть».

Поведение системы в целом обычно оказывается сложнее поведения отдельных частей, причём из-за более сильных внутренних связей особенности системы в основном обусловлены отношениями между её частями, а не частями как таковыми.

В природе существует ещё один вид иерархии – иерархия «простое-сложное» или иерархия развития (усложнения) систем в процессе эволюции. В этой иерархии любая функционирующая система является результатом развития более простой системы. Именно данный вид иерархии реализуется механизмом наследования объектно-ориентированного программирования.

Будучи в значительной степени отражением природных и технических систем, программные системы обычно являются иерархическими, т.е. обладают описанными выше свойствами. На этих свойствах иерархических систем строится блочно-иерархический подход к их исследованию или созданию. Этот подход предполагает сначала создавать части таких объектов (блоки, модули), а затем собирать из них сам объект.

Процесс разбиения сложного объекта на сравнительно независимые части получил название декомпозиции. При декомпозиции учитывают, что связи между отдельными частями должны быть слабее, чем связи элементов внутри частей. Кроме того, чтобы из полученных частей можно было собрать разрабатываемый объект, в процессе декомпозиции необходимо определить все виды связей частей между собой.

При создании очень сложных объектов процесс декомпозиции выполняется многократно: каждый блок, в свою очередь, декомпозируют на части, пока не получают блоки, которые сравнительно легко разработать. Данный метод разработки получил название пошаговой детализации.

Существенно и то, что в процессе декомпозиции стараются выделить аналогичные блоки, которые можно было бы разрабатывать на общей основе. Таким образом, как уже упоминалось выше, обеспечивают увеличение степени повторяемости кодов и, соответственно, снижение стоимости разработки.

Результат декомпозиции обычно представляют в виде схемы иерархии, на нижнем уровне которой располагают сравнительно простые блоки, а на верхнем – объект, подлежащий разработке. На каждом иерархическом уровне описание блоков выполняют с определённой степенью детализации, абстрагируясь от несущественных деталей. Следовательно, для каждого уровня используют свои формы документации и свои модели, отражающие сущность процессов, выполняемых каждым блоком. Так для объекта в целом, как правило, удаётся сформулировать лишь самые общие требования, а блоки нижнего уровня должны быть специфицированы так, чтобы из них действительно можно было собрать работающий объект. Другими словами, чем больше блок, тем более абстрактным должно быть его описание (рис. 1.8).

При соблюдении этого принципа разработчик сохраняет возможность осмысления проекта и, следовательно, может принимать наиболее правильные решения на каждом этапе, что называют локальной оптимизацией (в отличие от глобальной оптимизации характеристик объектов, которая для действительно сложных объектов не всегда возможна).

Итак, в основе блочно-иерархического подхода лежат декомпозиция и иерархическое упорядочение. Важную роль играют также следующие принципы:

– формализация – строгость методического подхода;

– повторяемость – необходимость выделения одинаковых блоков для удешевления и ускорения разработки;

– локальная оптимизация – оптимизация в пределах уровня иерархии;

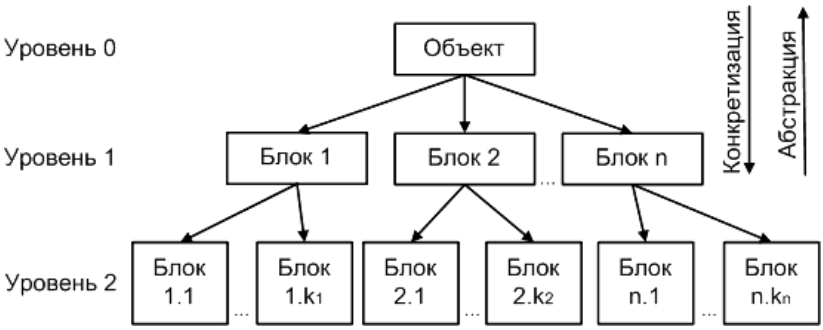


Рис. 1.8. Соотношение абстрактного и конкретного в описании блоков при блочно-иерархическом подходе

– непротиворечивость – контроль согласованности элементов между собой;

– полнота – контроль на присутствие лишних элементов.

Совокупность языков моделей, постановок задач, методов описаний некоторого иерархического уровня принято называть уровнем проектирования.

Каждый объект в процессе проектирования, как правило, приходится рассматривать с нескольких сторон. Различные взгляды на объект проектирования принято называть аспектами проектирования.

Помимо того, что использование блочно-иерархического подхода делает возможным создание сложных систем, он также:

– упрощает проверку работоспособности как системы в целом, так и отдельных блоков;

– обеспечивает возможность модернизации систем, например, замены ненадёжных блоков с сохранением их интерфейсов.

Необходимо отметить, что использование блочно-иерархического подхода применительно к программным системам стало возможным только после конкретизации общих положений подхода и внесения некоторых изменений в процесс проектирования. При этом структурный подход учитывает только свойства иерархии «целое-часть», а объектный – использует ещё и свойства иерархии «простое-сложное».

1.4. ЖИЗНЕННЫЙ ЦИКЛ И ЭТАПЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Жизненным циклом программного обеспечения называют период от момента появления идеи создания некоторого программного обеспечения до момента завершения его поддержки фирмой-разработчиком или фирмой, выполнявшей сопровождение.

Состав процессов жизненного цикла регламентируется международным стандартом ISO/IEC 12207:1995 «Information Technologe – Software Life Cycle Processes» («Информационные технологии – Процессы жизненного цикла программного обеспечения»). ISO (International Organization for Standardization) – Международная организация по стандартизации. IEC (International Electrotechnical Commission) – Международная комиссия по электротехнике.

Этот стандарт описывает структуру жизненного цикла программного обеспечения и его процессы. Процесс жизненного цикла определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные.

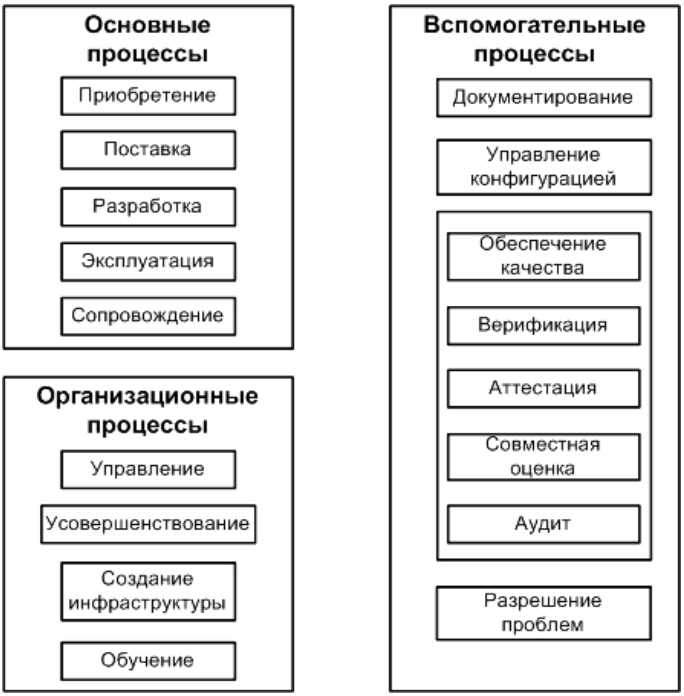


Рис. 1.9. Структура процессов жизненного цикла программного обеспечения

На рисунке 1.9 представлены процессы жизненного цикла по указанному стандарту. Каждый процесс характеризуется определёнными задачами и методами их решения, а также исходными данными и результатами.

Процесс разработки в соответствии со стандартом предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию программного обеспечения и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, а также подготовку материалов, необходимых для проверки работоспособности и соответствия качества программных продуктов, материалов, необходимых для обучения персонала, и т.д.

По стандарту процесс разработки включает следующие действия:

– подготовительную работу – выбор модели жизненного цикла, стандартов, методов и средств разработки, а также составление плана работ;

– анализ требований к системе – определение её функциональных возможностей, пользовательских требований, требований к надёжности и безопасности, требований к внешним интерфейсам и т.д.;

– проектирование архитектуры системы – определение состава необходимого оборудования, программного обеспечения и операций, выполняемых обслуживающим персоналом;

– анализ требований к программному обеспечению – определение функциональных возможностей, включая характеристики производительности, среды функционирования компонентов, внешних интерфейсов, спецификаций надёжности и безопасности, эргономических требований, требований к используемым данным, установке, приёмке, пользовательской документации, эксплуатации и сопровождению;

– проектирование архитектуры программного обеспечения – определение структуры программного обеспечения, документирование интерфейсов его компонентов, разработку предварительной версии пользовательской документации, а также требований к тестам и плана интеграции;

– детальное проектирование программного обеспечения – подробное описание компонентов программного обеспечения и интерфейсов между ними, обновление пользовательской документации, разработка и документирование требований к тестам и плана тестирования компонентов программного обеспечения, обновление плана интеграции компонентов;

– кодирование и тестирование программного обеспечения – разработку и документирование каждого компонента, а также совокупности тестовых процедур и данных для их тестирования, тестирование компонентов, обновление пользовательской документации, обновление плана интеграции программного обеспечения;

– интеграцию программного обеспечения – сборку программных компонентов в соответствии с планом интеграции и тестирование программного обеспечения на соответствие квалификационным требованиям, представляющих собой набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт, как соответствующий своим спецификациям и готовый к использованию в заданных условиях эксплуатации;

– квалификационное тестирование программного обеспечения – тестирование программного обеспечения в присутствии заказчика для демонстрации его соответствия требованиям и готовности к эксплуатации; при этом проверяются также готовность и полнота технической и пользовательской документации;

– интеграцию системы – сборку всех компонентов системы, включая программное обеспечение и оборудование;

– квалификационное тестирование системы – тестирование системы на соответствие требованиям к ней и проверку оформления и полноты документации;

– установку программного обеспечения – установку программного обеспечения на оборудовании заказчика и проверку его работоспособности;

– приёмку программного обеспечения – оценку результатов квалификационного тестирования программного обеспечения и системы в целом и документирование результатов оценки совместно с заказчиком, окончательную передачу программного обеспечения заказчику.

Указанные действия можно сгруппировать, условно выделив следующие основные этапы разработки программного обеспечения (в скобках указаны соответствующие стадии разработки по ГОСТ 19.102–77 «Стадии разработки»):

– постановка задачи (стадия «Техническое задание»);

– анализ требований и разработка спецификаций (стадия «Эскизный проект»);

– проектирование (стадия «Технический проект»);

– реализация (стадия «Рабочий проект»).

Традиционно разработка также включала этап сопровождения (началу этого этапа соответствует стадия «Внедрение» по ГОСТ). Однако по международному стандарту в соответствии с изменениями, произошедшими в индустрии разработки программного обеспечения, этот процесс теперь рассматривается отдельно.

Условность выделения этапов связана с тем, что на любом этапе возможно принятие решений, которые потребуют пересмотра решений, принятых ранее.

Постановка задачи. В процессе постановки задачи чётко формулируют назначение программного обеспечения и определяют основные требования к нему. Каждое требование представляет собой описание необходимого или желаемого свойства программного обеспечения. Различают функциональные требования, определяющие функции, которые должно выполнять разрабатываемое программное обеспечение, и эксплуатационные требования, определяющие особенности его функционирования.

Требования к программному обеспечению, имеющему прототипы, обычно определяют по аналогии, учитывая структуру и характеристики уже существующего программного обеспечения. Для формулирования требований к программному обеспечению, не имеющему аналогов, иногда необходимо провести специальные исследования, назы-

ваемые предпроектными. В процессе таких исследований определяют разрешимость задачи, возможно, разрабатывают методы её решения (если они новые) и устанавливают наиболее существенные характеристики разрабатываемого программного обеспечения. Для выполнения предпроектных исследований, как правило, заключают договор на выполнение научно-исследовательских работ. В любом случае этап постановки задачи заканчивается разработкой технического задания, фиксирующего принципиальные требования, и принятием основных проектных решений.

Анализ требований и определение спецификаций. Спецификациями называют точное формализованное описание функций и ограничений разрабатываемого программного обеспечения. Соответственно различают функциональные и эксплуатационные спецификации. Совокупность спецификаций представляет собой общую логическую модель проектируемого программного обеспечения.

Для получения спецификаций выполняют анализ требований технического задания, формулируют содержательную постановку задачи, выбирают математический аппарат формализации, строят модель предметной области, определяют подзадачи и выбирают или разрабатывают методы их решения. Часть спецификаций может быть определена в процессе предпроектных исследований и, соответственно, зафиксирована в техническом задании.

На этом этапе также целесообразно сформировать тесты для поиска ошибок в проектируемом программном обеспечении, обязательно указав ожидаемые результаты.

Проектирование. Основной задачей этого этапа является определение подробных спецификаций разрабатываемого программного обеспечения. Процесс проектирования сложного программного обеспечения обычно включает:

– проектирование общей структуры – определение основных компонентов и их взаимосвязей;

– декомпозицию компонентов и построение структурных иерархий в соответствии с рекомендациями блочно-иерархического подхода;

– проектирование компонентов.

Результатом проектирования является детальная модель разрабатываемого программного обеспечения вместе со спецификациями его компонентов всех уровней. Тип модели зависит от выбранного подхода (структурный, объектный или компонентный) и конкретной технологии проектирования. Однако в любом случае процесс проектирования охватывает как проектирование программ (подпрограмм) и определение взаимосвязей между ними, так и проектирование данных, с которыми взаимодействуют эти программы или подпрограммы.

Принято различать также два аспекта проектирования:

– логическое проектирование, которое включает те проектные операции, которые непосредственно не зависят от имеющихся технических и программных средств, составляющих среду функционирования будущего программного продукта;

– физическое проектирование – привязку к конкретным техническим и программным средствам среды функционирования, т.е. учёт ограничений, определённых в спецификациях.

Реализация. Реализация представляет собой процесс поэтапного написания кодов программы на выбранном языке программирования (кодирование), их тестирование и отладку.

Сопровождение. Сопровождение – это процесс создания и внедрения новых версий программного продукта. Причинами выпуска новых версий могут служить:

– необходимость исправления ошибок, выявленных в процессе эксплуатации предыдущих версий;

– необходимость совершенствования предыдущих версий, например, улучшения интерфейса, расширения состава выполняемых функций или повышения его производительности;

– изменение среды функционирования, например, появление новых технических средств и/или программных продуктов, с которыми взаимодействует сопровождаемое программное обеспечение.

На этом этапе в программный продукт вносят необходимые изменения, которые так же, как в остальных случаях, могут потребовать пересмотра проектных решений, принятых на любом предыдущем этапе. С изменением модели жизненного цикла программного обеспечения роль этого этапа существенно возросла, так как продукты теперь создаются итерационно: сначала выпускается сравнительно простая версия, затем следующая с большими возможностями, затем следующая и т.д. Именно это и послужило причиной выделения этапа сопровождения в отдельный процесс жизненного цикла в соответствии со стандартом ISO/IEC 12207.

1.5. ЭВОЛЮЦИЯ МОДЕЛЕЙ ЖИЗНЕННОГО ЦИКЛА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

На протяжении последних тридцати лет в программировании сменились три модели жизненного цикла программного обеспечения: каскадная, модель с промежуточным контролем и спиральная [1, 2].

Каскадная модель. Первоначально (1970 – 1985 гг.) была предложена и использовалась каскадная схема разработки программного обеспечения (рис. 1.10), которая предполагала, что переход на следующую

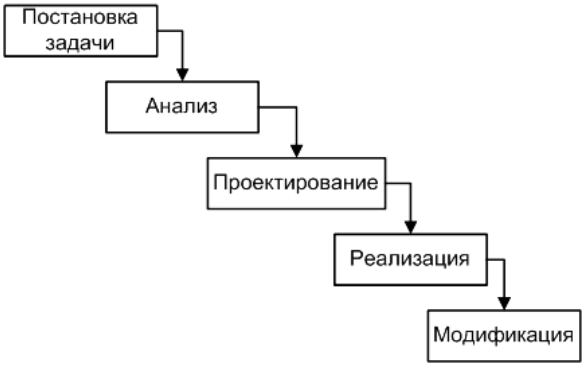


Рис. 1.10. Каскадная схема разработки программного обеспечения

стадию осуществляется после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии. Достоинствами такой схемы являются:

– получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности;

– простота планирования процесса разработки.

Именно такую схему и используют обычно при блочно-иерархическом подходе к разработке сложных технических объектов, обеспечивая очень высокие параметры эффективности разработки. Однако данная схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования. Это уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречаются крайне редко.

В целом необходимость возвратов на предыдущие стадии обусловлена следующими причинами:

– неточные спецификации, уточнение которых в процессе разработки может привести к необходимости пересмотра уже принятых решений;

– изменение требований заказчика непосредственно в процессе разработки;

– быстрое моральное устаревание используемых технических и программных средств;

– отсутствие удовлетворительных средств описания разработки на стадиях постановки задачи, анализа и проектирования.

Отказ от уточнения (изменения) спецификаций приведёт к тому, что законченный продукт не будет удовлетворять потребности пользователей. При отказе от учёта смены оборудования и программной среды пользователь получит морально устаревший продукт. А отказ от пересмотра неудачных проектных решений приводит к ухудшению структуры программного продукта и, соответственно, усложнит, растянет по времени и удорожит процесс его создания. Реальный процесс разработки, таким образом, носит итерационный характер.

Модель с промежуточным контролем. Схема, поддерживающая итерационный характер процесса разработки, была названа схемой с промежуточным контролем (рис. 1.11). Контроль, который выполняется по данной схеме после завершения каждого этапа, позволяет при необходимости вернуться на любой уровень и внести необходимые изменения.

Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

Спиральная модель. Для преодоления перечисленных проблем в середине 80-х гг. XX в. была предложена спиральная схема (рис. 1.12). В соответствии с данной схемой программное обеспечение создаётся не сразу, а итерационно с использованием метода прототипирования, базирующегося на создании прототипов. Именно появление прототипирования привело к тому, что процесс модификации программного обеспечения перестал восприниматься как «необходимое зло», а стал восприниматься как отдельный важный процесс.

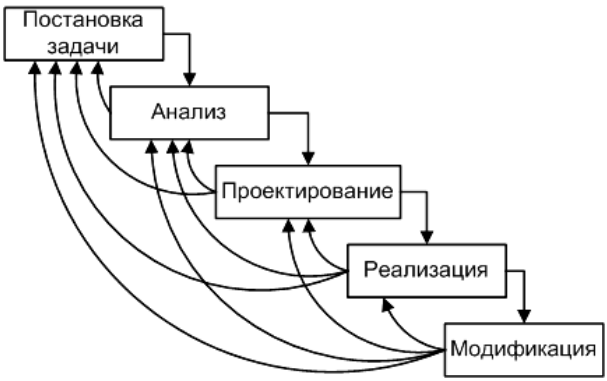


Рис. 1.11. Схема разработки программного обеспечения с промежуточным контролем

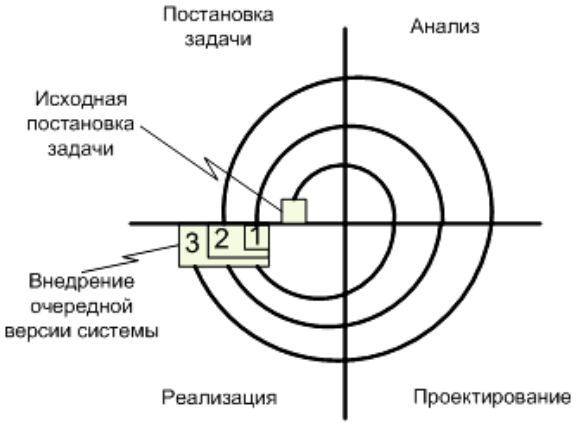


Рис. 1.12. Спиральная или итерационная схема разработки программного обеспечения

Прототипом называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемого программного обеспечения.

На первой итерации, как правило, специфицируют, проектируют, реализуют и тестируют интерфейс пользователя. На второй – добавляют некоторый ограниченный набор функций. На последующих этапах этот набор расширяют, наращивая возможности данного продукта.

Основным достоинством данной схемы является то, что, начиная с некоторой итерации, на которой обеспечена определённая функциональная полнота, продукт можно предоставлять пользователю, что позволяет:

– сократить время до появления первых версий программного продукта;

– заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;

– ускорить формирование и уточнение спецификаций за счёт появления практики использования продукта;

– уменьшить вероятность морального устаревания системы за время разработки.

Основной проблемой использования спиральной схемы является определение моментов перехода на следующие стадии. Для её решения обычно ограничивают сроки прохождения каждой стадии, основываясь на экспертных оценках.

1.6. ОЦЕНКА КАЧЕСТВА ПРОЦЕССОВ СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Текущий период на рынке программного обеспечения характеризуется переходом от штучного ремесленного производства программных продуктов к их промышленному созданию. Соответственно возросли требования к качеству разрабатываемого программного обеспечения, что требует совершенствования процессов их разработки. На настоящий момент существует несколько стандартов, связанных с оценкой качества этих процессов, которое обеспечивает организация-разработчик. К наиболее известным относят:

– международные стандарты серии ISO 9000 (ISO 9000 – ISO 9004);

– CMM (Capability Maturity Model) – модель зрелости (совершенствования) процессов создания программного обеспечения, предложенная SEI (Software Engineering Institute – институт программирования при университете Карнеги–Меллон);

– рабочая версия международного стандарта ISO/IEC 15504: Information Technology – Software Process Assessment; эта версия более известна под названием SPICE (Software Process Improvement and Capability dEtermination – определение возможностей и улучшение процесса создания программного обеспечения).

Единый стандарт оценки программных процессов SPICE предполагает обеспечение постоянного улучшения процессов разработки программного обеспечения и может быть применен не только к организации в целом, но и к отдельно взятым процессам. Стандарт позволяет проводить оценку проектирования программного обеспечения и при этом выявлять возможности улучшения процесса. В некоторых случаях стандарт имеет преимущество перед группой стандартов ISO 9000, поскольку предоставляет более полный набор средств по обеспечению качества и улучшению процессов.

Оба направления зарекомендовали себя как достаточно жизнеспособные, но имеющие ряд недостатков. Общие принципы управления процессами, изложенные в требованиях стандартов семейства ISO 9000, дают возможность выбрать наиболее подходящий метод обеспечения и оценки качества процесса проектирования конкретного программного продукта и способствуют развитию новых методов управления и оценки. В то же время SPICE даёт возможность существенно сократить время на «отслеживание» и оценку процессов проектирования за счёт конкретных методик, но не позволяет рассматривать процесс проектирования программного продукта и качество самого продукта как систему.

В то же время существуют и разнообразные методы оценки программных средств, которые должны обеспечивать требования различных групп потребителей. Следует отметить, что в данном случае круг потребителей такого рода продукции несколько расширен, что связано со специфическими особенностями программного обеспечения (ПО), поскольку к «классическим» группам потребителей (государство, организация, конкретный пользователь) необходимо добавить составляющую внешней среды (например, при использовании во время функционирования ПО локальных, глобальных сетей и т.п.) и потребителей, обслуживающих ПО.

Для определения групп потребительских свойств могут быть использованы как отечественные, так и международные нормативные документы. Например, российские стандарты предлагают использовать следующие группы и комплексные показатели качества:

– показатели надёжности программных средств (ПС) (устойчивость функционирования, работоспособность);

– показатели сопровождения (структурность, простота конструкции, наглядность, повторяемость);

– показатели удобства применения (лёгкость освоения, доступность эксплуатационных программных документов, удобство эксплуатации и обслуживания);

– показатели эффективности (уровень автоматизации, временная эффективность, ресурсоёмкость);

– показатели универсальности (гибкость, мобильность, модифицируемость);

– показатели корректности (полнота реализации, согласованность, логическая корректность, проверенность).

Согласно ГОСТ Р ИСО 9126 следует обеспечивать следующие качественные характеристики программных средств:

– функциональные возможности;
– функциональную пригодность;
– правильность (корректность);
– способность к взаимодействию;
– защищённость;
– надёжность;
– эффективность;
– практичность (применимость);
– сопровождаемость;
– мобильность.

2. ПРИЁМЫ ОБЕСПЕЧЕНИЯ ТЕХНОЛОГИЧНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В условиях индустриального подхода к разработке и сопровождению программного обеспечения особый вес приобретают технологические характеристики разрабатываемых программ. Для обеспечения необходимых технологических свойств применяют специальные технологические приёмы и следуют определённым методикам, сформулированным всем предыдущим опытом создания программного обеспечения. К таким приёмам и методикам относят правила декомпозиции, методы проектирования, программирования и контроля качества, которые под общим названием «структурный подход к программированию» были сформулированы ещё в 60-х гг. XX в. В его основу были положены следующие основные концепции:

- нисходящая разработка;
- модульное программирование;
- структурное программирование;
- сквозной структурный контроль.

2.1. ПОНЯТИЕ ТЕХНОЛОГИЧНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Под технологичностью понимают качество проекта программного продукта, от которого зависят трудовые и материальные затраты на его реализацию и последующие модификации.

Хороший проект сравнительно быстро и легко кодируется, тестируется, отлаживается и модифицируется.

Технологичность программного обеспечения определяется проработанностью его моделей, уровнем независимости модулей, стилем программирования и степенью повторного использования кодов.

Чем лучше проработана модель разрабатываемого программного обеспечения, тем чётче определены подзадачи и структуры данных, хранящие входную, промежуточную и выходную информацию, тем проще их проектирование и реализация и меньше вероятность ошибок, для исправления которых потребуется существенно изменять программу.

Чем выше независимость модулей, тем их легче понять, реализовать, модифицировать, а также находить в них ошибки и исправлять их.

Стиль программирования, под которым понимают стиль оформления программ и их «структурность», также существенно влияет на читаемость программного кода и количество ошибок программирования.

Увеличение степени повторного использования кодов предполагает как использование ранее разработанных библиотек подпрограмм или классов, так и унификацию кодов текущей разработки. Причём для данного критерия ситуация не так однозначна, как в предыдущих случаях: если степень повторного использования кодов повышается искусственно (например, путём разработки «суперуниверсальных» процедур), то технологичность проекта может существенно снизиться.

Как следует из определения, высокая технологичность проекта особенно важна, если разрабатывается программный продукт, рассчитанный на многолетнее интенсивное использование, или необходимо обеспечить повышенные требования к его качеству.

2.2. МОДУЛИ И ИХ СВОЙСТВА

При проектировании достаточно сложного программного обеспечения после определения его общей структуры выполняют декомпозицию компонентов в соответствии с выбранным подходом до получения элементов, которые, по мнению проектировщика, в дальнейшей декомпозиции не нуждаются.

Как уже упоминалось раньше, в настоящее время используют два способа декомпозиции разрабатываемого программного обеспечения, связанные с соответствующим подходом:

– процедурный (или структурный – по названию подхода);
– объектный.

Результатом процедурной декомпозиции является иерархия подпрограмм (процедур), в которой функции, связанные с принятием решения, реализуются подпрограммами верхних уровней, а непосредственно обработка – подпрограммами нижних уровней. Это согласуется с принципом вертикального управления, который был сформулирован вместе с другими рекомендациями структурного подхода к программированию. Он также ограничивает возможные варианты передачи управления, требуя, чтобы любая подпрограмма возвращала управление той подпрограмме, которая её вызвала.

Результатом объектной декомпозиции является совокупность объектов, которые затем реализуют как переменные некоторых специально разрабатываемых типов (классов), представляющих собой совокупность полей данных и методов, работающих с этими полями.

Таким образом, при любом способе декомпозиции получают набор связанных с соответствующими данными подпрограмм, которые в процессе реализации организуют в модули.

Модули. Модулем называют автономно компилируемую программную единицу. Термин «модуль» традиционно используется в двух смыслах. Первоначально, когда размер программ был сравни-

тельно невелик и все подпрограммы компилировались отдельно, под модулем понималась подпрограмма, т.е. последовательность связанных фрагментов программы, обращение к которой выполняется по имени. Со временем, когда размер программ значительно вырос и появилась возможность создавать библиотеки ресурсов: констант, переменных, описаний типов, классов и подпрограмм, термин «модуль» стал использоваться и в смысле автономно компилируемого набора программных ресурсов. Данные модуль может получать и/или возвращать через общие области памяти или параметры.

Первоначально к модулям (ещё понимаемым как подпрограммы) предъявлялись следующие требования:

- отдельная компиляция;
- одна точка входа;
- одна точка выхода;
- соответствие принципу вертикального управления;
- возможность вызова других модулей;
- небольшой размер (до 50 – 60 операторов языка);
- независимость от истории вызовов;
- выполнение одной функции.

Требования одной точки входа, одной точки выхода, независимости от истории вызовов и соответствия принципу вертикального управления были вызваны тем, что в то время из-за серьёзных ограничений на объём оперативной памяти программисты были вынуждены разрабатывать программы с максимально возможной повторяемостью кодов. В результате подпрограммы, имеющие несколько точек входа и выхода, были не только обычным явлением, но и считались высоким классом программирования. Следствием же было то, что программы было очень сложно не только модифицировать, но и понять, а иногда и просто полностью отладить.

Со временем, когда основные требования структурного подхода стали поддерживаться языками программирования и под модулем стали понимать отдельно компилируемую библиотеку ресурсов, требование независимости модулей стало основным.

Практика показала, что чем выше степень независимости модулей, тем:

– легче разобраться в отдельном модуле и всей программе и, соответственно, тестировать, отлаживать и модифицировать её;

– меньше вероятность появления новых ошибок при исправлении старых или внесении изменений в программу, т.е. вероятность появления «волнового» эффекта;

– проще организовать разработку программного обеспечения группой программистов и легче его сопровождать.

Таким образом, уменьшение зависимости модулей улучшает технологичность проекта.

Степень независимости модулей (как подпрограмм, так и библиотек) оценивают двумя критериями: сцеплением и связностью.

Сцепление модулей. Сцепление является мерой взаимозависимости модулей, которая определяет, насколько хорошо модули отделены друг от друга. Модули независимы, если каждый из них не содержит о другом никакой информации. Чем больше информации о других модулях хранит модуль, тем больше он с ними сцеплен.

Различают пять типов сцепления модулей:

- по данным;
- по образцу;
- по управлению;
- по общей области данных;
- по содержимому.

Сцепление по данным предполагает, что модули обмениваются данными, представленными скалярными значениями. При небольшом количестве передаваемых параметров этот тип обеспечивает наилучшие технологические характеристики программного обеспечения.

Сцепление по образцу предполагает, что модули обмениваются данными, объединёнными в структуры. Этот тип также обеспечивает неплохие характеристики, но они хуже, чем у предыдущего типа, так как конкретные передаваемые данные «спрятаны» в структуры, и потому уменьшается «прозрачность» связи между модулями. Кроме того, при изменении структуры передаваемых данных необходимо модифицировать все использующие её модули.

При сцеплении по управлению один модуль посылает другому некоторый информационный объект (флаг), предназначенный для управления внутренней логикой модуля. Таким способом часто выполняют настройку режимов работы программного обеспечения. Подобные настройки также снижают наглядность взаимодействия модулей и потому обеспечивают ещё худшие характеристики технологичности разрабатываемого программного обеспечения по сравнению с предыдущими типами связей.

Сцепление по общей области данных предполагает, что модули работают с общей областью данных. Этот тип сцепления считается недопустимым, поскольку:

- программы, использующие данный тип сцепления, очень сложны для понимания при сопровождении программного обеспечения;
- ошибка одного модуля, приводящая к изменению общих данных, может проявиться при выполнении другого модуля, что существенно усложняет локализацию ошибок;

– при ссылке к данным в общей области модули используют конкретные имена, что уменьшает гибкость разрабатываемого программного обеспечения.

Следует иметь в виду, что «подпрограммы с памятью», действия которых зависят от истории вызовов, используют сцепление по общей области, что делает их работу в общем случае непредсказуемой. Именно этот вариант используют статические переменные C и C++.

В случае сцепления по содержимому один модуль содержит обращения к внутренним компонентам другого (передаёт управление внутрь, читает и/или изменяет внутренние данные или сами коды), что полностью противоречит блочно-иерархическому подходу. Отдельный модуль в этом случае уже не является блоком («черным ящиком»): его содержимое должно учитываться в процессе разработки другого модуля. Современные универсальные языки процедурного программирования, например Pascal, данного типа сцепления в явном виде не поддерживают, но для языков низкого уровня, например Ассемблера, такой вид сцепления остаётся возможным.

В таблице 2.1 приведены характеристики различных типов сцепления по экспертным оценкам. Допустимыми считают первые три типа сцепления, так как использование остальных приводит к резкому ухудшению технологичности программ.

Как правило, модули сцепляются между собой несколькими способами. Учитывая это, качество программного обеспечения принято определять по типу сцепления с худшими характеристиками. Так, если использовано сцепление по данным и сцепление по управлению, то определяющим считают сцепление по управлению.

Таблица 2.1

Тип сцепления	Сцепление, балл	Устойчивость к ошибкам других модулей	Наглядность (понятность)	Возможность изменения	Вероятность повторного использования
По данным	1	Хорошая	Хорошая	Хорошая	Большая
По образцу	3	Средняя	Хорошая	Средняя	Средняя
По управлению	4	Средняя	Плохая	Плохая	Малая
По общей области	6	Плохая	Плохая	Средняя	Малая
По содержимому	10	Плохая	Плохая	Плохая	Малая

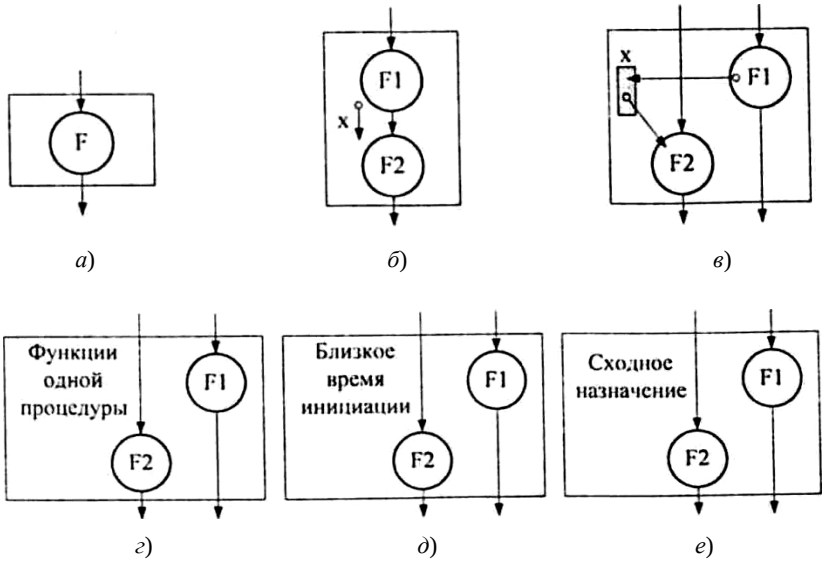
В некоторых случаях сцепление модулей можно уменьшить, удалив необязательные связи и структурировав необходимые связи. Примером может служить объектно-ориентированное программирование, в котором вместо большого количества параметров метод неявно получает адрес области (структуры), в которой расположены поля объекта, и явно-дополнительные параметры. В результате модули оказываются сцепленными по образцу.

Связность модулей. Связность – мера прочности соединения функциональных и информационных объектов внутри одного модуля. Если сцепление характеризует качество отделения модулей, то связность характеризует степень взаимосвязи элементов, реализуемых одним модулем. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи и, соответственно, взаимовлияние модулей. В то же время помещение сильно связанных элементов в разные модули не только усиливает межмодульные связи, но и усложняет понимание их взаимодействия. Объединение слабо связанных элементов также уменьшает технологичность модулей, так как такими элементами сложнее мысленно манипулировать.

Различают следующие виды связности (в порядке убывания уровня):

- функциональную;
- последовательную;
- информационную (коммуникативную);
- процедурную;
- временную;
- логическую;
- случайную.

При функциональной связности все объекты модуля предназначены для выполнения одной функции (рис. 2.1, *а*): операции, объединяемые для выполнения одной функции, или данные, связанные с одной функцией. Модуль, элементы которого связаны функционально, имеет чётко определённую цель, при его вызове выполняется одна задача, например, подпрограмма поиска минимального элемента массива. Такой модуль имеет максимальную связность, следствием которой являются его хорошие технологические качества: простота тестирования, модификации и сопровождения. Именно с этим связано одно из требований структурной декомпозиции «один модуль – одна связь между модулями-библиотеками ресурсов». Например, если при проектировании текстового редактора предполагается функция редактирования, то лучше организовать модуль-библиотеку функций редактирования, чем поместить часть функций в один модуль, а часть в другой.



а) *б*) *в*)

г) *д*) *е*)

Рис. 2.1. Связность модулей:
а – функциональная; *б* – последовательная; *в* – информационная; *г* – процедурная; *д* – временная; *е* – логическая

При последовательной связности функций выход одной функции служит исходными данными для другой функции (рис. 2.1, *б*). Как правило, такой модуль имеет одну точку входа, т.е. реализует одну подпрограмму, выполняющую две функции. Считают, что данные, используемые последовательными функциями, также связаны последовательно. Модуль с последовательной связностью функций можно разбить на два или более модулей, как с последовательной, так и с функциональной связностью. Такой модуль выполняет несколько функций, и, следовательно, его технологичность хуже: сложнее организовать тестирование, а при выполнении модификации мысленно приходится разделять функции модуля.

Информационно связанными считают функции, обрабатывающие одни и те же данные (рис. 2.1, *в*). При использовании структурных языков программирования раздельное выполнение функций можно осуществить только, если каждая функция реализуется своей подпрограммой.

Несмотря на объединение нескольких функций, информационно связанный модуль имеет неплохие показатели технологичности. Это объясняется тем, что все функции, работающие с некоторыми данны-

ми, собраны в одно место, что позволяет при изменении формата данных корректировать только один модуль. Информационно связанными также считают данные, которые обрабатываются одной функцией.

Процедурно связаны функции или данные, которые являются частями одного процесса (рис. 2.1, *г*). Обычно модули с процедурной связностью функций получают, если в модуле объединены функции альтернативных частей программы. При процедурной связности отдельные элементы модуля связаны крайне слабо, так как реализуемые ими действия связаны лишь общим процессом, следовательно, технологичность данного вида связи ниже, чем предыдущего.

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени (рис. 2.1, *д*). Временная связность данных означает, что они используются в некотором временном интервале. Например, временную связность имеют функции, выполняемые при инициализации некоторого процесса. Отличительной особенностью временной связности является то, что действия, реализуемые такими функциями, обычно могут выполняться в любом порядке. Содержание модуля с временной связностью функций имеет тенденцию меняться: в него могут включаться новые действия и/или исключаться старые. Большая вероятность модификации функции ещё больше уменьшает показатели технологичности модулей данного вида по сравнению с предыдущим.

Логическая связь базируется на объединении данных или функций в одну логическую группу (рис. 2.1, *е*). В качестве примера можно привести функции обработки текстовой информации или данные одного и того же типа. Модуль с логической связностью функций часто реализует альтернативные варианты одной операции, например, сложение целых чисел и сложение вещественных чисел. Из такого модуля всегда будет вызываться одна какая-либо его часть, при этом вызывающий и вызываемый модули будут связаны по управлению. Понять логику работы модулей, содержащих логически связанные компоненты, как правило, сложнее, чем модулей, использующих временную связность, следовательно, их показатели технологичности ещё ниже.

В том случае, если связь между элементами мала или отсутствует, считают, что они имеют случайную связность. Модуль, элементы которого связаны случайно, имеет самые низкие показатели технологичности, так как элементы, объединённые в нём, вообще не связаны.

В трёх предпоследних случаях связь между несколькими подпрограммами в модуле обусловлена внешними причинами, а в последнем – вообще отсутствует. Это соответствующим образом проецируется на технологические характеристики модулей. В таблице 2.2 представлены характеристики различных видов связности по экспертным оценкам.

Таблица 2.2

Вид связности	Сцепление, балл	Наглядность (понятность)	Возможность изменения	Сопровождаемость
Функциональная	10	Хорошая	Хорошая	Хорошая
Последовательная	9	Хорошая	Хорошая	Хорошая
Информационная	8	Средняя	Средняя	Средняя
Процедурная	5	Средняя	Средняя	Плохая
Временная	3	Средняя	Средняя	Плохая
Логическая	1	Плохая	Плохая	Плохая
Случайная	0	Плохая	Плохая	Плохая

Анализ табл. 2.2 показывает, что на практике целесообразно использовать функциональную, последовательную и информационную связности.

Как правило, при хорошо продуманной декомпозиции модули верхних уровней иерархии имеют функциональную или последовательную связность функций и данных. Для модулей обслуживания данных характерна информационная связность функций. Данные таких модулей могут быть связаны по-разному. Так, модули, содержащие описание классов при объектно-ориентированном подходе, характеризуются информационной связностью методов и функциональной связностью данных. Получение в процессе декомпозиции модулей с другими видами связности, скорее всего, означает недостаточно продуманное проектирование. Исключением являются лишь библиотеки ресурсов.

Библиотеки ресурсов. Различают библиотеки ресурсов двух типов: библиотеки подпрограмм и библиотеки классов.

Библиотеки подпрограмм реализуют функции, близкие по назначению, например, библиотека графического вывода информации. Связность подпрограмм между собой в такой библиотеке – логическая, а связность самих подпрограмм – функциональная, так как каждая из них обычно реализует одну функцию.

Библиотеки классов реализуют близкие по назначению классы. Связность элементов класса – информационная, связность классов между собой может быть функциональной – для родственных или ассоциированных классов и логической – для остальных.

В качестве средства улучшения технологических характеристик библиотек ресурсов в настоящее время широко используют разделение тела модуля на интерфейсную часть и область реализации.

Интерфейсная часть в данном случае содержит совокупность объявлений ресурсов (заголовков подпрограмм, имен переменных, типов, классов и т.п.), которые данная библиотека предоставляет другим модулям. Ресурсы, объявление которых в интерфейсной части отсутствует, извне не доступны. Область реализации содержит тела подпрограмм и, возможно, внутренние ресурсы (подпрограммы, переменные, типы), используемые этими подпрограммами. При такой организации любые изменения реализации библиотеки, не затрагивающие её интерфейс, не требуют пересмотра модулей, связанных с библиотекой, что улучшает технологические характеристики модулей-библиотек. Кроме того, подобные библиотеки, как правило, хорошо отлажены и продуманы, так как часто используются разными программами.

2.3. НИСХОДЯЩАЯ И ВОСХОДЯЩАЯ РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

При проектировании, реализации и тестировании компонентов структурной иерархии, полученной при декомпозиции, применяют два подхода:

– восходящий;
– нисходящий.

В литературе встречается ещё один подход, получивший название «расширение ядра». Он предполагает, что в первую очередь проектируют и разрабатывают некоторую основу – ядро программного обеспечения, например, структуры данных и процедуры, связанные с ними. В дальнейшем ядро наращивают, комбинируя восходящий и нисходящий методы. На практике данный подход в зависимости от уровня ядра практически сводится либо к нисходящему, либо к восходящему подходам.

Восходящий подход. При использовании восходящего подхода сначала проектируют и реализуют компоненты нижнего уровня, затем предыдущего и т.д. По мере завершения тестирования и отладки компонентов осуществляют их сборку, причём компоненты нижнего уровня при таком подходе часто помещают в библиотеки компонентов.

Для тестирования и отладки компонентов проектируют и реализуют специальные тестирующие программы. Подход имеет следующие недостатки:

– увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;

– наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;

– позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т.д.

Исторически восходящий подход появился раньше, что связано с особенностью мышления программистов, которые в процессе обучения привыкают при написании небольших программ сначала детализировать компоненты нижних уровней (подпрограммы, классы). Это позволяет им лучше осознавать процессы верхних уровней. При промышленном изготовлении программного обеспечения восходящий подход в настоящее время практически не используют.

Нисходящий подход. Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняется «сверху-вниз», т.е. вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней. В той же последовательности выполняют и реализацию компонентов. При этом в процессе программирования компоненты нижних, ещё не реализованных уровней заменяют специально разработанными отладочными модулями – «заглушками», что позволяет тестировать и отлаживать уже реализованную часть.

При использовании нисходящего подхода применяют иерархический, операционный и комбинированный методы определения последовательности проектирования и реализации компонентов.

Иерархический метод предполагает выполнение разработки строго по уровням. Исключения допускаются при наличии зависимости по данным, т.е. если обнаруживается, что некоторый модуль использует результаты другого, то его рекомендуется программировать после этого модуля. Основной проблемой данного метода является большое количество достаточно сложных заглушек. Кроме того, при использовании данного метода основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.

Операционный метод связывает последовательность выполнения при запуске программы. Применение метода усложняется тем, что порядок выполнения модулей может зависеть от данных. Кроме того, модули вывода результатов, несмотря на то, что они вызываются последними, должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании. С точки зрения распределения человеческих ресурсов сложным является начало работ, пока не закончены все модули, находящиеся на так называемом критическом пути.

Комбинированный метод учитывает следующие факторы, влияющие на последовательность разработки:

– достижимость модуля – наличие всех модулей в цепочке вызова данного модуля;

– зависимость по данным – модули, формирующие некоторые данные, должны создаваться раньше обрабатывающих;

– обеспечение возможности выдачи результатов – модули вывода результатов должны создаваться раньше обрабатывающих;

– готовность вспомогательных модулей – вспомогательные модули, например модули закрытия файлов, завершения программы, должны создаваться раньше обрабатывающих;

– наличие необходимых ресурсов.

Кроме того, при прочих равных условиях сложные модули должны разрабатываться прежде простых, так как при их проектировании могут выявиться неточности в спецификациях, а чем раньше это произойдет, тем лучше.

Нисходящий подход допускает нарушение нисходящей последовательности разработки компонентов в специально оговоренных случаях. Так, если некоторый компонент нижнего уровня используется многими компонентами более высоких уровней, то его рекомендуют проектировать и разрабатывать раньше, чем вызывающие его компоненты. И, наконец, в первую очередь проектируют и реализуют компоненты, обеспечивающие обработку правильных данных, оставляя компоненты обработки неправильных данных напоследок.

Нисходящий подход обычно используют и при объектно-ориентированном программировании. В соответствии с рекомендациями подхода вначале проектируют и реализуют пользовательский интерфейс программного обеспечения, затем разрабатывают классы некоторых базовых объектов предметной области, а уже потом, используя эти объекты, проектируют и реализуют остальные компоненты. Нисходящий подход обеспечивает:

– максимально полное определение спецификаций проектируемого компонента и согласованность компонентов между собой;

– раннее определение интерфейса пользователя, демонстрация которого заказчику позволяет уточнить требования к создаваемому программному обеспечению:

– возможность нисходящего тестирования и комплексной отладки.

2.4. СТРУКТУРНОЕ И «НЕСТРУКТУРНОЕ» ПРОГРАММИРОВАНИЕ

Одним из способов обеспечения высокого уровня технологичности разрабатываемого программного обеспечения является структурное программирование.

Различают три вида вычислительного процесса, реализуемого программами: линейный, разветвлённый и циклический.

Линейная структура процесса вычислений предполагает, что для получения результата необходимо выполнить некоторые операции в определённой последовательности.

Разветвлённая структура процесса вычислений предполагает, что конкретная последовательность операций зависит от значений одной или нескольких переменных.

Циклическая структура процесса вычислений предполагает, что для получения результата некоторые действия необходимо выполнить несколько раз.

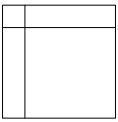
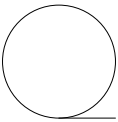
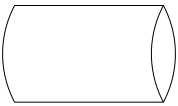
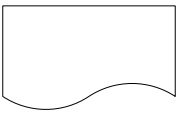
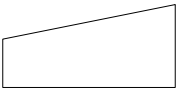
Для реализации указанных вычислительных процессов в программах используют соответствующие управляющие операторы. Первые процедурные языки программирования высокого уровня, такие как FORTRAN, понятием «тип вычислительного процесса» не оперировали. Для изменения линейной последовательности операторов в них, как в языках низкого уровня, использовались команды условной (при выполнении некоторого условия) и безусловной передач управления. Потому и программы, написанные на этих языках, имели запутанную структуру, присущую в настоящее время только низкоуровневым (машинным) языкам.

Именно для изображения схем алгоритмов таких программ в своё время был разработан ГОСТ 19.701–90, согласно которому каждой группе действий ставится в соответствие специальный блок (табл. 2.3). Хотя этот стандарт предусматривает блоки для обозначения циклов, он не запрещает и произвольной передачи управления, т.е. допускает использование команд условной и безусловной передачи управления при реализации алгоритма.

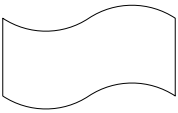
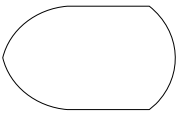
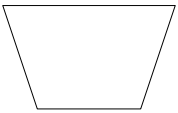
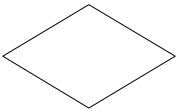
После того, как в 60-х гг. XX в. было доказано, что любой сколь угодно сложный алгоритм можно представить с использованием трёх основных управляющих конструкций, в языках программирования высокого уровня появились управляющие операторы для реализации соответствующих конструкций. Эти три конструкции принято считать базовыми. К ним относят конструкции:

– следование – обозначает последовательное выполнение действий (рис. 2.2, *а*);

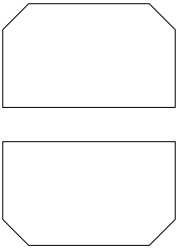
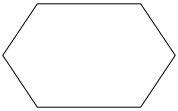
2.3. Символы ГОСТа 19.701–90

Название блока	Обозначение	Назначение блока
Терминатор		Выход во внешнюю среду и вход из внешней среды (начало или конец схемы программы, внешнее использование и источник или пункт назначения данных)
Данные		Данные, носитель данных не определён
Запоминаемые данные		Хранимые данные в виде, пригодном для обработки, носитель данных не определён
Оперативное запоминающее устройство		Данные, хранящиеся в оперативном запоминающем устройстве
Запоминающее устройство с последовательным доступом		Данные, хранящиеся в запоминающем устройстве с последовательным доступом (магнитная лента, кассета с магнитной лентой, магнитофонная кассета)
Запоминающее устройство с прямым доступом		Данные, хранящиеся в запоминающем устройстве с прямым доступом (магнитный диск, магнитный барабан, гибкий магнитный диск)
Документ		Данные, представленные на носителе в удобочитаемой форме
Ручной ввод		Данные, вводимые вручную во время обработки с устройств любого типа
Карта		Данные, представленные на носителе в виде карты (перфокарты, магнитные карты, карты со считываемыми метками и т.п.)

Продолжение табл. 2.3

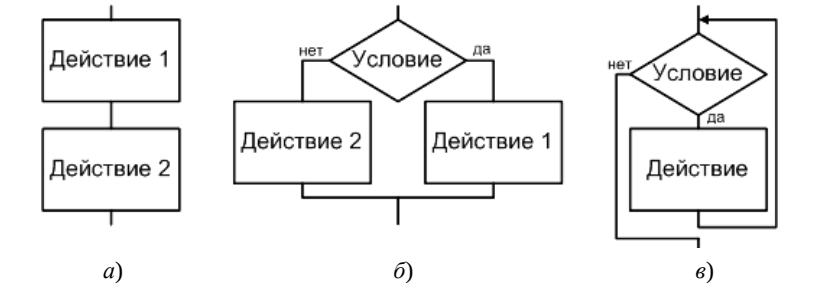
Название блока	Обозначение	Назначение блока
Бумажная лента		Данные, представленные на носителе в виде бумажной ленты
Дисплей		Данные, представленные в человекочитаемой форме на носителе в виде отображающего устройства (экран для визуального наблюдения, индикаторы ввода информации)
Процесс		Функция обработки данных любого вида (выполнение определённой операции или группы операций, приводящее к изменению значения, формы или размещения информации или к определению, по которому из нескольких направлений потока следует двигаться)
Предопределённый процесс		Предопределённый процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте (в подпрограмме, модуле)
Ручная операция		Любой процесс, выполняемый человеком
Решение		Решение или функция переключательного типа, имеющая один вход и ряд альтернативных выходов, один и только один из которых может быть активизирован после вычисления условий

Продолжение табл. 2.3

Название блока	Обозначение	Назначение блока
Граница цикла		Начало и конец цикла. Обе части символа имеют один и тот же идентификатор. Условия для инициализации, приращения, завершения и т.д. помещаются внутри символа в начале или в конце в зависимости от расположения операции, проверяющей условие
Подготовка		Модификация команды или группы команд с целью воздействия на некоторую последующую функцию (установка переключателя, модификация индексного регистра или инициализация программы)
Соединитель		Выход в часть схемы и вход из другой части этой схемы используются для обрыва линии и продолжения её в другом месте. Соответствующие символы-соединители должны содержать одно и то же уникальное обозначение
Комментарий		Добавление описательных комментариев или пояснительных записей в целях объяснения или примечаний
Параллельные действия		Синхронизация двух или более параллельных операций

– ветвление – соответствует выбору одного из двух вариантов действий (рис. 2.2, *б*);

– цикл-пока – определяет повторение действий, пока не будет нарушено некоторое условие, выполнение которого проверяется в начале цикла (рис. 2.2, *в*).



а) *б*) *в*)

Рис. 2.2. Базовые алгоритмические структуры:
а – следование; *б* – ветвление; *в* – цикл-пока

Помимо базовых, процедурные языки программирования высокого уровня обычно используют ещё три конструкции, которые можно составить из базовых:

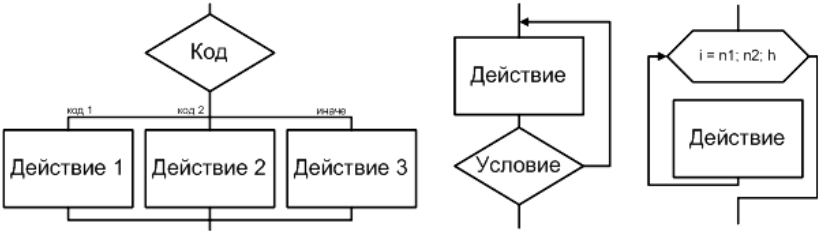
– выбор – обозначает выбор одного варианта из нескольких в зависимости от значения некоторой величины (рис. 2.3, *а*);

– цикл-до – обозначает повторение некоторых действий до выполнения заданного условия, проверка которого осуществляется после выполнения действий в цикле (рис. 2.3, *б*);

– цикл с заданным числом повторений (счётный цикл) – обозначает повторение некоторых действий указанное количество раз (рис. 2.3, *в*).

Любая из дополнительных конструкций легко реализуется через базовые. Перечисленные шесть конструкций были положены в основу структурного программирования.

Программы, написанные с использованием только структурных операторов передачи управления, называют структурными, чтобы подчеркнуть их отличие от программ, при проектировании или реализации которых использовались низкоуровневые способы передачи управления.



а) *б*) *в*)

Рис. 2.3. Дополнительные структуры алгоритмов:
а – выбор; *б* – цикл-до; *в* – цикл с заданным числом повторений

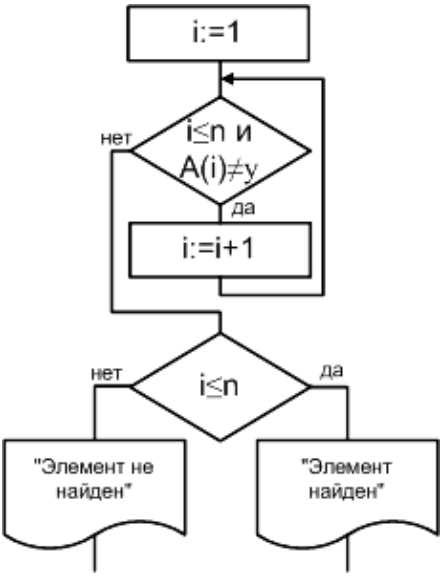


Рис. 2.4. Схема алгоритма поиска максимального элемента в массиве

Для примера представлена схема алгоритма поиска максимального элемента массива (см. рис. 2.4).

Кроме схем, для описания алгоритмов можно использовать псевдокоды, Flow-формы и диаграммы Насси-Шнейдермана. Все перечисленные нотации, с одной стороны, базируются на тех же основных структурах, что и структурное программирование, а с другой – допускают разные уровни детализации.

Псевдокоды. Псевдокод – формализованное текстовое описание алгоритма (текстовая нотация). В литературе были предложены несколько вариантов псевдокодов [2]. Один из них приведён в табл. 2.4.

Описать с помощью псевдокодов неструктурный алгоритм невозможно. Использование псевдокодов изначально ориентирует проектировщика только на структурные способы передачи управления, а потому требует более тщательного анализа разрабатываемого алгоритма. В отличие от схем алгоритмов, псевдокоды не ограничивают степень детализации проектируемых операций. Они позволяют соизмерять степень детализации действия с уровнем абстракции, на котором это действие рассматривают, и хорошо согласуются с основным методом структурного программирования – методом пошаговой детализации.

2.4. Соответствие алгоритмической структуры и псевдокода

Структура	Псевдокод
Следование	<действие 1> <действие 2> … <действие n>
Ветвление	**Если** <условие> **то** <действие 1> **иначе** <действие 2> **Все-если**
Цикл-пока	**Цикл-пока** <условие> <действие> **Все-цикл**
Выбор	**Выбор** <код> <код 1>: <действие 1> <код 2>: <действие 2> … **Все-выбор**
Цикл-до	**Выполнять** <действие> **До** <условие>
Цикл с заданным количеством повторений	**Для** <индекс>= <n>,<k>,<h> <действие> **Все-цикл**

В качестве примера посмотрим, как будет выглядеть на псевдокоде описание алгоритма поискового цикла, представленного на рис. 2.4:

```
i: = 1
Цикл-пока i ≥ < n и A[i] ≠ y
i: = i + l
Все-цикл
Если  i ≥ n
то Вывести «Элемент найден»
иначе Вывести «Элемент не найден»
Все-если
```

Flow-формы. Flow-формы представляют собой графическую нотацию описания структурных алгоритмов, которая иллюстрирует вложенность структур. Каждый символ Flow-формы соответствует управляющей структуре и изображается в виде прямоугольника. Для демонстрации вложенности структур символ Flow-формы может быть вписан в соответствующую область прямоугольника любого другого символа. В прямоугольниках символов содержится текст на естественном языке или в математической нотации. Размер прямоугольника определяется длиной вписанного в него текста и размерами вложенных прямоугольников. Символы Flow-форм приведены на рис. 2.5.

На рисунке 2.6 представлено описание рассмотренного ранее поискового цикла с использованием Flow-формы. Хорошо видны вложенность и следование конструкций, изображённых прямоугольниками.

Диаграммы Насси–Шнейдермана. Диаграммы Насси–Шнейдермана являются развитием Flow-форм. Основное их отличие от Flow-форм заключается в том, что область обозначения условий и вариантов ветвления изображают в виде треугольников (рис. 2.7). Такое обозначение обеспечивает большую наглядность представления алгоритма.

Как и при использовании псевдокодов, описать неструктурный алгоритм, применяя Flow-формы или диаграммы Насси–Шнейдермана, невозможно (для неструктурных передач управления в этих нотациях просто отсутствуют условные обозначения). В то же время, являясь

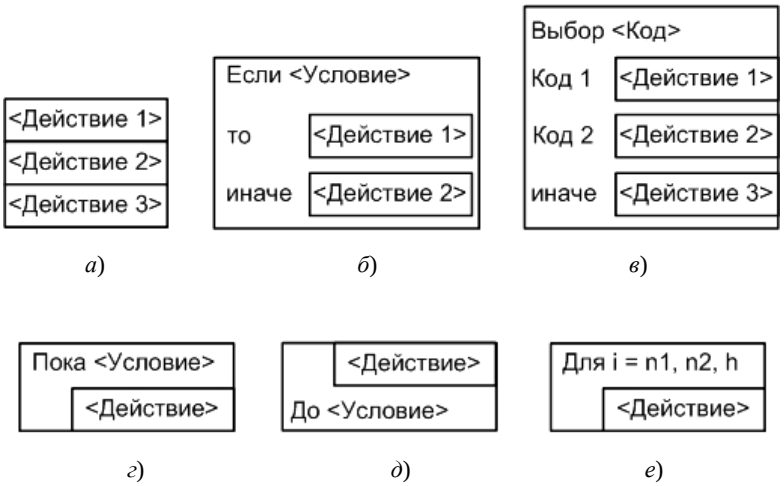


Рис. 2.5. Условные обозначения Flow-форм для основных конструкций:
а – следование; *б* – ветвление; *в* – выбор; *г* – цикл-до; *е* – счётный цикл

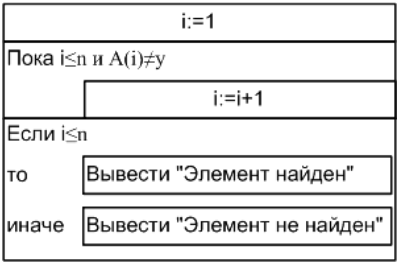
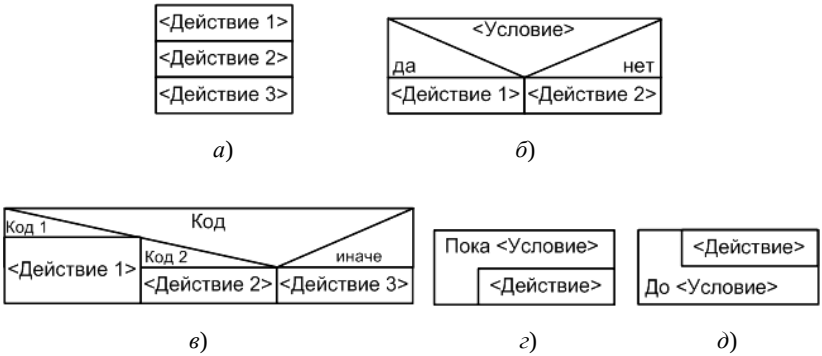


Рис. 2.6. Алгоритм поискового цикла



а) *б*) *в*) *г*) *д*)

Рис. 2.7. Условные обозначения диаграмм Насси–Шнейдермана для основных конструкций:
а – следование; *б* – ветвление; *в* – выбор; *г* – цикл-пока; *д* – цикл-до

графическими, эти нотации лучше отображают вложенность конструкций, чем псевдокоды.

Общим недостатком Flow-форм и диаграмм Насси–Шнейдермана является сложность построения изображений символов, что усложняет практическое применение этих нотаций для описания больших алгоритмов.

2.5. ПРОГРАММИРОВАНИЕ «С ЗАЩИТОЙ ОТ ОШИБОК»

Любая из ошибок программирования, которая не обнаруживается на этапах компиляции и компоновки программы, в конечном счёте может проявиться тремя способами: привести к выдаче системного сообщения об ошибке, «зависанию» компьютера и получению неверных результатов.

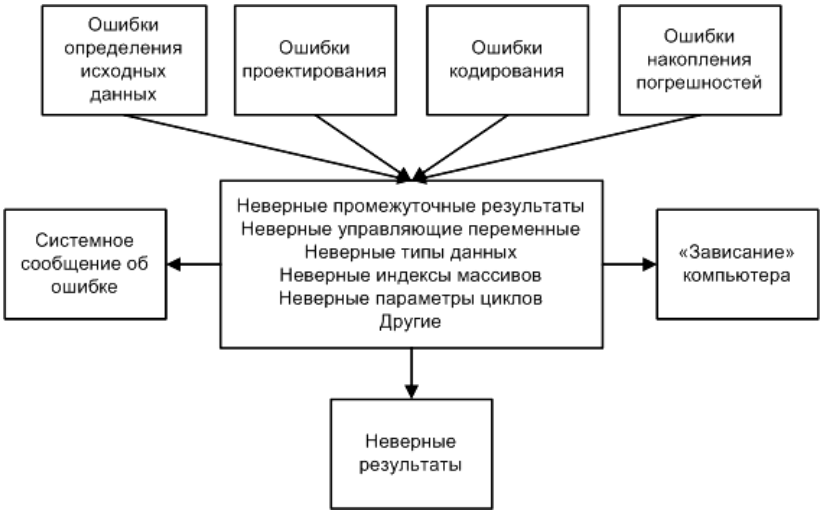


Рис. 2.8. Способы проявления ошибок

Однако до того, как результат работы программы становится фатальным, ошибки обычно много раз проявляются в виде неверных промежуточных результатов, неверных управляющих переменных, неверных типах данных, индексах структур данных и т.п. (рис. 2.8). Это значит, что часть ошибок можно попытаться обнаружить и нейтрализовать, пока они ещё не привели к тяжёлым последствиям.

Программирование, при котором применяют специальные приёмы раннего обнаружения и нейтрализации ошибок, было названо защитным или программированием с защитой от ошибок. При его использовании существенно уменьшается вероятность получения неверных результатов.

Детальный анализ ошибок и их возможных ранних проявлений показывает, что целесообразно проверять:

– правильность выполнения операций ввода-вывода;

– допустимость промежуточных результатов (значений управляющих переменных, значений индексов, типов данных, значений числовых аргументов и т.д.).

Проверки правильности выполнения операций ввода-вывода. Причинами неверного определения исходных данных могут являться, как внутренние ошибки – ошибки устройств ввода-вывода или программного обеспечения, так и внешние ошибки – ошибки пользователя. При этом принято различать:

– ошибки передачи – аппаратные средства, например вследствие неисправности, искажают данные;

– ошибки преобразования – программа неверно преобразует исходные данные из входного формата во внутренний;

– ошибки перезаписи – пользователь ошибается при вводе данных, например, вводит лишний или другой символ;

– ошибки данных – пользователь вводит неверные данные. Ошибки передачи обычно контролируются аппаратно.

Для защиты от ошибок преобразования данные после ввода обычно сразу демонстрируют пользователю («эхо»). При этом выполняют сначала преобразование во внутренний формат, а затем обратно. Однако предотвратить все ошибки преобразования на данном этапе крайне сложно, поэтому соответствующие фрагменты программы тщательно тестируют, используя методы эквивалентного разбиения и граничных значений.

Обнаружить и устранить ошибки перезаписи можно только, если пользователь вводит избыточные данные, например контрольные суммы. Если ввод избыточных данных по каким-либо причинам нежелателен, то следует по возможности проверять вводимые данные, хотя бы контролировать интервалы возможных значений, которые обычно определены в техническом задании, и выводить введённые данные для проверки пользователю. Неверные данные обычно может обнаружить только пользователь.

Проверка допустимости промежуточных результатов. Проверки промежуточных результатов позволяют снизить вероятность позднего проявления не только ошибок неверного определения данных, но и некоторых ошибок кодирования и проектирования. Для того чтобы такая проверка была возможной, необходимо в программе использовать переменные, для которых существуют ограничения любого происхождения, например связанные с сущностью моделируемых процессов.

Однако следует иметь в виду, что любые дополнительные операции в программе требуют использования дополнительных ресурсов (времени, памяти и т.п.) и могут также содержать ошибки. Поэтому имеет смысл проверять не все промежуточные результаты, а только те, проверка которых целесообразна, т.е. возможно позволит обнаружить ошибку, и не сложна. Например:

– если каким-либо образом вычисляется индекс элемента массива, то следует проверить, что этот индекс является допустимым;

– если строится цикл, количество повторений которого определяется значением переменной, то целесообразно убедиться, что значение этой переменной не отрицательно;

– если определяется вероятность какого-либо события, то целесообразно проверить, что полученное значение не более 1, а сумма вероятностей всех возможных независимых событий равна 1 и т.д.

Предотвращение накопления погрешностей. Чтобы снизить погрешности результатов вычислений, необходимо соблюдать следующие рекомендации:

– избегать вычитания близких чисел (машинный ноль);
– избегать деления больших чисел на малые;
– сложение длинной последовательности чисел начинать с меньших по абсолютной величине;
– стремиться по возможности уменьшать количество операций;
– использовать методы с известными оценками погрешностей;
– не использовать условие равенства вещественных чисел;
– вычисления производить с двойной точностью, а результат выдавать с одинарной.

Обработка исключений. Поскольку полный контроль данных на входе и в процессе вычислений, как правило, невозможен, следует предусматривать перехват обработки аварийных ситуаций.

Для перехвата и обработки аппаратно и программно фиксируемых ошибок в некоторых языках программирования, например Delphi Pascal, C++, Java, предусмотрены средства обработки исключений. Использование эти средств позволяет не допустить выдачи пользователю сообщения об аварийном завершении программы, ничего ему не говорящего. Вместо этого программист получает возможность предусмотреть действия, которые позволяют исправить эту ошибку или, если это невозможно, выдать пользователю сообщение с точным описанием ситуации и продолжить работу.

2.6. СКВОЗНОЙ СТРУКТУРНЫЙ КОНТРОЛЬ

Сквозной структурный контроль представляет собой совокупность технологических операций контроля, позволяющих обеспечить как можно более раннее обнаружение ошибок в процессе разработки. Термин «сквозной» в названии отражает выполнение контроля на всех этапах разработки. Термин «структурный» означает наличие чётких рекомендаций по выполнению контролирующих операций на каждом этапе. Сквозной структурный контроль должен выполняться на специальных контрольных сессиях, в которых, помимо разработчиков, могут участвовать специально приглашённые эксперты. Время между сессиями определяет объём материала, который выносится на сессию: при частых сессиях материал рассматривают небольшими порциями, при редких – существенными фрагментами.

Материалы для очередной сессии должны выдаваться участникам заранее, чтобы они могли их обдумать.

Одна из первых сессий должна быть организована на этапе определения спецификаций. На этой сессии проверяют полноту и точность спецификаций, при этом целесообразно присутствие заказчика или специалиста по предметной области, которые смогут определить, насколько правильно и полно составлены спецификации программного обеспечения.

На этапе проектирования вручную по частям проверяют алгоритмы разрабатываемого программного обеспечения на конкретных наборах данных и сверяют полученные результаты с соответствующими спецификациями. Основная задача – убедиться в правильности понимания спецификаций и проанализировать достоинства и недостатки концептуальных решений, закладываемых в проект.

На этапе реализации проверяют план (последовательность) реализации модулей, набор тестов, а также тексты отдельных модулей.

Для всех этапов целесообразно иметь списки наиболее часто встречающихся ошибок, которые формируют по литературным источникам и исходя из опыта предыдущих разработок. Такие списки позволяют сконцентрировать усилия на конкретных моментах, а не проверять всё подряд. При этом все найденные ошибки фиксируют в специальном документе, но не исправляют их.

Помимо раннего обнаружения ошибок, сквозной структурный контроль обеспечивает своевременную подготовку качественной документации по проекту.

3. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

3.1. ОСНОВНЫЕ ПОНЯТИЯ ЯЗЫКА С++

В тексте на любом естественном языке можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Подобные элементы содержит и алгоритмический язык, только слова называют лексемами (элементарными конструкциями), словосочетания – выражениями, а предложения – операторами. Лексемы образуются из символов, выражения – из лексем и символов, а операторы – из символов, выражений и лексем (рис. 3.1):

- Алфавит языка, или его символы – это основные неделимые знаки, с помощью которых пишутся все тексты на языке.
- Лексема, или элементарная конструкция – минимальная единица языка, имеющая самостоятельный смысл.
- Выражение задаёт правило вычисления некоторого значения.
- Оператор задаёт законченное описание некоторого действия.

Для описания сложного действия требуется последовательность операторов. Операторы могут быть объединены в составной оператор, или блок. В этом случае они рассматриваются как один оператор.

Операторы бывают исполняемые и неисполняемые. Исполняемые операторы задают действия над данными. Неисполняемые операторы служат для описания данных, поэтому их часто называют операторами описания или просто описаниями.

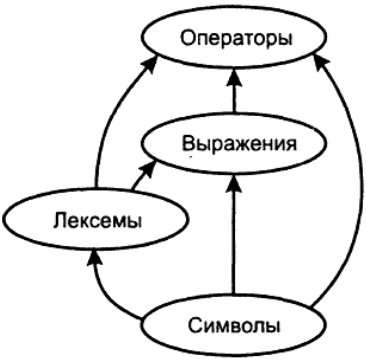


Рис. 3.1. Состав алгоритмического языка

Каждый элемент языка определяется синтаксисом и семантикой. Синтаксические определения устанавливают правила построения элементов языка, а семантика определяет их смысл и правила использования.

3.1.1. АЛФАВИТ ЯЗЫКА

Алфавит C++ включает:

– прописные и строчные латинские буквы и знак подчёркивания;
– арабские цифры от 0 до 9;
– специальные знаки: { } , | [ ] ( ) + – / % * . \ ? < = > ! & # – ;
– пробельные символы: пробел, символы табуляции, символы перехода на новую строку.

Из символов алфавита формируются лексемы языка:

– идентификаторы;
– ключевые (зарезервированные) слова;
– знаки операций;
– константы;
– разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими как разделители или знаки операций.

3.1.2. ИДЕНТИФИКАТОРЫ

Идентификатор – это имя программного объекта. В идентификаторе могут использоваться латинские буквы, цифры и знак подчёркивания. Прописные и строчные буквы различаются, например, sysop, SySoP и SYSOP – три различных имени. Первым символом идентификатора может быть буква или знак подчёркивания, но не цифра. Пробелы внутри имён не допускаются.

Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на неё ограничения. Идентификатор создаётся на этапе объявления переменной, функции, типа и т.п., после этого его можно использовать в последующих операторах программы. При выборе идентификатора необходимо иметь в виду следующее:

– идентификатор не должен совпадать с ключевыми словами и именами используемых стандартных объектов языка;
– не рекомендуется начинать идентификаторы с символа подчёркивания, поскольку они могут совпасть с именами системных функций или переменных, кроме того, это снижает мобильность программы;

– на идентификаторы, используемые для определения внешних переменных, налагаются ограничения компоновщика (использование различных компоновщиков или версий компоновщика накладывает разные требования на имена внешних переменных).

3.1.3. КЛЮЧЕВЫЕ СЛОВА

Ключевые слова – это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов C++ приведён в табл. 3.1.

3.1.4. ЗНАКИ ОПЕРАЦИЙ

Знак операции – это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в них операндов. Один и тот же знак может интерпретироваться по-разному в зависимости от контекста. Все знаки операций за исключением [ ], ( ) и ? : представляют собой отдельные лексемы.

3.1. Список ключевых слов в C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	expert	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

3.1.5. КОНСТАНТЫ

Константами называют неизменяемые величины. Различаются целые, вещественные, символьные и строковые константы. Компилятор, выделив константу в качестве лексемы, относит её к одному из типов по её внешнему виду.

Форматы констант, соответствующие каждому типу, приведены в табл. 3.2.

Если требуется сформировать отрицательную целую или вещественную константу, то перед константой ставится знак унарной операции изменения знака (–), например: –218, –022, –0x3C, –4.8, –0.1e4.

Вещественная константа в экспоненциальном формате представляется в виде мантиссы и порядка. Мантисса записывается слева от знака экспоненты (E или e), порядок – справа от знака. Значение константы определяется как произведение мантиссы и возведённого в указанную в порядке степень числа 10. Обратите внимание, что пробелы внутри числа не допускаются, а для отделения целой части от дробной используется не запятая, а точка.

Таблица 3.2

Константа	Формат	Примеры
Целая	Десятичный: последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль	8, 0, 199226
	Восьмеричный: нуль, за которым следуют восьмеричные цифры (0, 1, 2, 3, 4, 5, 6, 7)	01, 020, 07155
	Шестнадцатеричный: 0x или 0X, за которым следуют шестнадцатеричные (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)	0Xa, 0x1B8, 0X00FF
Вещественная	Десятичный: [цифры]. [цифры][2]	5.7, .001, 35.
	Экспоненциальный: [цифры] [.] [цифры] {E|e} [+|–] [цифры][3]	0.2E6, .11e–3, 5E10
Символьная	Один или два символа, заключённых в апострофы	'A', 'ю', '*', 'db', '\0', '\n', '\012', '\x07\x07'
Строковая	Последовательность символов, заключённая в кавычки	"Здесь был Vasta", "\tЗначение r=\0xFS\n"

Символьные константы, состоящие из одного символа, занимают в памяти один байт и имеют стандартный тип char. Двухсимвольные константы занимают два байта и имеют тип int, при этом первый символ размещается в байте с меньшим адресом.

Символ обратной косой черты используется для представления:

– кодов, не имеющих графического изображения (например, \a – звуковой сигнал, \n – перевод курсора в начало следующей строки);

– символов апострофа ('), обратной косой черты (\ ), знака вопроса (?) и кавычки (");

– любого символа с помощью его шестнадцатеричного или восьмеричного кода, например, \073, \0xF5. Числовое значение должно находиться в диапазоне от 0 до 255.

Последовательности символов, начинающиеся с обратной косой черты, называют управляющими, или escape-последовательностями. Управляющая последовательность интерпретируется как одиночный символ. Управляющие последовательности могут использоваться и в строковых константах, называемых иначе строковыми литералами. Например, если внутри строки требуется записать кавычку, её предваряют косой чертой, по которой компилятор отличает её от кавычки, ограничивающей строку.

```
"ФГБОУ ВПО \"Тамбовский государственный технический университет\""
```

Все строковые литералы рассматриваются компилятором как различные объекты. Строковые константы, отделённые в программе только пробельными символами, при компиляции объединяются в одну. Длинную строковую константу можно разместить на нескольких строках, используя в качестве знака переноса обратную косую черту, за которой следует перевод строки. Эти символы игнорируются компилятором, при этом следующая строка воспринимается как продолжение предыдущей.

В конец каждого строкового литерала компилятором добавляется нулевой символ, представляемый управляющей последовательностью \0. Поэтому длина строки всегда на единицу больше количества символов в её записи. Таким образом, пустая строка "" имеет длину 1 байт. Обратите внимание на разницу между строкой из одного символа, например "A", и символьной константой 'A'.

Пустая символьная константа недопустима.

3.1.6. КОММЕНТАРИИ

Комментарий либо начинается с двух символов «прямая косая черта» (//) и заканчивается символом перехода на новую строку, либо заключается между символами-скобками /* и */. Внутри комментария можно использовать любые допустимые на данном компьютере символы, а не только символы из алфавита языка С++, поскольку компилятор комментарии игнорирует. Вложенные комментарии-скобки стандартом не допускаются, хотя в некоторых компиляторах разрешены.

3.1.7. ТИПЫ ДАННЫХ С++

Основная цель любой программы состоит в обработке данных. Данные различного типа хранятся и обрабатываются по-разному. В любом алгоритмическом языке каждая константа, переменная, результат вычисления выражения или функции должны иметь определённый тип.

Тип данных определяет:

– внутреннее представление данных в памяти компьютера;

– множество значений, которые могут принимать величины этого типа;

– операции и функции, которые можно применять к величинам этого тина.

Исходя из этих характеристик, программист выбирает тип каждой величины, используемой в программе для представления реальных объектов. Обязательное описание типа позволяет компилятору производить проверку допустимости различных конструкций программы. От типа величины зависят машинные команды, которые будут использоваться для обработки данных.

Все типы языка С++ можно разделить на основные и составные [3 – 13]. В языке С++ определено шесть основных типов данных для представления целых, вещественных, символьных и логических величин. На основе этих типов программист может вводить описание составных типов. К ним относятся массивы, перечисления, функции, структуры, ссылки, указатели, объединения и классы.

Основные (стандартные) типы данных часто называют арифметическими, поскольку их можно использовать в арифметических операциях. Для описания основных типов определены следующие ключевые слова [3 – 13]:

- int (целый);
- char (символьный);
- wchar_t (расширенный символьный);

- bool (логический);
- float (вещественный);
- double (вещественный с двойной точностью).

Первые четыре типа называют целочисленными (целыми), последние два – типами с плавающей точкой. Код, который формирует компилятор для обработки целых величин, отличается от кода для величин с плавающей точкой.

Существует четыре спецификатора типа, уточняющих внутреннее представление и диапазон значений стандартных типов:

- short (короткий);
- long (длинный);
- signed (знаковый);
- unsigned (беззнаковый).

Целый тип (int). Размер типа int не определяется стандартом, а зависит от компьютера и компилятора. Для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного – 4 байта.

Спецификатор short перед именем типа указывает компилятору, что под число требуется отвести 2 байта независимо от разрядности процессора. Спецификатор long означает, что целая величина будет занимать 4 байта. Таким образом, на 16-разрядном компьютере эквиваленты int и short int, а на 32-разрядном – int и long int.

Внутреннее представление величины целого типа – целое число в двоичном коде. При использовании спецификатора signed старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Спецификатор unsigned позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. Таким образом, диапазон значений типа int зависит от спецификаторов. Диапазоны значений величин целого типа с различными спецификаторами для IBM PC-совместимых компьютеров приведены в табл. 3.3.

По умолчанию все целочисленные типы считаются знаковыми, т.е. спецификатор signed можно опускать.

Константам, встречающимся в программе, приписывается тот или иной тип в соответствии с их видом. Если этот тип по каким-либо причинам не устраивает программиста, он может явно указать требуемый тип с помощью суффиксов L, l (long) и u, U (unsigned). Например, константа 32L будет иметь тип long и занимать 4 байта. Можно использовать суффиксы L и U одновременно, например 0x22UL или 05LU.

Типы short int, long int, signed int и unsigned int можно сокращать до short, long, signed и unsigned соответственно.

3.3. Диапазоны значений простых типов данных

Тип	Диапазон значений	Размер (байт)
bool	true и false	1
signed char	–128…127	1
unsigned char	0…255	1
signed short int	–32 768…32 767	2
unsigned short int	0…65 535	2
signed long int	–2 147 483 648…2 147 483 647	4
unsigned long int	0…4 294 967 295	4
float	3.4e–38…3.4e+38	4
double	1.7e–308…1.7e+308	8
long double	3.4e–4932…3.4e+4932	10

Символьный тип (char). Под величину символьного типа отводится количество байт, достаточное для размещения любого символа из набора символов для данного компьютера, что и обусловило название типа. Как правило, это 1 байт. Тип char, как и другие целые типы, может быть со знаком или без знака. В величинах со знаком можно хранить значения в диапазоне от –128 до 127. При использовании спецификатора unsigned значения могут находиться в пределах от 0 до 255. Этого достаточно для хранения любого символа из 256-символьного набора ASCII. Величины типа char применяются также для хранения целых чисел, не превышающих границы указанных диапазонов.

Расширенный символьный тип (wchar_t). Тип wchar_t предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например Unicode. Размер этого типа зависит от реализации; как правило, он соответствует типу short. Строковые константы типа wchar_t записываются с префиксом L, например, L"Здравствуйте".

Логический тип (bool). Величины логического типа могут принимать только значения true и false, являющиеся зарезервированными словами. Внутренняя форма представления значения false – 0 (нуль). Любое другое значение интерпретируется как true. При преобразовании к целому типу true имеет значение 1.

Типы с плавающей точкой (float, double и long double). Стандарт C++ определяет три типа данных для хранения вещественных значений: float, double и long double.

Типы данных с плавающей точкой хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей – мантиссы и порядка. В IBM PC-совместимых компьютерах величины типа float занимают 4 байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Мантисса – это число, большее 1.0, но меньшее 2.0. Поскольку старшая цифра мантиссы всегда равна 1, она не хранится.

Для величин типа double, занимающих 8 байт, под порядок и мантиссу отводятся 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка – его диапазон. Как можно видеть из табл. 3.3, при одинаковом количестве байт, отводимом под величины типа float и long int, диапазоны их допустимых значений сильно различаются из-за внутренней формы представления.

Спецификатор long перед именем типа double указывает, что под величину отводится 10 байт.

Константы с плавающей точкой имеют по умолчанию тип double. Можно явно указать тип константы с помощью суффиксов F, f (float) и L, l (long). Например, константа 2E+6L будет иметь тип long double, а константа 1.82f – тип float.

Для вещественных типов в таблице приведены абсолютные величины минимальных и максимальных значений.

Тип void. Кроме перечисленных, к основным типам языка относится тип void, но множество значений этого типа пусто. Он используется для определения функций, которые не возвращают значения, для указания пустого списка аргументов функции, как базовый тип для указателей и в операции приведения типов.

3.1.8. ПЕРЕМЕННЫЕ И ВЫРАЖЕНИЯ

В любой программе требуется производить вычисления. Для вычисления значений используются выражения, которые состоят из операндов, знаков операций и скобок. Операнды задают данные для вычислений. Операции задают действия, которые необходимо выполнить. Каждый операнд является, в свою очередь, выражением или одним из его частных случаев, например константой или переменной.

Операции выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки.

Рассмотрим составные части выражений и правила их вычисления.

Переменная – это именованная область памяти, в которой хранятся данные определённого типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится зна-

чение. Во время выполнения программы значение переменной можно изменять. Перед использованием любая переменная должна быть описана.

Пример описания целой переменной с именем a и вещественной переменной x:

```
int a;
float x;
```

Общий вид оператора описания переменных:

```
[класс памяти] [const] тип имя [инициализатор];
```

Рассмотрим правила задания составных частей этого оператора.

1. Необязательный класс памяти может принимать одно из значений auto, extern, static и register.

2. Модификатор const показывает, что значение переменной изменять нельзя. Такую переменную называют именованной константой, или просто константой.

3. При описании можно присвоить переменной начальное значение, это называется инициализацией. Инициализатор можно записывать в двух формах – со знаком равенства:

```
= значение
```

или в круглых скобках:

```
( значение )
```

Константа должна быть инициализирована при объявлении. В одном операторе можно описать несколько переменных одного типа, разделяя их запятыми. Примеры:

```
short int a = 1; // целая переменная a
const char C = 'C' ; // символьная константа C
char s, sf = 'f '; // инициализация относится только к sf
char t(54);
float c = 0.22, x(3), sum;
```

Если тип инициализирующего значения не совпадает с типом переменной, выполняются преобразования типа по определённым правилам.

Описание переменной, кроме типа и класса памяти, явно или по умолчанию задаёт её область действия. Класс памяти и область дейст-

вия зависят не только от собственно описания, но и от места его размещения в тексте программы.

Область действия идентификатора – это часть программы, в которой его можно использовать для доступа к связанной с ним области памяти. В зависимости от области действия переменная может быть локальной или глобальной.

Если переменная определена внутри блока, а блок ограничен фигурными скобками, она называется локальной, область её действия – от точки описания до конца блока, включая все вложенные блоки. Если переменная определена вне любого блока, она называется глобальной и областью её действия считается файл, в котором она определена, от точки описания до его конца.

Класс памяти определяет время жизни и область видимости программного объекта (в частности, переменной). Если класс памяти не указан явным образом, он определяется компилятором исходя из контекста объявления.

Время жизни может быть постоянным (в течение выполнения программы) и временным (в течение выполнения блока).

Областью видимости идентификатора называется часть текста программы, из которой допустим обычный доступ к связанной с идентификатором области памяти.

Чаще всего область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в её область действия. Тем не менее к этой переменной, если она глобальная, можно обратиться, используя операцию доступа к области видимости ::.

Для задания класса памяти используются следующие спецификаторы:

auto – автоматическая переменная. Память под неё выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, содержащего её определение. Освобождение памяти происходит при выходе из блока, в котором описана переменная. Время её жизни – с момента описания до конца блока. Для глобальных переменных этот спецификатор не используется, а для локальных он принимается по умолчанию, поэтому задавать его явным образом большого смысла не имеет;

extern – означает, что переменная определяется в другом месте программы (в другом файле или дальше по тексту);

static – статическая переменная. Время жизни – постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной;

register – аналогично auto, но память выделяется по возможности в регистрах процессора. Если такой возможности у компилятора нет, переменные обрабатываются как auto.

```
int a; //1 глобальная переменная a
int main()
{
int b; // 2 локальная переменная b
extern int X; //3 переменная x определена в другом месте
static int c; // 4 локальная статическая переменная c
a = 1; //5 присваивание глобальной переменной
int a; // 6 локальная переменная a
a = 2; //7 присваивание локальной переменной
::a = 3; //8 присваивание глобальной переменной
return 0;
}
int X = 4; // 9 определение и инициализация X
```

В этом примере глобальная переменная a определена вне всех блоков. Память под нее выделяется в сегменте данных в начале работы программы, областью действия является вся программа. Область видимости – вся программа, кроме строк 6 – 8, так как в первой из них определяется локальная переменная с тем же именем, область действия которой начинается с точки её описания и заканчивается при выходе из блока. Переменные b и c – локальные, область их видимости – блок, но время жизни различно: память под b выделяется в стеке при входе в блок и освобождается при выходе из него, а переменная c располагается в сегменте данных и существует все время, пока работает программа.

Если при определении начальное значение переменных явным образом не задаётся, компилятор присваивает глобальным и статическим переменным нулевое значение соответствующего типа. Автоматические переменные не инициализируются.

Имя переменной должно быть уникальным в своей области действия (например, в одном блоке не может быть двух переменных с одинаковыми именами).

Описание переменной может выполняться в форме объявления или определения.

Объявление информирует компилятор о типе переменной и классе памяти, а определение содержит, кроме этого, указание компилятору выделить память в соответствии с типом переменной. В C++ боль-

шинство объявлений являются одновременно и определениями. В приведённом выше примере только описание 3 является объявлением, но не определением.

Переменная может быть объявлена многократно, но определена только в одном месте программы, поскольку объявление просто описывает свойства переменной, а определение связывает её с конкретной областью памяти.

3.1.9. ОПЕРАЦИИ И ВЫРАЖЕНИЯ

В таблице 3.4 приведён список основных операций, определённых в языке C++, в соответствии с их приоритетами (по убыванию приоритетов, операции с разными приоритетами разделены чертой).

В соответствии с количеством используемых операндов операции делятся на унарные (один операнд), бинарные (два операнда) и тернарную (три операнда).

Все приведённые в таблице операции, кроме условной и sizeof, могут быть перегружены.

Таблица 3.4

Операция	Краткое описание
	Унарные операции
++	Увеличение на 1
--	Уменьшение на 1
sizeof	Размер
~	Поразрядное отрицание
!	Логическое отрицание
-	Арифметическое отрицание (унарный минус)
+	Унарный плюс
&	Взятие адреса
*	Разадресация
new	Выделение памяти
delete	Освобождение памяти
(type)	Преобразование типа

Продолжение табл. 3.4

Операция	Краткое описание
	Бинарные и тернарная операции
*	Умножение
/	Деление
%	Остаток от деления
+	Сложение
–	Вычитание
<<	Сдвиг влево
>>	Сдвиг вправо
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно
&	Поразрядная конъюнкция (И)
^	Поразрядное исключающее ИЛИ
|	Поразрядная дизъюнкция (ИЛИ)
&&	Логическое И
||	Логическое ИЛИ
?:	Условная операция (тернарная)
=	Присваивание
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Остаток от деления с присваиванием
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
<<=	Сдвиг влево с присваиванием
>>=	Сдвиг вправо с присваиванием
&=	Поразрядное И с присваиванием
|=	Поразрядное ИЛИ с присваиванием
^=	Поразрядное исключающее ИЛИ с присваиванием
,	Последовательное вычисление

Операции увеличения и уменьшения на 1 (++ и –). Эти операции, называемые также инкрементом и декрементом, имеют две формы записи – префиксную, когда операция записывается перед операндом, и постфиксную. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

Результат работы программы:
Значение префиксного выражения: 4
Значение постфиксного выражения: 3
Значение x после приращения: 4
Значение y после приращения: 4

Операндом операции инкремента в общем случае является так называемое L-значение (L-value). Так обозначается любое выражение, адресующее некоторый участок памяти, в который можно занести значение. Название произошло от операции присваивания, поскольку именно её левая (Left) часть определяет, в какую область памяти будет занесён результат операции.

Операция определения размера sizeof предназначена для вычисления размера объекта или типа в байтах и имеет две формы:

```
sizeof выражение
sizeof ( тип )
```

Пример:

```
#include <stdio.h>
int main()
{
float x = 1;
printf("sizeof (float) : %d", sizeof (float));
printf("\nsizeof (x) : %d", sizeof (x));
printf("\nsizeof (x + 1.0) : %d", sizeof (x +1.0));
return 0;
}
```

Результат работы программы:
sizeof(float) : 4
sizeof(x) : 4
sizeof(x + 1.0) : 8

Последний результат связан с тем, что вещественные константы по умолчанию имеют тип double, к которому, как к более длинному, приводится тип переменной x и всего выражения. Скобки необходимы для того, чтобы выражение, стоящее в них, вычислялось раньше операции приведения типа, имеющей больший приоритет, чем сложение.

Операции отрицания (-, **!** и ~). Арифметическое отрицание (унарный минус) изменяет знак операнда целого или вещественного типа на противоположный. Логическое отрицание (!) даёт в результате значение 0, если операнд есть истина (не нуль), и значение 1, если операнд равен нулю. Операнд должен быть целого или вещественного типа, а может иметь также тип указатель. Поразрядное отрицание (~), часто называемое побитовым, инвертирует каждый разряд в двоичном представлении целочисленного операнда.

Деление (/) и остаток от деления (%). Операция деления применима к операндам арифметического типа. Если оба операнда целочисленные, результат операции округляется до целого числа, в противном случае тип результата определяется правилами преобразования. Операция остатка от деления применяется только к целочисленным операндам. Знак результата зависит от реализации.

```
#include <stdio.h>
int main()
{
int x = 11, y = 4;
float z = 4;
printf("Результаты деления: %d %f\n", x/y, x/z);
printf("Остаток: %d\n", x%y);
return 0;
}
```

Результат работы программы:
Результаты деления: 2 2.750000
Остаток: 3.

Операции сдвига (<< и >>) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом. При сдвиге влево (<<) освободившиеся разряды обнуляются. При сдвиге вправо (>>) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа, и знаковым разрядом в противном случае. Операции сдвига не учитывают переполнение и потерю значимости.

Результат выполнения операций сдвига и поразрядных операций:

4 << 2 равняется 16;
5 >> 1 равняется 2.

Двоичный код для 4 равен 100, для 5 – это 101. При сдвиге влево на две позиции код 100 становится равным 10000 (десятичное значение равно 16), а при сдвиге вправо на одну позицию код 101 становится 10.

Операции отношения (<, <=, >, >=, == , !=) сравнивают первый операнд со вторым. Операнды могут быть арифметического типа или указателями. Результатом операции является значение true или false (любое значение, не равное нулю, интерпретируется как true). Операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

Поразрядные операции (&, |, ^) применяются только к целочисленным операндам и работают с их двоичными представлениями. При выполнении операций операнды сопоставляются побитово (первый бит первого операнда с первым битом второго, второй бит первого операнда со вторым битом второго и т д.).

При поразрядной конъюнкции, или поразрядном И (операция обозначается &) бит результата равен 1 только тогда, когда соответствующие биты обоих операндов равны 1.

При поразрядной дизъюнкции, или поразрядном ИЛИ (операция обозначается |) бит результата равен 1 тогда, когда соответствующий бит хотя бы одного из операндов равен 1.

При поразрядном исключающем ИЛИ (операция обозначается ^) бит результата равен 1 только тогда, когда соответствующий бит только одного из операндов равен 1.

```
#incluclе <stdio.h>
int main()
{
printf("\n 6 & 5 = %d", 6 & 5);
printf("\n 6 | 5 = %d", 6 | 5);
printf("\n 6 ^ 5 = %d", 6 ^ 5);
return 0;
}
```

Результат работы программы:
6 & 5 = 4
6 | 5 = 7
6 ^ 5 = 3

Логические операции (&& и ||). Операнды логических операций И (&&) и ИЛИ (||) могут иметь арифметический тип или быть указателями, при этом операнды в каждой операции могут быть различных типов. Преобразования типов не производятся, каждый операнд оценивается с точки зрения его эквивалентности нулю (операнд, равный нулю, рассматривается как false, не равный нулю – как true).

Результатом логической операции является true или false. Результат операции логическое И имеет значение true, только если оба операнда имеют значение true.

Результат операции логическое ИЛИ имеет значение true, если хотя бы один из операндов имеет значение true. Логические операции выполняются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

Операции присваивания (=, +=, -=, *= и т.д.). Операции присваивания могут использоваться в программе как законченные операторы.

Формат операции простого присваивания (=):

```
операнд_1 = операнд_2
```

Первый операнд должен быть L-значением, второй – выражением. Сначала вычисляется выражение, стоящее в правой части операции, а потом его результат записывается в область памяти, указанную в левой части (мнемоническое правило: «присваивание – это передача данных "налево"»). То, что ранее хранилось в этой области памяти, естественно, теряется.

При присваивании производится преобразование типа выражения к типу L-значения, что может привести к потере информации.

В сложных операциях присваивания (+=, *=, /= и т.п.) при вычислении выражения, стоящего в правой части, используется и L-значение из левой части. Например, при сложении с присваиванием ко второму операнду прибавляется первый, и результат записывается в первый операнд, т.е. выражение a + = b является более компактной записью выражения a = a + b.

Условная операция (?:). Эта операция тернарная, т.е. имеет три операнда. Её формат:

```
операнд_1 ? операнд_2 : операнд_3
```

Первый операнд может иметь арифметический тип или быть указателем. Он оценивается с точки зрения его эквивалентности нулю

(операнд, равный нулю, рассматривается как false, не равный пулю – как true). Если результат вычисления операнда 1 равен true, то результатом условной операции будет значение второго операнда, иначе – третьего операнда. Вычисляется всегда либо второй операнд, либо третий. Их тип может различаться. Условная операция является сокращенной формой условного оператора if.

```
#include <stdio.h>
int main()
{
int a = 11, b = 4, max;
max = (b > a)? b : a;
printf("Наибольшее число : %d", max);
return 0;
}
```

Результат работы программы:
Наибольшее число: 11

Другой пример применения условной операции. Требуется, чтобы некоторая целая величина увеличивалась на 1, если её значение не превышает n, а иначе принимала значение 1:

```
i = (i < n) ? i + 1 : 1;
```

Выражения состоят из операндов, знаков операций и скобок и используются для вычисления некоторого значения определённого типа. Каждый операнд является, в свою очередь, выражением или одним из его частных случаев – константой или переменной.

Примеры выражений:

```
(a + 0.12) / 6
x && y || !z
(t * sin(x) – 1.05e4) / ((2 * k + 2) * (2 * k + 3))
```

Операции выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, унарные операции, условная операция и операции присваивания выполняются справа налево, остальные – слева направо. Например, a = b = c означает a = (b = c), a a + b + c означает (a + b) + c. Порядок вычисления подвыражений внутри выражений не определён: например, нельзя считать, что в выражении (sin(x + 2) + cos(y + 1))

обращение к синусу будет выполнено раньше, чем к косинусу, и что x + 2 будет вычислено раньше, чем y + 1.

Результат вычисления выражения характеризуется значением и типом. Например, если a и b – переменные целого типа и описаны так:

```
int a = 2, b = 5;
```

то выражение a + b имеет значение 7 и тип int, а выражение a = b имеет значение, равное помещённому в переменную a (в данному случае 5), и тип, совпадающий с типом этой переменной. Таким образом, в C++ допустимы выражения вида a = b = c; сначала вычисляется выражение b = c, а затем его результат становится правым операндом для операции присваивания переменной a.

В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат операции будет иметь тот же тип. Если операнды разного типа, перед вычислениями выполняются преобразования типов по определённым правилам, обеспечивающим преобразование коротких типов в более длинные для сохранения значимости и точности.

Преобразования бывают двух типов:

– изменяющие внутреннее представление величин (с потерей точности или без потери точности);

– изменяющие только интерпретацию внутреннего представления.

К первому типу относится, например, преобразование целого числа в вещественное (без потери точности) и наоборот (возможно, с потерей точности), ко второму – преобразование знакового целого в беззнаковое.

В любом случае величины типов char, signed char, unsigned char, short int и unsigned short int преобразуются в тип int, если он может представить все значения, или в unsigned int в противном случае. После этого операнды преобразуются к типу наиболее длинного из них, и он используется как тип результата.

3.2. СТРУКТУРА И КОМПОНЕНТЫ ПРОГРАММЫ НА ЯЗЫКЕ C++

Каждая программа на языке C++ есть последовательность препроцессорных директив, описаний и определений глобальных объектов и функций. Препроцессорные директивы управляют преобразованием текста программы до её компиляции. Определения вводят функции и объекты. Объекты необходимы для представления в программе

обрабатываемых данных. Функции определяют потенциально возможные действия программы. Описания уведомляют компилятор о свойствах и именах тех объектов и функций, которые определены в других частях программы (например, ниже по её тексту или в другом файле).

Программа на языке С++ должна быть оформлена в виде одного или нескольких текстовых файлов. Текстовый файл разбит на строки. В конце каждой строки есть признак её окончания (плюс управляющий символ перехода к началу новой строки).

Для того чтобы выполнить программу, требуется перевести её на язык, понятный процессору – в машинные коды. Этот процесс состоит из нескольких этапов. Рисунок 3.2 иллюстрирует эти этапы для языка С++.

Сначала программа передаётся препроцессору, который выполняет директивы, содержащиеся в её тексте (например, включение в текст так называемых заголовочных файлов – текстовых файлов, в которых содержатся описания используемых в программе элементов).

Получившийся полный текст программы поступает на вход компилятора, который выделяет лексемы, а затем на основе грамматики языка распознает выражения и операторы, построенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и в случае их отсутствия строит объектный модуль.

Компоновщик, или редактор связей, формирует исполняемый модуль программы, подключая к объектному модулю другие объектные модули, в том числе содержащие функции библиотек, обращение к которым содержится в любой программе (например, для осуществления вывода на экран). Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. Исполняемый модуль имеет расширение .exe и запускается на выполнение обычным образом.

Определения и описания программы на языке С++ могут размещаться в строках текстового файла достаточно произвольно (в свободном формате.) Для препроцессорных директив существуют ограничения. Во-первых, препроцессорная директива обычно размещается в одной строке, т.е. признаком её окончания является признак конца строки текста программы. Во-вторых, символ '#', вводящий каждую директиву препроцессора, должен быть первым отличным от пробела символом в строке с препроцессорной директивой.

Задача препроцессора – преобразование текста программы до её компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора. Каждая препроцессорная директива начинается с символа '#'.

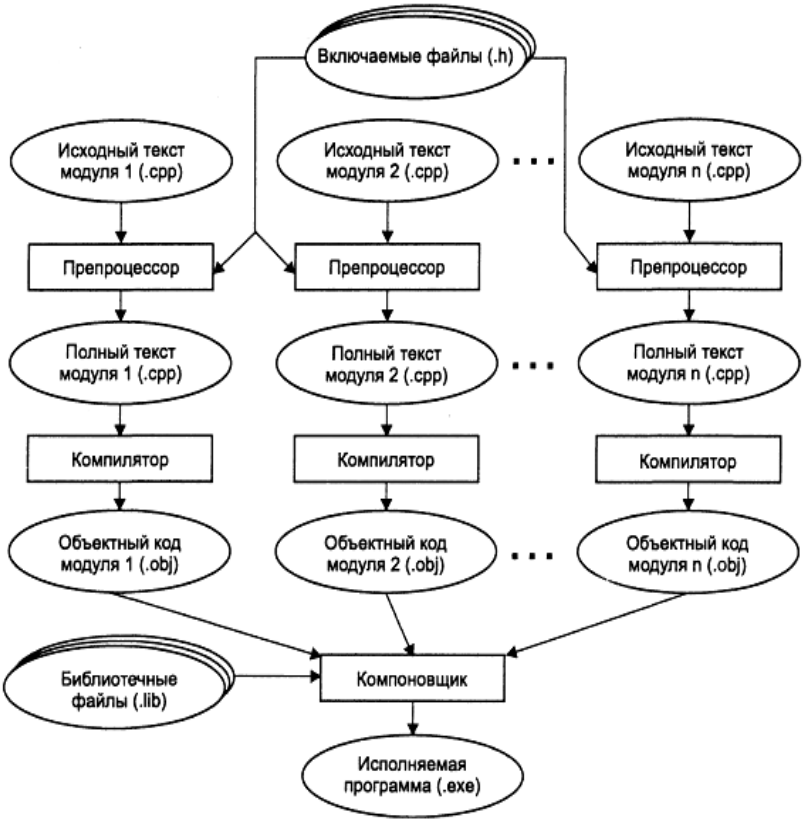


Рис. 3.2. Этапы создания исполняемой программы

Препроцессор «сканирует» исходный текст программы в поиске строк, начинающихся с символа '#'. Такие строки воспринимаются препроцессором как команды (директивы), которые определяют действия по преобразованию текста. Так директива #define указывает правила замены в тексте, а директива #include определяет, какие текстовые файлы нужно включить в этом месте текста программы.

Директива #include <...> предназначена для включения в текст программы текста файла из каталога «заголовочных файлов», поставляемых вместе со стандартными библиотеками компилятора. Каждая библиотечная функция, определённая стандартом языка, имеет соответствующее описание (прототип библиотечной функции плюс определения типов, переменных, макроопределений и констант) в одном из

заголовочных файлов. Список заголовочных файлов для стандартных библиотек определён стандартом языка.

Важно понимать, что употребление в программе препроцессорной директивы

#include < имя заголовочного файла >

не подключает к программе соответствующую стандартную библиотеку. Препроцессорная обработка выполняется на уровне исходного текста программы. Директива #include только позволяет вставить в текст программы описания из указанного заголовочного файла. Подключение к программе кодов библиотечных функций осуществляется только на этапе редактирования связей (этап компоновки), т.е. после компиляции, когда уже получен машинный код программы. Доступ к кодам библиотечных функций нужен только на этапе компоновки. Именно поэтому компилировать программу и устранять синтаксические ошибки в её тексте можно без стандартной библиотеки, но обязательно с заголовочными файлами.

Отметим, что хотя в заголовочных файлах содержатся описания всех стандартных функций, в код программы включаются только те функции, которые используются в программе. Выбор нужных функций выполняет компоновщик на этапе, называемом «редактирование связей».

Термин «заголовочный файл» (header file) в применении к файлам, содержащим описания библиотечных функций стандартных библиотек, не случаен. Он предполагает включение этих файлов именно в начало программы. Описание или определения функций должны быть «выше» по тексту, чем вызовы функций. Именно поэтому заголовочные файлы нужно помещать в начале текста программы, т.е. заведомо раньше обращений к соответствующим библиотечным функциям.

Хотя заголовочный файл может быть включён в программу не в её начале, а непосредственно перед обращением к нужной библиотечной функции, такое размещение директив #include <...> не рекомендуется.

Структура программы. После выполнения препроцессорной обработки в тексте программы не остаётся ни одной препроцессорной директивы. Теперь программа представляет собой набор описаний и определений. Если не рассматривать определений глобальных объектов и описаний, то программа будет набором определений функций.

Среди этих функций всегда должна присутствовать функция с фиксированным именем main. Именно эта функция является главной функцией программы, без которой программа не может быть выпол-

нена. Имя этой главной функции для всех программ одинаково (всегда main) и не может выбираться произвольно. Таким образом, исходный текст программы в простом случае (когда программа состоит только из одной функции) имеет такой вид:

```
директивы препроцессора
описания
void main()
{
определения объектов;
исполняемые_операторы;
}
```

Перед именем каждой функции программы следует помещать сведения о типе возвращаемого функцией значения (тип результата). Если функция ничего не возвращает, то указывается тип void. Функция main() является той функцией программы, которая запускается на исполнение по командам операционной системы. Возвращаемое функцией main() значение также передаётся операционной системе. Если не предполагается, что операционная система будет анализировать результат выполнения программы, то проще всего указать, что возвращаемое значение отсутствует, т.е. имеет тип void. Если сведения о типе результата отсутствуют, то считается по умолчанию, что функция main возвращает целочисленное значение типа int.

Каждая функция (в том числе и main) в языке C++ должна иметь набор параметров. Этот набор может быть пустым, тогда в скобках после имени функции помещается служебное слово void либо скобки остаются пустыми. В отличие от обычных функций, главная функция main() может использоваться как с параметрами, так и без них.

Вслед за заголовком void main() размещается тело функции. Тело функции – это блок, последовательность определений, описаний и исполняемых операторов, заключённая в фигурные скобки. Определения и описания в блоке будем размещать до исполняемых операторов. Каждое определение, описание и каждый оператор завершается символом ';' (точка с запятой).

Определения вводят объекты, необходимые для представления в программе обрабатываемых данных. Примером таких объектов служат именованные константы и переменные разных типов. Описания уведомляют компилятор о свойствах и именах объектов и функций, определённых в других частях программы. Операторы определяют действия программы на каждом шаге её выполнения.

3.3. БАЗОВЫЕ КОНСТРУКЦИИ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трёх структур, называемых следованием, ветвлением и циклом. Этот результат установлен Боймом и Якопини ещё в 1966 г. путём доказательства того, что любую программу можно преобразовать в эквивалентную, состоящую только из этих структур и их комбинаций [2, 3].

Рассмотрим операторы языка, реализующие базовые конструкции структурного программирования.

3.3.1. ОПЕРАТОР «ВЫРАЖЕНИЕ»

Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении выражения. Частным случаем выражения является пустой оператор ; (он используется, когда по синтаксису оператор требуется, а по смыслу – нет).

Примеры:

```
i++; // выполняется операция инкремента
a* = b + c; // выполняется умножение с присваиванием
fund(к); // выполняется вызов функции
```

3.3.2. ОПЕРАТОРЫ ВЕТВЛЕНИЯ

Условный оператор if используется для разветвления процесса вычислений на два направления. Структурная схема оператора приведена на рис. 3.3.

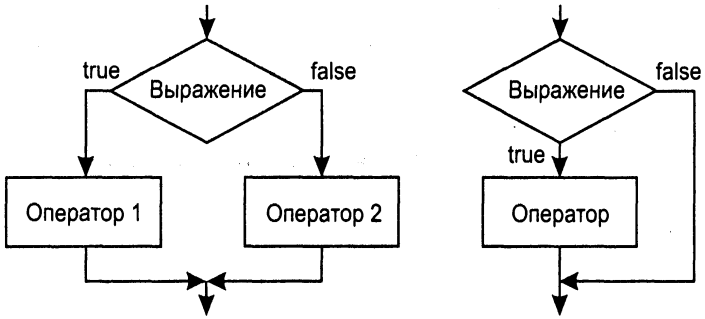


Рис. 3.3. Структурная схема оператора if

Формат оператора:

```
if ( выражение ) оператор_1; [else оператор_2;]
```

Сначала вычисляется выражение, которое может иметь арифметический тип или тип указателя. Если оно не равно нулю (имеет значение true), выполняется первый оператор, иначе – второй. После этого управление передаётся на оператор, следующий за условным.

Для примера присвоим переменной a значение максимума из двух величин x и y

```
if   (x > y)
a = x;
else
a = y;
```

Конструкция else является необязательной в операторе if. Так например:

```
if   (x < 0)
x = -x;
abs = x;
```

Здесь оператор x = -x; выполняется только в том случае, если значение переменной x было отрицательным. Присваивание переменной abs выполняется в любом случае. Таким образом, приведённый фрагмент программы изменит значение переменной x на его абсолютное значение и присвоит переменной abs новое значение x.

Если в случае истинности условия необходимо выполнить несколько операторов, их можно заключить в фигурные скобки {}:

```
if   (x < 0)
{
x = -x;
printf("Изменить значение x на противоположное по знаку");
}
abs = x;
```

Теперь, если x отрицательно, то не только его значение изменится на противоположное, но и будет выведено соответствующее сообщение. Фактически, заключая несколько операторов в фигурные скобки, мы сделали из них один сложный оператор или блок. Приём заключения нескольких операторов в блок работает везде, где нужно поместить несколько операторов вместо одного.

Условный оператор можно расширить для проверки нескольких условий:

```
if   (x  < 0)
printf("Отрицательная величина");
else if   (x > 0)
printf("Положительная величина");
else
printf("Ноль");
```

Следует отметить, что конструкций else if может быть несколько.

Оператор выбора switch предназначен для разветвления процесса вычислений на несколько направлений. Структурная схема оператора приведена на рис. 3.4. Формат оператора:

```
switch ( выражение )
{
case константное_выражение_1: [список_операторов_1]
case константное_выражение_2: [список_операторов_2]
…
case константное_выражение_n: [список_операторов_n]
[default: операторы ]
}
```

Выполнение оператора начинается с вычисления выражения (оно должно быть целочисленным), а затем управление передаётся первому оператору из списка, помеченного константным выражением, значение которого совпало с вычисленным. После этого, если выход из переключателя явно не указан, последовательно выполняются все остальные ветви.

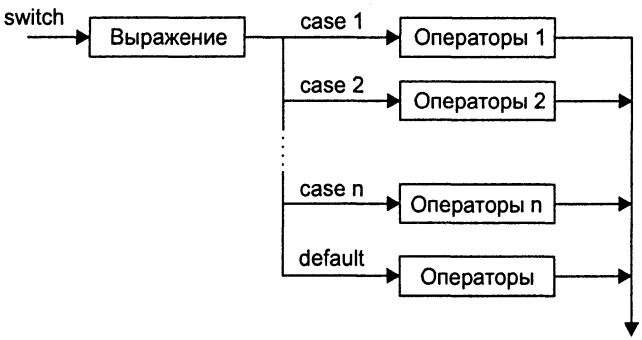


Рис. 3.4. Структурная схема оператора switch

Выход из переключателя обычно выполняется с помощью операторов break или return. Оператор break выполняет выход из самого внутреннего из объемлющих его операторов switch, for, while и do. Оператор return выполняет выход из функции, в теле которой он записан.

Например, предположим, что в переменной code хранится целое число от 0 до 2, и нам нужно выполнить различные действия в зависимости от её значения:

```
switch (code)
{
case 0:
printf("код ноль");
x = x + 1;
break;
case 1:
printf("код один");
y = y + 1;
break;
case 2:
printf("код два");
z = z + 1;
break;
default:
printf("Необрабатываемое значение");
}
```

В зависимости от значения code управление передаётся на одну из меток case. Выполнение оператора заканчивается по достижении либо оператора break, либо конца оператора switch. Таким образом, если code равно 1, выводится "код один", а затем переменная y увеличивается на единицу. Если бы после этого не стоял оператор break, то управление "провалилось" бы дальше, была бы выведена фраза "код два", и переменная z тоже увеличилась бы на единицу.

Если значение переключателя не совпадает ни с одним из значений меток case, то выполняются операторы, записанные после метки default. Метка default может быть опущена, что эквивалентно записи:

```
default:
;  // пустой оператор, не выполняющий
// никаких действий
```

Приведённый пример можно переписать с помощью оператора if:

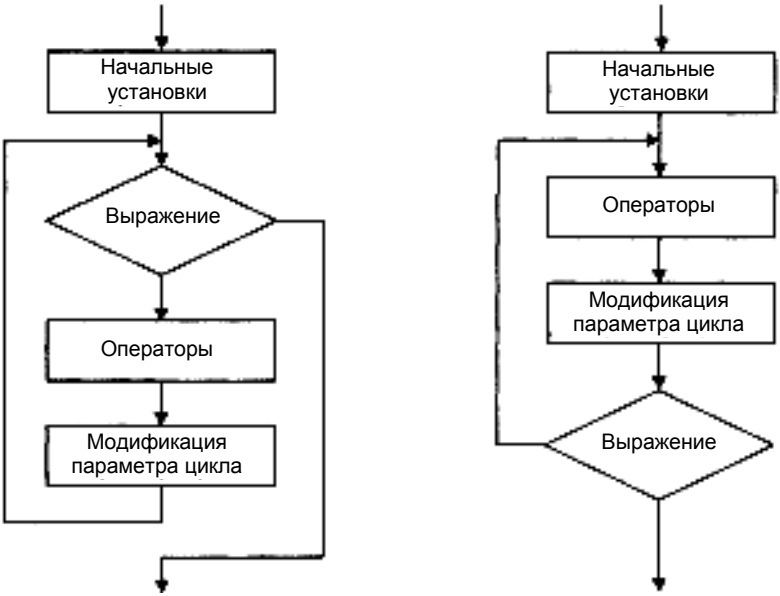
```
if   (code == 0)
{
printf("код ноль");
x = x + 1;
}
else if (code == 1)
{
printf("код один");
y = y + 1;
}
else if (code == 2)
{
printf("код два");
z = z + 1;
}
else printf("Необрабатываемое значение");
```

Запись с помощью оператора переключения switch более наглядна. Особенно часто переключатель используется, когда значение выражения имеет тип набора.

3.3.3. ОПЕРАТОРЫ ЦИКЛА

Циклы используются для организации многократно повторяющихся вычислений. Любой цикл состоит из тела цикла, т.е. тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла (рис. 3.5).

Один проход цикла называется итерацией. Проверка условия выполняется на каждой итерации либо до тела цикла (тогда говорят о цикле с предусловием), либо после тела цикла (цикл с постусловием). Разница между ними состоит в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять ещё раз. Проверка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу. Переменные, изменяющиеся в теле цикла и используемые при проверке цикла для продолжения, называются параметрами цикла. Целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации, называются *счётчиками цикла*.



а) *б*)

Рис. 3.5. Структурные схемы операторов циклов:
а – цикл с предусловием; *б* – цикл с постусловием

Начальные установки могут явно не присутствовать в программе, их смысл состоит в том, чтобы до входа в цикл задать значения переменным, которые в нём используются.

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы break, continue, return и goto. Передавать управление извне внутрь цикла не рекомендуется.

3.3.3.1. Цикл с предусловием (while)

Цикл с предусловием реализует структурную схему, приведённую на рис. 3.5, *а*, и имеет следующий формат:

```
while (выражение) тело;
```

В качестве выражения допускается использовать любое выражение языка C++, а в качестве тела любой оператор, в том числе пустой или составной. Схема выполнения оператора while следующая:

1. Вычисляется выражение.
2. Если выражение ложно, то выполнение оператора while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора while.
3. Процесс повторяется с пункта 1.

Например, выведем таблицу значений функции y = x*x + 1 для заданного диапазона.

```
#include <stdio.h>
void main()
{
float Xn, Xk, Dx;
printf("Введите диапазон и шаг изменения аргумента: " );
scanf ("%f%f%f", &Xn, &Xк, &Dx):
printf("|  X  |  Y  |\n"): // шапка таблицы
float X = Xn; // установка параметра цикла
while (X <= Xk) // проверка условия продолжения
{
printf("| %5.2f | %5.2f |\n", X, X*X + 1); // тело цикла
X += Dx; // модификация параметра
}
}
```

3.3.3.2. Цикл с постусловием (do while)

Оператор цикла do while называется оператором цикла с постусловием и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз, структурная схема показана на рис. 3.5, *б*. Формат оператора имеет следующий вид:

```
do тело while (выражение);
```

Схема выполнения оператора do while:

1. Выполняется тело цикла (которое может быть составным оператором).
2. Вычисляется выражение.
3. Если выражение ложно, то выполнение оператора do while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с п. 1.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор break.

Вычислим квадратный корень вещественного аргумента X с заданной точностью Eps по итерационной формуле

$$y_n = 0{,}5\left(y_{n-1} + \frac{x}{y_{n-1}} \right),$$

где y_{n-1} – предыдущее приближение к корню (в начале вычислений выбирается произвольно); y_n – последующее приближение.

Процесс вычислений прекращается, когда приближения станут отличаться друг от друга по абсолютной величине менее, чем на величину заданной точности. Для вычисления абсолютной величины используется стандартная функция fabs(), объявление которой находится в заголовочном файле <math.h>.

```
#include <stdio.h>
#include <math.h>
int main()
{
double X, Eps; // аргумент и точность
double Yp, Y = 1; // предыдущее и последующее приближение
pr1ntf("Введите аргумент и точность: " );
scanf("%lf%lf", &X, &Eps);
do
{
Yp = Y;
Y = (Yp + X / Yp) / 2;
}while (fabs(Y – Yp) >= Eps);
printf("\nКорень из %lf равен %lf", X, Y);
return 0;
}
```

3.3.3.3. Цикл с параметром (for)

Оператор цикла с параметром for имеет следующий формат:

```
for (выражение_1; выражение_2; выражение_3) тело
```

Выражение_1 обычно используется для установления начального значения переменных, управляющих циклом. Выражение_2 – это выражение, определяющее условие, при котором тело цикла будет выполняться. Выражение_3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора for:

1. Вычисляется выражение_1.
2. Вычисляется выражение_2.
3. Если значение выражения_2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение_3 и осуществляется переход к пункту 2, если выражение_2 равно нулю (ложь), то управление передаётся на оператор, следующий за оператором for.

Существенно то, что проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

В этом примере вычисляются квадраты чисел от 1 до 9.

```
for (i = 1; i < 10; i++) b = i*i;
```

Рассмотрим пример из параграфа 3.3.3.1 о таблице значений функции y = x*x + 1 для заданного диапазона, но решённую с помощью цикла for.

```
#include <stdio.h>
void main()
{
float X, Xn, Xk, Dx;
printf("Введите диапазон и шаг изменения аргумента: " );
scanf ("%f%f%f", &Xn, &Xк, &Dx):
printf("|   X   |   Y   |\n"): // шапка таблицы
for(X = Xn; X <= Xk; X += Dx)
printf("|  %5.2f  |  %5.2f  |\n", X, X*X + 1); // тело цикла
}
```

Некоторые варианты использования оператора for повышают его гибкость за счёт возможности использования нескольких переменных, управляющих циклом.

```
int top,  bot;
char string[100],  temp;
for (top = 0, bot = 100 ; top < bot ; top++, bot--)
{
temp = string[top];
string[bot] = temp;
}
```

Здесь реализуется запись строки символов в обратном порядке, для управления циклом используются две переменные top и bot. Отметим, что на месте выражения_1 и выражения_3 здесь используются

несколько выражений, записанных через запятую и выполняемых последовательно.

Проиллюстрируем особенности трёх типов цикла на примере вычисления приближённого значения

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

для заданного значения *x*. Вычисления будем продолжать до тех пор, пока очередной член ряда остаётся больше заданной точности. Обозначим точность через eps, результат – b, очередной член ряда – r, номер члена ряда – i. Для получения i-го члена ряда нужно (i – l)-й член умножить на x и разделить на i, что позволяет исключить операцию возведения в степень и явное вычисление факториала. Опустив определения переменных, операторы ввода и проверки исходных данных, а также вывода результатов, запишем три фрагмента программ.

```
/*  Цикл с предусловием  */
i = 2;
b = 1.0;
r = x;
while(r > eps || r < -eps)
{
b = b + r;
r = r*x / i;
i++ ;
}
```

Так как проверка точности проводится до выполнения тела цикла, то для окончания цикла абсолютное значение очередного члена должно быть меньше или равно заданной точности.

```
/*  Цикл  с постусловием  */
i = l;
b = 0.0;
r = l.0;
do
{
b = b + r;
r = r*x / i;
i++;
}
while(r >= eps || r <= -eps);
```

Так как проверка точности осуществляется после выполнения тела цикла, то условие окончания цикла – абсолютное значение очередного члена строго меньше заданной точности. Соответствующие изменения внесены и в операторы, выполняемые до цикла.

```
/* Параметрический цикл */
i = 2;
b = 1.0;
r = x;
for( ; r > eps || r < -eps ; )
{
b = b + r;
r = r*x/ i;
i = i + l;
}
```

Условие окончания параметрического цикла такое же, как и в цикле while.

Все три цикла записаны по возможности одинаково, чтобы подчеркнуть сходство циклов. Однако в данном примере цикл for имеет существенные преимущества. В заголовок цикла (в качестве выражения_1) можно ввести инициализацию всех переменных:

```
for   (i = 2, b = 1.0, r = x ; r > eps || r < -eps ;)
{
b = b + r;
r = r*x/ i;
i = i + l;
}
```

В выражение_3 можно включать операцию изменения счётчика членов ряда:

```
for(i = 2, b = 1.0, r = x ; r > eps || r < -eps ;   i++)
{
b = b + r;
r = r*x /i;
}
```

Можно ещё более усложнить заголовок, перенеся в него все исполнимые операторы тела цикла:

```
for(i = 2, b = 1.0, r = x ; r > eps || r < -eps; b+ = r,   r* = x / i,   i++);
```

В данном случае тело цикла – пустой оператор. Для сокращения выражения_3 в нём использованы составные операции присваивания и операция ++.

3.3.4. Операторы передачи управления

В C++ есть четыре оператора, изменяющих естественный порядок выполнения вычислений:

– оператор безусловного перехода goto;
– оператор выхода из цикла break;
– оператор перехода к следующей итерации цикла continue;
– оператор возврата из функции return.

Оператор goto (оператор безусловного перехода) имеет формат:

```
goto метка;
```

В теле той же функции должна присутствовать ровно одна конструкция вида:

```
метка: оператор;
```

Оператор goto передаёт управление на помеченный оператор. Метка – это обычный идентификатор, областью видимости которого является функция, в теле которой он задан.

Использование оператора безусловного перехода оправдано в двух случаях: принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей; переход из нескольких мест функции в одно (например, если перед выходом из функции всегда необходимо выполнять какие-либо действия).

В остальных случаях для записи любого алгоритма существуют более подходящие средства, а использование goto приводит только к усложнению структуры программы и затруднению отладки. Применение goto нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и один выход.

Оператор break обеспечивает прекращение выполнения самого внутреннего из объединяющих его операторов switch, do, for, while. После выполнения оператора break управление передаётся оператору, следующему за прерванным.

Оператор continue используется только внутри операторов цикла, но в отличие от оператора break, выполнение программы продолжается не с оператора, следующего за прерванным оператором, а с начала прерванного оператора. Оператор continue, как и оператор break, прерывает самый внутренний из объемлющих его циклов.

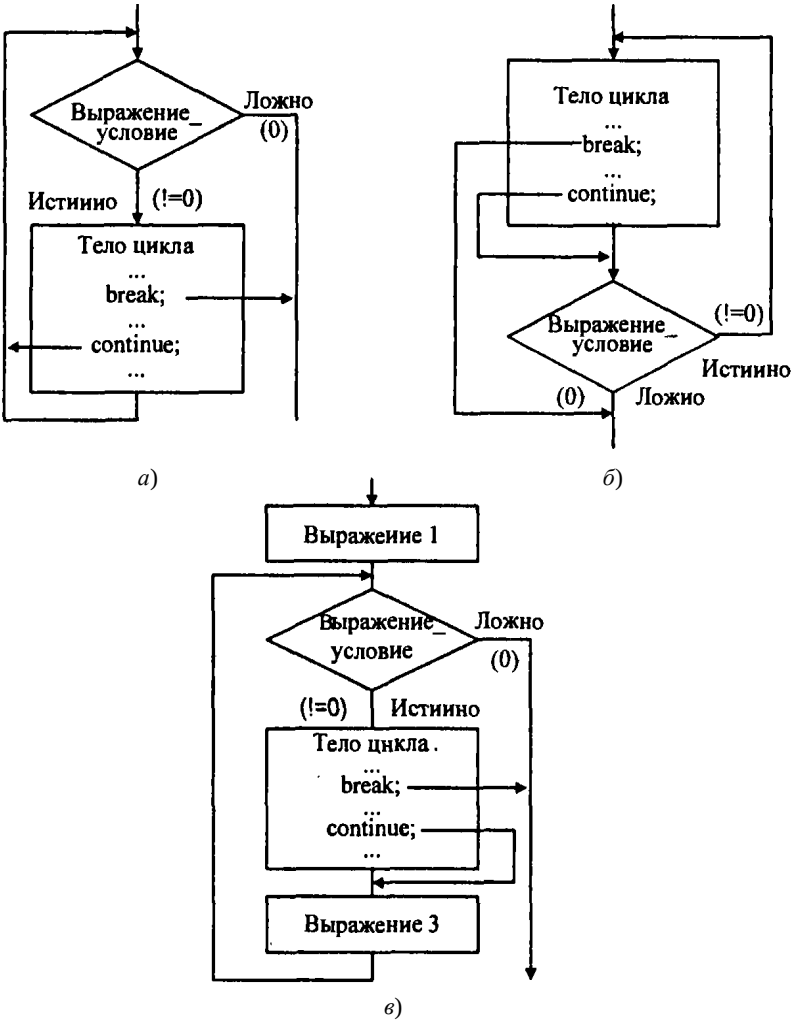


Рис. 3.6. Схема выполнения в циклах операторов break и continue:
а – цикл с предусловием; *б* – цикл с постусловием; *в* – цикл с параметром

Схема выполнения в циклах операторов break и continue показана на рис. 3.6.

Оператор return (оператор возврата из функции) завершает выполнение функции и передаёт управление в точку её вызова. Вид оператора:

```
return [выражение];
```

Выражение должно иметь скалярный тип. Если тип возвращаемого функцией значения описан как void, выражение должно отсутствовать.

3.4. МАССИВЫ

Математическими понятиями, которые привели к появлению в языках программирования понятия «массив», являются матрица и её частные случаи: вектор-столбец или вектор-строка. Элементы матриц в математике принято обозначать с использованием индексов. Существенно, что все элементы матриц либо вещественные, либо целые и т.п.

Такая «однородность» элементов свойственна и массиву, определение которого описывает тип элементов, имя массива и его размерность, т.е. число индексов, необходимое для обозначения конкретного элемента.

Кроме того, в определении указывается количество значений, принимаемых каждым индексом.

Объявление массива имеет два формата:

```
спецификатор_типа описатель [константное_выражение];
спецификатор_типа описатель [ ];
```

Описатель – это идентификатор массива. Спецификатор_типа задаёт тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа void.

Константное_выражение в квадратных скобках задаёт количество элементов массива. Константное_выражение при объявлении массива может быть опущено в следующих случаях: при объявлении массив инициализируется; массив объявлен как формальный параметр функции; массив объявлен как ссылка на массив, явно определённый в другом файле.

Так, массив из 10 элементов a[0], a[1], ..., a[9] определяется как

```
int a[10];
```

Двумерный массив, первый индекс которого принимает 13 значений от 0 до 12, второй индекс принимает 6 значений от 0 до 5, определяется следующим образом:

```
float Z[13][[6];
```

Элементы двумерного массива Z можно перечислить так: Z[0][0], Z[0][l], Z[0][2], ..., Z[12][4], Z[12][5].

В соответствии с синтаксисом языка С существуют только одномерные массивы, однако элементами одномерного массива, в свою очередь, могут быть массивы. Поэтому двумерный массив определяется как массив массивов. Таким образом, в примере определён массив Z из 13 элементов-массивов, каждый из которых, в свою очередь, состоит из 6 элементов типа float. Обратите внимание, что нумерация элементов любого массива всегда начинается с 0, т.е. индекс изменяется от 0 до N – 1, где N – количество значений индекса.

Инициализация массива. При определении массивов возможна их инициализация, т.е. присваивание начальных значений их элементам. По существу (точнее по результату), инициализация – это объединение определения объекта с одновременным присваиванием ему конкретного значения. Использование инициализации позволяет изменить формат определения массива. Например, можно явно не указывать количество элементов одномерного массива, а только перечислить их начальные значения в списке инициализации:

```
double d[ ] = {1.0,   2.0,   3.0,   4.0,   5.0};
```

Здесь длину массива компилятор вычисляет по количеству начальных значений, перечисленных в фигурных скобках. После такого определения элемент d[0] равен 1.0, d[l] равен 2.0 и т.д. до d[4], который равен 5.0.

Если в определении массива явно указан его размер, то количество начальных значений не может быть больше количества элементов в массиве. Если количество начальных значений меньше, чем объявленная длина массива, то начальные значения получат только первые элементы массива (с меньшими значениями индекса):

```
int M[8] = {8, 4, 2};
```

В данном примере определены значения только переменных M[0], M[1] и M[2], равные соответственно 8, 4 и 2. Элементы M[3], ..., M[7] не инициализируются.

Правила инициализации многомерных массивов соответствуют определению многомерного массива как одномерного, элементами которого служат массивы, размерность которых на единицу меньше, чем у исходного массива. Одномерный массив инициализируется заключённым в фигурные скобки списком начальных значений. В свою очередь, начальное значение, если оно относится к массиву, также

представляет собой заключённый в фигурные скобки список начальных значений. Например, присвоить начальные значения вещественным элементам двумерного массива A, состоящего из трёх строк и двух столбцов, можно следующим образом:

```
double A[3][2] = {{10, 20),  {30, 40},  {50, 60}};
```

Эта запись эквивалентна последовательности операторов присваивания:

```
A[0][0] = 10; A[0][1] = 20; A[1][0] = 30; A[1][1] = 40; A[2][0] = 50;
A[2][1] = 60;
```

Тот же результат можно получить с одним списком инициализации:

```
double A[3][2] = {10, 20, 30, 40, 50, 60};
```

С помощью инициализаций можно присваивать значения не всем элементам многомерного массива. Например, чтобы инициализировать только элементы первого столбца матрицы, её можно описать так:

```
double Z[4][6] = {{1},  {2},  {3},  {4}};
```

Следующее описание формирует «треугольную» матрицу в целочисленном массиве из 5 строк и 4 столбцов:

```
int x[5][4] = {{1),  {2, 3},  {4, 5, 6},  {7, 8, 9, 10}  };
```

В данном примере последняя пятая строка x[4] остаётся незаполненной. Первые три строки заполнены не до конца. Схема размещения элементов массива изображена на рис. 3.7.

№/№	0	1	2	3
0	1	–	–	–
1	2	3	–	–
2	4	5	6	–
3	7	8	9	10
4	–	–	–	–

Рис. 3.7. Схема размещения элементов массива

Для работы с массивом мы индексируем (нумеруем) его элементы, а доступ к ним осуществляется с помощью операции взятия индекса. Так, инициализируем одномерный массив и запишем первый элемент массива в переменную first_elem:

```
int m[9] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
int first_elem = m[0];
```

Для перебора элементов массива обычно употребляют инструкцию цикла. Вот пример программы, которая инициализирует массив из десяти элементов числами от 0 до 9 и затем печатает их в обратном порядке:

```
int ia[10];
int index;
for (index = 0; index < 10; index++) ia[index] = index;
for (index = 9; index >= 0; index--) printf("%d ", ia[index]);
```

Несмотря на то, что в C++ встроена поддержка для типа данных «массив», она весьма ограничена. Фактически мы имеем лишь возможность доступа к отдельным элементам массива. C++ не поддерживает абстракцию массива, не существует операций над массивами в целом, таких, например, как присвоение одного массива другому или сравнение двух массивов на равенство, и даже такой простой, на первый взгляд, операции, как получение размера массива. Мы не можем скопировать один массив в другой, используя простой оператор присваивания.

Для этого мы должны программировать такую операцию с помощью цикла:

```
int A[10], B[10];
for (int index = 0; index < 10; ++index) A[index] = B[index];
```

Массив «не знает» собственный размер, поэтому мы должны сами следить за тем, чтобы случайно не обратиться к несуществующему элементу массива. Это становится особенно утомительным в таких ситуациях, как передача массива функции в качестве параметра. Можно сказать, что этот встроенный тип достался языку C++ в наследство от C и процедурно-ориентированной парадигмы программирования.

3.5. УКАЗАТЕЛИ

Когда компилятор обрабатывает оператор определения переменной, например,

```
int i = 10;
```

он выделяет память в соответствии с типом (int) и инициализирует её указанным значением (10). Все обращения в программе к переменной по её имени (i) заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Мы можем определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями.

Указатели предназначены для хранения адресов областей памяти. В C++ различают три вида указателей – указатели на объект, на функцию и на void, отличающиеся свойствами и набором допустимых операций. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим конкретным типом.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определённого типа (основного или составного). Простейшее объявление указателя на объект (в дальнейшем называемого просто указателем) имеет вид:

```
тип *имя;
```

где тип может быть любым, кроме ссылки и битового поля, причём тип может быть к этому моменту только объявлен, но ещё не определён (следовательно, в структуре, например, может присутствовать указатель на структуру того же типа).

Звёздочка относится непосредственно к имени, поэтому для того, чтобы объявить несколько указателей, требуется ставить её перед именем каждого из них. Например, в операторе

```
int *a, b, *c;
```

описываются два указателя на целое с именами a и c, а также целая переменная b.

Размер указателя зависит от модели памяти. Можно определить указатель на указатель и т.д.

Указатель на void применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определён (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Указателю на void можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

Указатель может быть константой или переменной, а также указывать на константу или переменную. Рассмотрим примеры:

```
int i; // целая переменная
const int ci = 1; // целая константа
int *pi; // указатель на целую переменную
const int * pci; // указатель на целую константу
int * const cp = &i; // указатель-константа на целую переменную
const int * const cpc = &ci; // указатель-константа на целую константу
```

Как видно из примеров, модификатор const, находящийся между именем указателя и звёздочкой, относится к самому указателю и запрещает его изменение, а const слева от звёздочки задаёт постоянство значения, на которое он указывает. Для инициализации указателей использована операция получения адреса &.

Величины типа указатель подчиняются общим правилам определения области действия, видимости и времени жизни.

3.5.1. ИНИЦИАЛИЗАЦИЯ УКАЗАТЕЛЕЙ

Указатели чаще всего используют при работе с динамической памятью, называемой иногда кучей (heap). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти, называемым динамическими переменными, производится только через указатели. Время жизни динамических переменных – от точки создания до конца программы или до явного освобождения памяти. В C++ используется два способа работы с динамической памятью. Первый использует семейство функций malloc и достался в наследство от C, второй использует операции new и delete.

При определении указателя надо стремиться выполнить его инициализацию, т.е. присвоение начального значения. Непреднамеренное использование неинициализированных указателей – распространённый источник ошибок в программах. Инициализатор записывается после имени указателя либо в круглых скобках, либо после знака равенства.

Существуют следующие способы инициализации указателя:

1. Присваивание указателю адреса существующего объекта:

– с помощью операции получения адреса:

```
int a = 5; // целая переменная
int* p = &a; //в указатель записывается адрес a
int* p (&a); // то же самое другим способом
```

– с помощью значения другого инициализированного указателя:

```
int* r = p;
```

– с помощью имени массива или функции, которые трактуются как адрес:

```
int b[10]; // массив
int * t = b; // присваивание адреса начала массива
void f (int a){ /* ... * / } // определение функции
void (*pf) (int); // указатель на функцию
pf = f; // присваивание адреса функции
```

2. Присваивание указателю адреса области памяти в явном виде:

```
char* vp = (char *)0xB8000000;
```

Здесь 0xB8000000 – шестнадцатеричная константа, (char *) – операция приведения типа: константа преобразуется к типу «указатель на char».

3. Присваивание пустого значения:

```
int* s = NULL;
int* r = 0;
```

В первой строке используется константа NULL, определённая в некоторых заголовочных файлах C как указатель, равный нулю. Рекомендуется использовать просто 0, так как это значение типа int будет правильно преобразовано стандартными способами в соответствии с контекстом. Поскольку гарантируется, что объектов с нулевым адресом нет, пустой указатель можно использовать для проверки, ссылается указатель на конкретный объект или нет.

4. Выделение участка динамической памяти и присваивание её адреса указателю:

– с помощью операции new:

```
int* n = new int; //1
int* m = new int (10); // 2
int* q = new int [10]; // 3
```

– с помощью функции malloc:

```
int* u = (int *)malloc(sizeof(int)); // 4
```

В операторе 1 операция new выполняет выделение достаточного для размещения величины типа int участка динамической памяти и записывает адрес начала этого участка в переменную n. Память под саму переменную n (размера, достаточного для размещения указателя) выделяется на этапе компиляции.

В операторе 2, кроме описанных выше действий, производится инициализация выделенной динамической памяти значением 10.

В операторе 3 операция new выполняет выделение памяти под 10 величин типа int (массива из 10 элементов) и записывает адрес начала этого участка в переменную q, которая может трактоваться как имя массива. Через имя можно обращаться к любому элементу массива. Работу с динамическими массивами рассмотрим ниже.

Если память выделить не удалось, по стандарту должно порождаться исключение bad_alloc. Старые версии компиляторов могут возвращать 0.

В операторе 4 делается то же самое, что и в операторе 1, но с помощью функции выделения памяти malloc, унаследованной из библиотеки C. В функцию передаётся один параметр – количество выделяемой памяти в байтах. Конструкция (int*) используется для приведения типа указателя, возвращаемого функцией, к требуемому типу. Если память выделить не удалось, функция возвращает 0.

Операцию new использовать предпочтительнее, чем функцию malloc, особенно при работе с объектами.

Освобождение памяти, выделенной с помощью операции new, должно выполняться с помощью delete, а памяти, выделенной функцией malloc – посредством функции free. При этом переменная-указатель сохраняется и может инициализироваться повторно. Приведённые выше динамические переменные уничтожаются следующим образом:

```
delete n;
delete m;
delete [ ] q;
free (u);
```

Если память выделялась с помощью new[], для освобождения памяти необходимо применять delete[]. Размерность массива при этом не указывается. Если квадратных скобок нет, то никакого сообщения об ошибке не выдаётся, но помечен как свободный будет только первый элемент массива, а остальные окажутся недоступны для дальнейших операций. Такие ячейки памяти называются мусором.

Если переменная-указатель выходит из области своего действия, отведённая под неё память освобождается. Следовательно, динамиче-

ская переменная, на которую ссылался указатель, становится недоступной. При этом память из-под самой динамической переменной не освобождается. Другой случай появления «мусора» – когда инициализированному указателю присваивается значение другого указателя. При этом старое значение бесследно теряется.

3.5.2. ОПЕРАЦИИ С УКАЗАТЕЛЯМИ

С указателями можно выполнять следующие операции: разадресация, или косвенное обращение к объекту (*), присваивание, сложение с константой, вычитание, инкремент (++), декремент (--), сравнение, приведение типов. При работе с указателями часто используется операция получения адреса (&).

Операция разадресации, или разыменования, предназначена для доступа к величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины (если она не объявлена как константа):

```
char a; // переменная типа char
char * p = new char; // выделение памяти под указатель и под динамическую
// переменную типа char
*p = 'Ю'; a = *p; // присваивание значения обеим переменным
```

Как видно из примера, конструкцию *имя_указателя можно использовать в левой части оператора присваивания, так как она является L-значением, т.е. определяет адрес области памяти. Для простоты эту конструкцию можно считать именем переменной, на которую ссылается указатель. С ней допустимы все действия, определённые для величин соответствующего типа (если указатель инициализирован). На одну и ту же область памяти могут ссылаться несколько указателей различного типа. Применённая к ним операция разадресации даст разные результаты. Например, программа

```
#include <std1o.h>
int main()
{
unsigned long int A = 0Xcc77ffaa;
unsigned short int* pint = (unsigned short int*) &A;
unsigned char* pchar = (unsigned char *) &A;
printf(" | %x  | %x  | %x  | ", A, *pint, *pchar);
return 0;
}
```

на IBM PC-совместимом компьютере выведет на экран строку:

| cc77ffaa | ffaa | aa |

Значения указателей pint и pchar одинаковы, но разадресация pchar даёт в результате один младший байт по этому адресу, а pint – два младших байта.

В приведённом выше примере при инициализации указателей были использованы операции приведения типов. Синтаксис операции явного приведения типа прост: перед именем переменной в скобках указывается тип, к которому её требуется преобразовать. При этом не гарантируется сохранение информации, поэтому в общем случае явных преобразований типа следует избегать.

При смешивании в выражении указателей разных типов явное преобразование типов требуется для всех указателей, кроме void*. Указатель может неявно преобразовываться в значение типа bool (например, в выражении условного оператора), при этом ненулевой указатель преобразуется в true, а нулевой в false.

Присваивание без явного приведения типов допускается в двух случаях: указателям типа void*, если тип указателей справа и слева от операции присваивания один и тот же.

Таким образом, неявное преобразование выполняется только к типу void*. Значение 0 неявно преобразуется к указателю на любой тип. Присваивание указателей на объекты указателям на функции (и наоборот) недопустимо. Запрещено и присваивать значения указателям-константам, впрочем, как и константам любого типа (присваивать значения указателям на константу и переменным, на которые ссылается указатель-константа, допускается).

Арифметические операции с указателями (сложение с константой, вычитание, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, последовательно размещёнными в памяти, например, с массивами.

Инкремент перемещает указатель к следующему элементу массива, декремент – к предыдущему. Фактически значение указателя изменяется на величину sizeof(тип). Если указатель на определённый тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа, например:

```
short * p = new short [5];
p++; // значение p увеличивается на 2
long * q = new long [5];
q++; // значение q увеличивается на 4
```

Разность двух указателей – это разность их значений, делённая на размер типа в байтах (в применении к массивам разность указателей, например, на третий и шестой элементы равна 3). Суммирование двух указателей не допускается.

При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе

```
*p++ = 10;
```

Операции разадресации и инкремента имеют одинаковый приоритет и выполняются справа налево, но, поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе p, будет записано значение 10, а затем указатель будет увеличен на количество байт, соответствующее его типу. То же самое можно записать подробнее:

```
*p = 10; p++;
```

Выражение (*p)++, напротив, инкрементирует значение, на которое ссылается указатель.

Унарная операция получения адреса & применима к величинам, имеющим имя и размещённым в оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной.

Ссылка представляет собой синоним имени, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается.

Формат объявления ссылки:

```
тип & имя;
```

где тип – это тип величины, на которую указывает ссылка; & – оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа, например:

```
int k;
int& p = k; // ссылка p – альтернативное имя для k
const char& c = ' \n '; // ссылка на константу
```

Запомните следующие правила.

1. Переменная-ссылка должна явно инициализироваться при её описании, кроме случаев, когда она является параметром функции, описана как extern или ссылается на поле данных класса.

2. После инициализации ссылке не может быть присвоена другая переменная.

3. Тип ссылки должен совпадать с типом величины, на которую она ссылается.

4. Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

5. Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость.

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

3.5.3. ДИНАМИЧЕСКИЕ МАССИВЫ

Динамические массивы создают с помощью операции распределения памяти (new, malloc и т.п.), при этом необходимо указать тип и размерность, например:

```
int n = 100;
float *p = new float [n];
```

Здесь создаётся переменная-указатель на float, в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов вещественного типа, и адрес её начала записывается в указатель p. Динамические массивы нельзя при создании инициализировать, и они не обнуляются.

Преимущество динамических массивов состоит в том, что размерность может быть переменной, т.е. объём памяти, выделяемой под массив, определяется на этапе выполнения программы. Доступ к элементам динамического массива осуществляется точно так же, как к статическим, например, к элементу номер 5 приведённого выше массива можно обратиться как p[5] или *(p + 5).

Альтернативный способ создания динамического массива – использование функции malloc библиотеки C:

```
int n = 100;
float *q = (float *) malloc(n *sizeof(float));
```

Операция преобразования типа, записанная перед обращением к функции malloc, требуется потому, что функция возвращает значение указателя типа void*, а инициализируется указатель на float.

Память, зарезервированная под динамический массив с помощью new [], должна освобождаться оператором delete [], а память, выделенная функцией malloc – посредством функции free, например:

```
delete [ ] p;
free (q);
```

При несоответствии способов выделения и освобождения памяти результат не определён. Размерность массива в операции delete не указывается, но квадратные скобки обязательны.

Многомерные массивы задаются указанием каждого измерения в квадратных скобках, например, оператор

```
int matr [6][8];
```

задаёт описание двумерного массива из 6 строк и 8 столбцов. В памяти такой массив располагается в последовательных ячейках построчно. Многомерные массивы размещаются так, что при переходе к следующему элементу быстрее всего изменяется последний индекс. Для доступа к элементу многомерного массива указываются все его индексы, например matr[i][j], или более экзотическим способом: *(matr[i] + j) или *(*(matr + i) + j). Это возможно, поскольку matr[i] является адресом начала i-й строки массива.

При инициализации многомерный массив представляется либо как массив из массивов, при этом каждый массив заключается в свои фигурные скобки (в этом случае левую размерность при описании можно не указывать), либо задаётся общий список элементов в том порядке, в котором элементы располагаются в памяти:

```
int mass2 [][2] = { {1, 1}, {0, 2}, {1, 0} };
int mass2 [3][2] = {1, 1, 0, 2, 1, 0};
```

Для создания динамического массива необходимо указать все его размерности. Рассмотрим самый универсальный способ выделения памяти под двумерный массив, когда обе его размерности задаются на этапе выполнения программы:

```
int nstr. nstb:
printf("Введите количество строк и столбцов :");
scanf("%d%d", nstr, nstb);
int **a = new int * [ nstr ] ; // 1
for(int i = 0; i<nstr; i++) // 2
a[i] = new int [nstb]; // 3
```

В операторе 1 объявляется переменная типа «указатель на указатель на int» и выделяется память под массив указателей на строки массива (количество строк – nstr). В операторе 2 организуется цикл для выделения памяти под каждую строку массива. В операторе 3 каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двумерного массива. Каждая строка состоит из nstb элементов типа int (рис. 3.8).

Освобождение памяти из-под массива с любым количеством измерений выполняется с помощью операции delete [ ]:

```
for(int i = 0; i<nstr; i++)
delete [] a[i];
delete [] a;
```

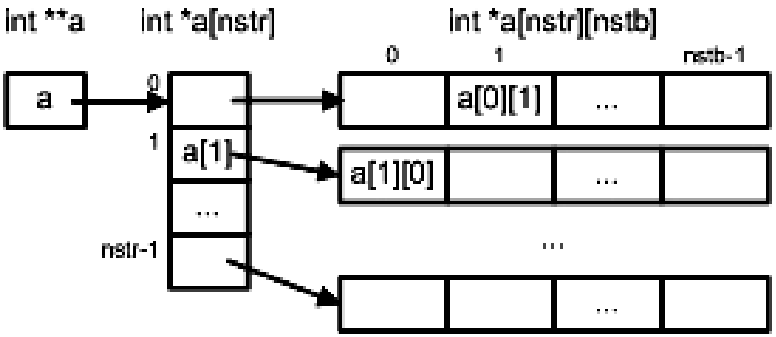


Рис. 3.8. Выделение памяти под двухмерный массив

3.6. ФУНКЦИИ В ЯЗЫКЕ C++

С увеличением объёма программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является её разбиение на части. В C++ задача может быть разделена на более простые и обозримые с помощью функций, после чего программу можно рассматривать в более укрупнённом виде – на уровне взаимодействия функций. Использование функций является первым шагом к повышению степени абстракции программы и ведёт к упрощению её структуры.

Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать её на выполнение можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции можно помещать в библиотеки. Таким образом создаются более простые в отладке и сопровождении программы.

3.6.1. ОБЪЯВЛЕНИЕ ФУНКЦИИ

Функцию можно рассматривать как операцию, определённую пользователем. В общем случае она задаётся своим именем. Операнды функции, или формальные параметры, задаются в списке параметров, через запятую. Такой список заключается в круглые скобки. Результатом функции может быть значение, которое называют возвращаемым. Об отсутствии возвращаемого значения сообщают ключевым словом void. Действия, которые производит функция, составляют её тело; оно заключено в фигурные скобки. Тип возвращаемого значения, её имя, список параметров и тело составляют определение функции. Вот несколько примеров:

```
inline int abs(int obj)
{
// возвращает абсолютное значение iobj
return(iobj < 0 ? -iobj : iobj);
}

inline int min(int p1, int p2)
{
// возвращает меньшую из двух величин
return(pi < p2 ? pi :  p2);
}
```

```
int gcd(int vl, int v2)
{
// возвращает наибольший общий делитель
while(v2)
{
int temp = v2;
v2 = vl % v2;
vl = temp;
}
return vl;
}
```

Выполнение функции происходит тогда, когда в тексте программы встречается оператор вызова. Если функция принимает параметры, при её вызове должны быть указаны фактические параметры, аргументы.

Вызов функции может обрабатываться двумя разными способами. Если она объявлена встроенной (inline), то компилятор подставляет в точку вызова её тело. Во всех остальных случаях происходит нормальный вызов, который приводит к передаче управления ей, а активный в этот момент процесс на время приостанавливается. По завершении работы выполнение программы продолжается с точки, непосредственно следующей за точкой вызова. Работа функции завершается выполнением последней инструкции её тела или специальной инструкции return.

Функция должна быть объявлена до момента её вызова, попытка использовать необъявленное имя приводит к ошибке компиляции. Определение функции может служить её объявлением, но ему разрешено появиться в программе только один раз. Поэтому обычно его помещают в отдельный исходный файл. Иногда в одном файле находятся определения нескольких функций, логически связанных друг с другом. Чтобы использовать их в другом исходном файле, необходим механизм, позволяющий объявить её, не определяя.

Объявление функции состоит из типа возвращаемого значения, имени и списка параметров. Вместе эти три элемента составляют прототип. Объявление может появиться в файле несколько раз.

Объявления (а равно определения встроенных функций) лучше всего помещать в заголовочные файлы, которые могут включаться всюду, где необходимо вызвать функцию. Таким образом, все файлы используют одно общее объявление. Если его необходимо модифицировать, изменения будут локализованы.

В объявлении функции описывается её интерфейс. Он содержит все данные о том, какую информацию должна получать функция (спи-

сок параметров) и какую информацию она возвращает. Для пользователей важны только эти данные, поскольку лишь они фигурируют в точке вызова. Интерфейс помещается в заголовочный файл.

Прототип функции описывает её интерфейс и состоит из типа возвращаемого функцией значения, имени и списка параметров.

Тип возвращаемого функцией значения бывает встроенным, как int или double, составным, как int& или double*, или определённым пользователем – перечислением или классом. Можно также использовать специальное ключевое слово void, которое говорит о том, что функция не возвращает никакого значения:

```
bool look_up( int *, int );
double calc( double );
int count( const string &, char );
int *foo_bar();
```

Список параметров не может быть опущен. Функция, которая не требует параметров, должна иметь пустой список либо список, состоящий из одного ключевого слова void. Например, следующие объявления эквивалентны:

```
int fork();
int fork( void );
```

Такой список состоит из названий типов, разделённых запятыми. После имени типа может находиться имя параметра, хотя это и необязательно. В списке параметров не разрешается использовать сокращённую запись, соотнося одно имя типа с несколькими параметрами:

```
int manip( int vl, v2 );     // ошибка
int manip( int vl, int v2 ); // правильно
```

Имена параметров не могут повторяться. Имена, фигурирующие в определении функции, можно и даже нужно использовать в её теле. В объявлении же функции они не обязательны и служат средством документирования её интерфейса. Например:

```
void print( int *array, int size );
```

Имена параметров в объявлении и в определении одной и той же функции не обязаны совпадать.

C++ допускает сосуществование двух или более функций, имеющих одно и то же имя, но разные списки параметров. Такие функции

называются перегруженными. О списке параметров в этом случае говорят как о сигнатуре функции, поскольку именно он используется для различения разных версий одноимённых функций. Имя и сигнатура однозначно идентифицируют версию.

C++ является строго типизированным языком. Компилятор проверяет аргументы на соответствие типов в каждом вызове функции. Если тип фактического аргумента не соответствует типу формального параметра, то производится попытка неявного преобразования. Если же это оказывается невозможным или число аргументов неверно, компилятор выдаёт сообщение об ошибке. Именно поэтому функция должна быть объявлена до того, как программа впервые обратится к ней: без объявления компилятор не обладает информацией для проверки типов.

3.6.2. ПЕРЕДАЧА АРГУМЕНТОВ ФУНКЦИИ

Функции используют память из стека программы. Некоторая область стека отводится функции и остаётся связанной с ней до окончания её работы, по завершении которой отведённая ей память освобождается и может быть занята другой функцией. Эту часть стека называют областью активации.

Каждому параметру функции отводится место в данной области, причём его размер определяется типом параметра. При вызове функции память инициализируется значениями фактических аргументов.

Стандартным способом передачи аргументов является копирование их значений, т.е. передача по значению. При этом способе функция не получает доступа к реальным объектам, являющимся её аргументами. Вместо этого она получает в стеке локальные копии этих объектов. Изменение значений копий никак не отражается на значениях самих объектов. Локальные копии теряются при выходе из функции.

Значения аргументов при передаче по значению не меняются. Следовательно, программист не должен заботиться о сохранении и восстановлении их значений при вызове функции. Без этого механизма любой вызов мог бы привести к нежелательному изменению аргументов, не объявленных константными явно. Передача по значению освобождает человека от лишних забот в наиболее типичной ситуации.

Однако такой способ передачи аргументов может не устраивать нас в следующих случаях:

– передача большого объекта типа класса. Временные и пространственные расходы на размещение и копирование такого объекта могут оказаться неприемлемыми для реальной программы;

– иногда значения аргументов должны быть модифицированы внутри функции.

Например, swap() должна обменять значения своих аргументов, что невозможно при передаче по значению:

```
// swap() не меняет значений своих аргументов!
void swap( int vl, int v2 )
{
int tmp = v2;
v2 = vl;
vl = tmp;
}
```

Функция swap() обменивает значения локальных копий своих аргументов. Те же переменные, что были использованы в качестве аргументов при вызове, остаются неизменными, что иллюстрирует следующая программа

```
#include <stdio.h>
void swap( int, int );

int main()
{
int i = 10;
int j = 20;
printf("Перед swap() i = %d j = %d\n", i, j);
swap( i, j );
printf("После swap() i = %d j = %d\n", i, j);
return 0;
}
```

Достичь желаемого можно двумя способами. Первый – объявление параметров указателями. Вот как будет выглядеть реализация swap() в этом случае:

```
void pswap( int *vl, int *v2 ) {
int tmp = *v2;
*v2 = *vl;
*vl = tmp;
}
```

Функция main() тоже нуждается в модификации. Вместо передачи самих объектов необходимо передавать их адреса:

```
pswap( &i, &j );
```

Альтернативой может стать объявление параметров ссылками. В данном случае реализация swap() выглядит так:

```
// rswap() обменивает значения объектов,
// на которые ссылаются vl и v2
void rswap( int &vl, int &v2 ) {
int tmp = v2;
v2 = vl;
vl = tmp;
}
```

Вызов этой функции из main() аналогичен вызову первоначальной функции swap():

```
rswap( i, j );
```

Массив в C++ никогда не передаётся по значению, а только как указатель на его первый, точнее нулевой, элемент. Например, объявление

```
void putValues( int[ 10 ] );
```

рассматривается компилятором так, как будто оно имеет вид

```
void putValues( int* );
```

Так же эквивалентна им и следующая запись

```
void putValues( int[] )
```

Передача массивов как указателей имеет следующие особенности:

– изменение значения аргумента внутри функции затрагивает сам переданный объект, а не его локальную копию. Если такое поведение нежелательно, то необходимо позаботиться о сохранении исходного значения. Можно также при объявлении функции указать, что она не должна изменять значение параметра, объявив этот параметр константой:

```
void putValues( const int[ 10 ] );
```

– размер массива не является частью типа параметра. Поэтому функция не знает реального размера передаваемого массива. Компилятор тоже не может это проверить. Рассмотрим пример:

```
void putValues( int[ 10 ] ); // рассматривается как int*
int main() {
int i, j [ 2 ];
putValues( &i ); // правильно: &i это int*;
// однако при выполнении возможна ошибка
putValues( j ); // правильно: j – адрес 0-го элемента – int*;
// однако при выполнении возможна ошибка
}
```

При проверке типов параметров компилятор способен распознать, что в обоих случаях тип аргумента int* соответствует объявлению функции. Однако контроль за тем, не является ли аргумент массивом, не производится.

Другой способ сообщить функции размер массива-параметра – объявить параметр как ссылку. В этом случае размер становится частью типа, и компилятор может проверить аргумент в полной мере.

```
// параметр – ссылка на массив из 10 целых
void putValues( int (&arr)[10] );
int main() {
int i, j [ 2 ];
putValues(i); // ошибка:
// аргумент не является массивом из 10 целых
putValues(j); // ошибка:
// аргумент не является массивом из 10 целых
return 0;
}
```

Поскольку размер массива теперь является частью типа параметра, новая версия putValues() способна работать только с массивами из 10 элементов. Конечно, это ограничивает её область применения, зато реализация значительно проще:

```
#include <stdio.h>
void putValues( int (&ia)[10] )
{
printf("( 10 )< ";
for ( int 1 =0; i < 10; ++i ) { printf("%d",ia[ i ]);

// разделитель, печатаемый после каждого элемента,
// кроме последнего
```

```
    if ( i != 9 )
    printf(" , ");
    }
    printf("\n");
    }
```

Параметр может быть многомерным массивом. Для такого параметра должны быть заданы правые границы всех измерений, кроме первого. Например:

```
    putValues( int matrix[][10], int rowSize );
```

Здесь matrix объявляется как двумерный массив, который содержит десять столбцов и неизвестное число строк. Эквивалентным объявлением для matrix будет:

```
    int (*matrix)[10]
```

Многомерный массив передаётся как указатель на его нулевой элемент. В нашем случае тип matrix – указатель на массив из десяти элементов типа int. Как и для одномерного массива, граница первого измерения не учитывается при проверке типов. Если параметры являются многомерными массивами, то контролируются все измерения, кроме первого.

Заметим, что скобки вокруг *matrix необходимы из-за более высокого приоритета операции взятия индекса. Инструкция объявляет matrix как массив из десяти указателей на int.

```
    int *matrix[10];
```

Значение параметра по умолчанию – это значение, которое разработчик считает подходящим в большинстве случаев употребления функции, хотя и не во всех. Оно освобождает программиста от необходимости уделять внимание каждой детали интерфейса функции.

Значения по умолчанию для одного или нескольких параметров функции задаются с помощью того же синтаксиса, который употребляется при инициализации переменных. Например, функция для создания и инициализации двумерного массива, моделирующего экран терминала, может использовать такие значения для высоты, ширины и символа фона экрана:

```
    char *screenInit( int height = 24, int width = 80, char background =
' ' );
```

Функция, для которой задано значение параметра по умолчанию, может вызываться по-разному. Если аргумент опущен, используется значение по умолчанию, в противном случае – значение переданного аргумента. Все следующие вызовы screenInit() корректны:

```
char *cursor;
// эквивалентно screenInit(24, 80, ' ')
cursor = screenInit();
// эквивалентно screenInit(66, 80, ' ')
cursor = screenlnit(66);
// эквивалентно screenInit(66, 256, ' ')
cursor = screenlnit(66, 256);
```

Фактические аргументы сопоставляются с формальными параметрами позиционно (в порядке следования), и значения по умолчанию могут использоваться только для подстановки вместо отсутствующих последних аргументов. В нашем примере невозможно задать значение для background, не задавая его для height и width.

```
// эквивалентно screenInit('?', 80, ' ')
cursor = screenInit('?');
// ошибка, неэквивалентно screenInit(24, 80, '?')
cursor = screenInit( ,  , '?');
```

При разработке функции с параметрами по умолчанию придётся позаботиться об их расположении. Те, для которых значения по умолчанию вряд ли будут употребляться, необходимо поместить в начало списка.

Функции с переменным числом параметров. Иногда нельзя перечислить типы и количество всех возможных аргументов функции. В этих случаях список параметров представляется многоточием (...), которое отключает механизм проверки типов. Наличие многоточия говорит компилятору, что у функции может быть произвольное количество аргументов неизвестных заранее типов. Многоточие употребляется в двух форматах:

```
void foo( parm_list, ... );
void foo( ... );
```

Первый формат предоставляет объявления для части параметров. В этом случае проверка типов для объявленных параметров производится, а для оставшихся фактических аргументов – нет. Запятая после объявления известных параметров необязательна.

Примером вынужденного использования многоточия служит функция printf() стандартной библиотеки C. Её первый параметр является C-строкой:

```
int printf( const char* ... );
```

Это гарантирует, что при любом вызове printf() ей будет передан первый аргумент типа const char*. Содержание такой строки, называемой форматной, определяет, необходимы ли дополнительные аргументы при вызове. При наличии в строке формата метасимволов, начинающихся с символа %, функция ждёт присутствия этих аргументов. Например, вызов

```
printf( "hello, world\n" );
```

имеет один строковый аргумент. Но

```
printf( "hello, %s\n", userName );
```

имеет два аргумента. Символ % говорит о наличии второго аргумента, а буква s, следующая за ним, определяет его тип – в данном случае символьную строку.

Большинство функций с многоточием в объявлении получают информацию о типах и количестве фактических параметров по значению явно объявленного параметра. Следовательно, первый формат многоточия употребляется чаще.

3.6.3. ВОЗВРАТ ФУНКЦИЕЙ ЗНАЧЕНИЯ

В теле функции может встретиться инструкция return. Она завершает выполнение функции. После этого управление возвращается той функции, из которой была вызвана данная. Инструкция return может употребляться в двух формах:

```
return;
return expression;
```

Первая форма используется в функциях, для которых типом возвращаемого значения является void. Использовать return в таких случаях обязательно, если нужно принудительно завершить работу. После конечной инструкции функции подразумевается наличие return. Например:

```
void d_copy( double *src, double *dst, int sz )
{
/* копируем массив "src" в "dst"
для простоты предполагаем, что они одного размера*/
// завершение, если хотя бы один из указателей равен 0
if ( !src || !dst )
return;
// завершение,
// если указатели адресуют один и тот же массив
if ( src == dst )
return;
// копировать нечего
if ( sz == 0 )
return;
// все ещё не закончили?
// тогда самое время что-то сделать
for ( int ix = 0; ix < sz; ++ix )
dst[ix] = src[ix];
// явного завершения не требуется
}
```

Во второй форме инструкции return указывается то значение, которое функция должна вернуть. Это значение может быть сколь угодно сложным выражением, даже содержать вызов функции. В реализации функции factorial(), которую мы рассмотрим в следующем разделе, используется return следующего вида:

```
return val * factorial(val-1);
```

В функции, не объявленной с void в качестве типа возвращаемого значения, обязательно использовать вторую форму return, иначе произойдёт ошибка компиляции. Хотя компилятор не отвечает за правильность результата, он сможет гарантировать его наличие.

3.6.4. РЕКУРСИЯ

Функция, которая прямо или косвенно вызывает сама себя, называется рекурсивной. Например:

```
int rgcd( int vl, int v2 )
{
if ( v2 != 0 )
return rgcd( v2, vl%v2 );
return vl;
}
```

3.5. Трассировка вызова rgcd (15, 123)

vl	v2	return
15	123	rgcd(123, 15)
123	15	rgcd(15, 3)
15	3	rgcd(3, 0)
3	0	3

Такая функция обязательно должна определять условие окончания, в противном случае рекурсия будет продолжаться бесконечно. Подобную ошибку так иногда и называют – бесконечная рекурсия. Для rgcd() условием окончания является равенство нулю остатка.

Вызов rgcd(15, 123); возвращает 3 (см. табл. 3.5).

Рекурсивные функции обычно выполняются медленнее, чем их нерекурсивные (итеративные) аналоги. Это связано с затратами времени на вызов функции. Однако, как правило, они компактнее и понятнее.

Приведём пример. Факториалом числа n является произведение натуральных чисел от 1 до n. Так, факториал 5 равен 120: $1 \times 2 \times 3 \times \times 4 \times 5 = 120$.

Вычислять факториал удобно с помощью рекурсивной функции:

```
unsigned long factorial( int val ) {
if ( val > 1 )
return val * factorial( val – 1 );
return 1;
}
```

3.6.5. ФУНКЦИЯ MAIN(). РАЗБОР ПАРАМЕТРОВ КОМАНДНОЙ СТРОКИ

При запуске программы мы, как правило, передаём ей информацию в командной строке. Например, можно написать

```
prog -d -o ofile data0
```

Фактические параметры являются аргументами функции main() и могут быть получены из массива С-строк с именем argv; мы покажем, как их использовать.

Во всех предыдущих примерах определение main() содержало пустой список:

```
int main() { ... }
```

Развёрнутая сигнатура main() позволяет получить доступ к параметрам, которые были заданы пользователем в командной строке:

```
int main( int argc, char *argv[] ) {...}
```

argc содержит их количество, а argv – С-строки, представляющие собой отдельные значения (в командной строке они разделяются пробелами). Скажем, при запуске команды

```
prog -d -o ofile data0
```

argc получает значение 5, а argv включает следующие строки:

```
argv[ 0 ] = "prog";
argv[ 1 ] = "-d";
argv[ 2 ] = "-o";
argv[ 3 ] = "ofile";
argv[ 4 ] = "data0";
```

В argv[0] всегда входит имя команды (программы). Элементы с индексами от 1 до argc-1 служат параметрами.

3.6.6. УКАЗАТЕЛЬ НА ФУНКЦИЮ

Указатель на функцию содержит адрес в сегменте кода, по которому располагается исполняемый код функции, т.е. адрес, по которому передаётся управление при вызове функции. Указатели на функции используются для косвенного вызова функции (не через её имя, а через обращение к переменной, хранящей её адрес), а также для передачи имени функции в другую функцию в качестве параметра. Указатель функции имеет тип «указатель функции, возвращающей значение заданного типа и имеющей аргументы заданного типа»:

```
тип (*имя) ( список_типов_аргументов );
```

Например, объявление:

```
int (*fun) (double, double);
```

задаёт указатель с именем fun на функцию, возвращающую значение типа int и имеющую два аргумента типа double.

Передача имен функций в качестве параметров. Функцию можно вызвать через указатель на неё. Для этого объявляется указатель соответствующего типа и ему с помощью операции взятия адреса присваивается адрес функции:

```
void f ( int a ) { / * . . . * / } // определение функции
void (*pf)(int); // указатель на функцию
pf = &f; // указателю присваивается адрес функции
// (можно написать pf = f ;)
pf(10); // функция f вызывается через указатель pf
// (можно написать (*pf)(10) )
```

Для того чтобы сделать программу легко читаемой, при описании указателей на функции используют переименование типов (typedef). Можно объявлять массивы указателей на функции:

```
// Описание типа PF как указателя
// на функцию с одним параметром типа int;
typedef void (*PF)(int);
// Описание и инициализация массива указателей:
PF menu[] = {&new, &open, &save};
menu[1](10); // Вызов функции open
```

Здесь new, open и save – имена функций, которые должны быть объявлены ранее.

Указатели на функции передаются в подпрограмму таким же образом, как и параметры других типов:

```
#include <stdio.h>
typedef void (*PF)(int);
void f1(PF pf) // функция fl получает в качестве параметра указатель типа PF
{
pf(5); // вызов функции, переданной через указатель
}
void f( int i ){printf("%d",i);}
int main()
{
f1( f );
return 0;
}
```

Тип указателя и тип функции, которая вызывается посредством этого указателя, должны совпадать в точности.

4. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

Для объектно-ориентированного программирования (ООП) характерен особый концептуальный подход к разработке программ.

Объектная ориентированность в простейшем смысле означает представление программного обеспечения в виде дискретных объектов, содержащих в себе структуры данных и поведение. До появления объектно-ориентированного подхода структуры данных и поведение были связаны между собой очень слабо.

Наиболее важными инструментальными средствами ООП являются [3 – 5, 7, 9, 10, 12]:

– абстрагирование;
– инкапсуляция и сокрытие данных;
– полиморфизм;
– наследование;
– повторное использование программных кодов.

Индивидуальность означает, что данные делятся на дискретные сущности, хорошо отличимые друг от друга. Эти сущности называются объектами. Объекты с одинаковыми структурами данных (атрибутами) и поведением (операциями) группируются в классы. Класс – это абстракция, описывающая свойства, важные для конкретного приложения, и игнорирующая все остальные. Любой выбор классов произволен и зависит от приложения.

Каждый класс описывает множество индивидуальных объектов, которое может быть бесконечным. Каждый объект из этого множества называется экземпляром класса. Объект имеет свои собственные значения атрибутов, но названия атрибутов и операции являются общими для всех экземпляров класса.

Наследование – это наличие у разных классов, образующих иерархию, общих атрибутов и операций (составляющих). Суперкласс задаёт наиболее общую информацию, которую затем уточняют и улучшают его подклассы. Каждый подкласс соединяет в себе (наследует) все черты его суперкласса, к которым добавляет собственные уникальные черты. Подклассам не обязательно воспроизводить все черты суперкласса. Возможность выделять общие черты нескольких классов в суперкласс значительно сокращает количество повторений в проектах и программах и является одним из основных достоинств объектно-ориентированной технологии.

Полиморфизм означает, что одна и та же операция может подразумевать разное поведение в разных классах. Операция – это процедура или трансформация, которую объект выполняет сам или которая осуществляется с данным объектом. Реализация операции в конкретном классе называется методом. Поскольку объектно-ориентированная операция является полиморфной, в разных классах объектов она может быть реализована разными методами.

Абстракция означает сосредоточение на важнейших аспектах приложения и игнорирование всех остальных. Сначала принимается решение о том, что представляет собой объект и что он делает, а уже затем подбирается способ его реализации. Использование абстракций позволяет сохранить свободу принятия решений как можно дольше благодаря тому, что детали не фиксируются раньше времени.

Инкапсуляция или, иначе говоря, сокрытие информации состоит в отделении внешних аспектов объекта, доступных другим объектам, от деталей внутренней реализации, которые от других объектов скрываются. Инкапсуляция исключает возникновение взаимозависимостей участков программы, из-за которых небольшие изменения приводят к значительным непредвиденным последствиям. Реализация объекта может быть изменена безо всяких последствий для использующих его приложений. Изменение реализации может быть предпринято для повышения производительности, устранения ошибки, консолидации кода или для подготовки к переносу программы на другие системы.

4.1. ОПИСАНИЕ КЛАССОВ И ОБЪЕКТОВ

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются атрибутами или полями (по аналогии с полями структуры), а функции класса – методами. Поля и методы называются элементами класса. Описание класса в первом приближении выглядит так:

```
class <имя>
{
[ private: ]
<описание скрытых элементов>
public:
<описание доступных элементов>
} ; / / Описание заканчивается точкой с запятой
```

Спецификаторы доступа private и public управляют видимостью элементов класса.

Элементы, описанные после служебного слова private, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Интерфейс класса описывается после спецификатора public. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций private и public, порядок их следования значения не имеет.

Поля класса:

– могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);

– могут быть описаны с модификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;

– могут быть описаны с модификатором static, но не как auto, extern и register.

Инициализация полей при описании не допускается.

Классы могут быть глобальными (объявленными вне любого блока) и локальными (объявленными внутри блока, например, функции или другого класса).

Ниже перечислены некоторые особенности локального класса:

– внутри локального класса можно использовать типы, статические (static) и внешние (extern) переменные, внешние функции и элементы перечислений из области, в которой он описан; запрещается использовать автоматические переменные из этой области;

– локальный класс не может иметь статических элементов;

– методы этого класса могут быть описаны только внутри класса;

– если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

В качестве примера рассмотрим класс, который представляет пакет акций. Представим текущие вклады субъекта в виде конкретного пакета акций, приняв его за базовую единицу. Предусмотрим ведение записей о таких событиях, как исходная покупная цена и дата покупки, для отчётности перед налоговыми органами. Рассмотрим следующий перечень операций, которые нам предстоит выполнить: получить пакет в компании; приобрести дополнительные акции того же пакета; продать пакет; корректировать среднюю стоимость одной акции пакета; отображать данные о пакете акций.

Используем этот список в целях определения общедоступного интерфейса для класса пакета акций. Для поддержки этого интерфейса нам нужно хранить некоторые виды информации: наименование ком-

пании; число акций в пакете; цена каждой акции; общая стоимость всех акций пакета.

В программе, представленной ниже, приводится объявление класса Stock.

```
#include <iostream>
#include <stdlib.h>
#include <string.h>
using namespace std;

class Stock
{
private:
char company[30];
int shares;
double share_val;
double total_val;
void set_tot()
{
total_val = shares * share_val;
}
public:
void acquire (const char * co, int n, double pr);
void buy(int num, double price);
void sell(int num, double price);
void update(double price);
void show();
};

void Stock::acquire(const char * co, int n, double pr)
{
strncpy(company, co, 29) ; //при необходимости выполняется усечение строки co
company[29] = '\0';
shares = n;
share_val = pr;
set_tot() ;
}

void Stock::buy(int num, double price)
{
shares += num; share_val = price; set_tot();
}
```

```
void Stock::sell(int num, double price)
{
if (num > shares)
{
cerr<<"Вы не можете продать больше чем имеете!\n";
exit(1);
}
shares -= num;
share_val = price;
set_tot();
}

void Stock::update(double   price)
{
share_val = price;
set_tot();
}

void   Stock::show()
{
cout<<"Компания:"<<company<<"Число акций:"<<shares
<<'\n'<<"Цена акции: $"<<share_val
<<"Общая стоимость: $"<<total_val<<'\n';
}

void main()
{
Stock Ivanov;
Ivanov.acquire("NanoSmart", 20, 12.50);
Ivanov.show() ;
Ivanov.buy(15, 18.25);
Ivanov.show();
}
```

Отметим, что ключевое слово class в языке C++ идентифицирует этот программный код как код, определяющий конструкцию класса. Синтаксис отождествляет Stock с именем типа этого нового класса. Это объявление позволяет нам объявлять переменные, именуемые объектами или экземплярами типа Stock. Каждый отдельный объект представляет собой отдельный вклад. Например, объявления

```
Stock   Ivanov;
Stock   Petrov;
```

создают два объекта типа Stock с именами Ivanov и Petrov. Объект Ivanov, например, может представлять вклады Иванова.

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция **.** (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель.

Обратиться таким образом можно только к элементам со спецификаторам public. Получить или изменить значения элементов со спецификатором private можно только через обращение к соответствующим методам.

Информация, которую мы решили сохранить, поступает в форме элементов данных класса, таких как company и shares. Элемент данных company объекта Ivanov, например, содержит имя компании, элемент данных share – число акций, которым владеет Ivanov, элемент share_val – стоимость каждой акции, а элемент total_val – значение общей стоимости всех акций. То же самое можно сказать и о тех операциях, которые по нашему замыслу должны быть представлены в виде функций-элементов, таких как sell() и update(). Функции-элементы класса получили название методов класса. Вместо их может быть объявлена функция-элемент, такая как, например, set_tot(), либо она может быть представлена прототипом, как и остальные функции этого класса. Полное определение других функций-элементов приводится далее, однако чтобы описать интерфейсы функций при описании класса, достаточно их прототипов.

4.2. КОНСТРУКТОР КЛАССА

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. В объявлении конструктора его имя совпадает с именем класса.

Перечислим основные свойства конструкторов:

– Конструктор не возвращает значение, даже типа void. Нельзя получить указатель на конструктор.

– Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).

– Конструктор, вызываемый без параметров, называется конструктором по умолчанию.

– Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

– Если программист не указал ни одного конструктора, компилятор создаёт его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.

– Конструкторы не наследуются.

– Конструкторы нельзя описывать с модификаторами const. virtual и static.

– Конструкторы глобальных объектов вызываются до вызова функции main.

– Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

– Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

```
// Список параметров не должен быть пустым
имя_класса имя_объекта [(список параметров)];

// Создаётся объект без имени (список может быть пустым)
имя_класса (список параметров);

// Создаётся объект без имени и копируется
имя_класса имя_объекта = выражение;
```

Создадим конструктор Stock. Поскольку предусмотрено, что объект класса Stock получает три значения из внешнего мира, необходимо назначать этому конструктору три аргумента. Вполне возможно, что вы намерены всего лишь задать значение для элемента company, а другим элементам присвоить нулевое значение. Это можно сделать с помощью аргументов, заданных по умолчанию. Следовательно, прототип будет иметь следующий вид:

```
// прототип конструктора с некоторыми
// аргументами, заданными по умолчанию
Stock (const char * co, int n = 0, double pr = 0.0);
```

Первый аргумент является указателем на строку, которая используется для инициализации элементов класса символьного массива company. Аргументы n и pr передают значения элементам shares и share_val. Обратите внимание на то, что возвращаемый тип отсутствует. Прототип находится в общедоступном разделе объявления класса.

Ниже представлено возможное определение конструктора:

```
// определение конструктора
Stock::Stock(const char * co, int n, double pr)
{
strncpy(company, co, 29);
company[29] = ' \0 ' ;
shares = n ;
share_val = pr;
set_tot() ;
}
```

Это тот же программный код, который мы использовали в функции acquire(). Различие заключается в том, что программа автоматически вызывает конструктор в тот момент, когда она объявляет объект.

В C++ предусмотрены два способа инициализации объекта с использованием конструктора. Первый из них – это явный вызов конструктора:

```
Stock food = Stock("World Cabbage", 250, 1.25);
```

С помощью этой команды элементу company объекта food присваивается строка "World Cabbage", элементу shares – значение 250 и т.д.

Второй способ предусматривает неявный вызов конструктора:

```
Stock   garment("Doir",    50,    2.5);
```

Более компактная форма вызова эквивалентна следующему явному обращению:

```
Stock  garment = Stock("Doir", 50, 2.5) ;
```

C++ использует конструктор класса всякий раз, когда создаётся объект этого класса, даже если вы используете спецификатор new в целях динамического распределения памяти. Конструктор со спецификатором new используется следующим образом:

```
Stock *pstock = new Stock("Electroshock Games",  18,    19.0);
```

Этот оператор создаёт объект класса Stock, инициализирует его значениями, переданными с помощью аргументов, и присваивает адрес объекта указателю pstock. В этом случае у объекта нет имени, однако можно воспользоваться указателем при работе с объектом.

Конструктором, заданным по умолчанию, является конструктор, используемый для построения объекта, когда явные значения для инициализации отсутствуют. Другими словами, конструктор используется для объявлений такого рода:

```
Stock stock1; // используется конструктор, заданный по умолчанию
```

Если нет никаких конструкторов, C++ автоматически использует конструкторы, заданные по умолчанию. Таким конструктором является версия по умолчанию конструктора, заданного по умолчанию, и она не выполняет никаких действий. Для класса Stock он будет выглядеть следующим образом:

```
Stock :: Stock() {       }
```

Окончательный результат состоит в том, что объект stock1 создаётся без инициализации его элементов, точно так же как оператор

```
int   x;
```

создаёт переменную x, не присваивая ей значения. То обстоятельство, что конструктор, заданный по умолчанию, не имеет аргументов, отражает тот факт, что в объявлении не появляются никакие значения.

Отметим, что компилятор предоставляет конструктор, заданный по умолчанию, только в том случае, если вы не определите никакого конструктора. После того как вы назначите конкретный конструктор конкретному классу, обязанность по предоставлению конструктора, заданного по умолчанию, переходит от компилятора к вам. Если вы воспользуетесь конструктором, который не используется по умолчанию, таким как, например,

```
Stock(const char * co, int n, double pr);
```

и не предложите своей собственной версии конструктора, заданного по умолчанию, то объявление вида

```
Stock stock1; // невозможно с текущим конструктором
```

вызовет ошибку. Причина такого поведения заключается в том, что у вас может возникнуть необходимость сделать невозможным создание неинициализированных объектов. С другой стороны, вам может понадобиться создавать объекты без явной инициализации. В таком случае придется определить собственный конструктор.

Это должен быть конструктор, которому не нужны аргументы. Вы можете определить конструктор по умолчанию двумя способами. Один из них состоит в том, чтобы присвоить значения, заданные по умолчанию, всем аргументам существующего конструктора:

```
Stock(const char * co = "Error", int n =  0, double pr = 0.0);
```

Второй способ предусматривает использование перегрузки функции для определения второго конструктора, который при этом не имеет аргументов:

```
Stock () ;
```

На практике обычно требуется инициализировать объекты, чтобы быть уверенным, что все элементы начинаются с известных, корректно выбранных значений. Таким образом, конструктор, заданный по умолчанию, как правило, осуществляет неявную инициализацию значений всех элементов. В данном случае, например, можно определить конструктор для класса Stock следующим образом:

```
Stock: : Stock ()
{
strcpy(company, "no name");
shares = 0;
share_yal = 0.0;
total_val = 0.0;
}
```

После того как вы воспользовались одним из методов (не указывая аргументов или значений, заданных по умолчанию для всех аргументов), чтобы создать конструктор по умолчанию, можно объявить переменные объекта, не выполняя их явной инициализации:

```
Stock first; // неявно вызывает конструктор
// заданный по умолчанию
Stock first = Stock(); // вызывает его явно
Stock *prelief = new Stock; // вызывает его неявно
```

Тем не менее, не дайте ввести себя в заблуждение неявной формой конструктора, не используемого по умолчанию:

```
Stock first("Concrete Conglomerate"); // вызывает конструктор
Stock second () ; // объявляет функцию
Stock third; // вызывает конструктор, заданный по умолчанию
```

Первое объявление вызывает конструктор, который не является конструктором, заданным по умолчанию, т.е. конструктор, который принимает аргументы. Второе объявление утверждает, что second() – функция, которая возвращает объект Stock. При неявном вызове конструктора, заданного по умолчанию, не употребляйте круглые скобки.

Конструктор копирования – это специальный вид конструктора, получающий в качестве единственного параметра указатель на объект этого же класса.

```
Stock:: Stock (const Stock &) { ... / * Тело конструктора */ }
```

Этот конструктор вызывается в тех случаях, когда новый объект создаётся путём копирования существующего:

– при описании нового объекта с инициализацией другим объектом;

– при передаче объекта в функцию по значению;

– при возврате объекта из функции.

Если программист не указал ни одного конструктора копирования, компилятор создаёт его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

4.3. ДЕСТРУКТОР КЛАССА

Деструктор – это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. Деструктор вызывается автоматически, когда объект выходит из области видимости:

– для локальных объектов – при выходе из блока, в котором они объявлены;

– для глобальных – как часть процедуры выхода из main;

– для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции delete.

При выходе из области действия указателя на объект автоматический вызов деструктора объекта не производится.

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса. Деструктор:

– не имеет аргументов и возвращаемого значения;
– не может быть объявлен как const или static;
– не наследуется;
– может быть виртуальным.

Если деструктор явным образом не определён, компилятор автоматически создаёт пустой деструктор.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически – иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор для рассматриваемого примера выглядит так:

```
Stock :: ~Stock()
{
/* Тело деструктора */
}
```

Деструктор можно вызвать явным образом путём указания полностью уточнённого имени, например:

```
Stock *st;
st -> ~Stock();
```

Это может понадобиться для объектов, которым с помощью перегруженной операции new выделялся конкретный адрес памяти. Без необходимости явно вызывать деструктор объекта не рекомендуется.

Когда следует обращаться к деструктору? Это решение принимает компилятор, ваш программный код не должен содержать явных обращений к деструктору. Если создаётся объект класса статической памяти, то его деструктор вызывается автоматически в момент окончания выполнения программы. Если создаётся объект класса автоматической памяти, то его деструктор вызывается автоматически, когда программа выходит из блока программного кода, в котором объект был определён. Если объект создаётся с использованием спецификатора new, он размещается в динамически распределяемой области памяти или в свободной памяти, а его деструктор вызывается автоматически, когда используется оператор delete для освобождения памяти. И наконец, программа может создавать временные объекты для того, чтобы выполнять определённые операции; в этом случае программа

автоматически вызывает деструктор, чтобы тот удалил объект, когда программа прекращает использование этого объекта.

Поскольку деструктор вызывается автоматически, когда объект класса прекращает функционировать, деструктор должен быть наготове. Если вы не позаботились о своем деструкторе, компилятор предоставит вам деструктор, заданный по умолчанию, который не выполняет никаких действий.

Усовершенствуем класс Stock. Организуем программу в виде нескольких отдельных файлов. Поместим описание рассматриваемого класса в заголовочный файл с именем stockl.h.

```
// stockl.h
#ifndef _STOCKl_H_
#define _STOCKl_H_
class Stock {
private:
char company[30];
int shares;
double share_val;
double total_val;
void set_tot()
{ total_val = shares * share_val; }
public:
Stock(); // конструктор по умолчанию
Stock(const char * co, int n = 0, double pr = 0.0);
~Stock(); // деструктор по умолчанию
void buy(int num, double price);
void sell(int num, double price);
void update(double price);
void show() ;
};
#endif
```

Методы класса помещаются в файл с именем stockl.cpp. В общем, заголовочный файл, содержащий объявление класса, и файл исходного программного кода, содержащий определения методов, должны иметь одно и то же базовое имя, чтобы можно было отслеживать, какие файлы принадлежат друг другу. Использование отдельных файлов для объявления класса и для функций-элементов отделяет абстрактное определение интерфейса (объявление класса) от деталей реализации (определения функций-элементов).

```
// stockl.cpp – методы класса Stock
#include <iostream>
#include <cstdlib> //или stdlib.h для exit()
#include <cstring> //или string.h для strncpy()
using namespace std;
#include "stockl.h"


// конструкторы
Stock::Stock() // конструктор по умолчанию
{
strcpy(company, "noname");
shares = 0;
share_val = 0.0;
total_val = 0.0;
}
Stock: : Stock (const char * co, int n, double pr)
{
strncpy(company, co, 29);
company[29] = ' \0';
shares = n ;
share_val = pr;
set_tot() ;
}


// деструктор класса
Stock::~Stock() // деструктор класса
{
cout << "Bye, " << company << "! \n" ;
}


// другие методы
void Stock::buy(int num, double price)
{
shares += num;
share_val = price;
set_tot() ;
}
```

```
void Stock::sell(int num, double price)
{
if (num > shares)
{
cerr « "Я не могу продать больше чем Вы имеете \n" ;
exit (1) ;
}
shares -= num;
share_val = price;
set_tot() ;
}

void Stock::update(double price)
{
share_val = price;
set_tot() ;
}

void Stock:: show ()
{
cout << "Company: " << company << "Shares: " << shares << '\n'
<< "Share Price: $" << share_val << "Total Worth: $" << total_val << ' \n' ;
}
```

И наконец, используя эти ресурсы, поместим программу в третий файл, который называется usestokl.cpp.

```
#include <iostream>
using namespace std;
#include "stockl.h"
int main()
{
// использование конструкторов для
// построения новых объектов
Stock stockl("NanoSmart", 12, 20.0); // синтаксис 1
Stock stock2 = Stock ("Boffo Objects", 2, 2.0); // синтаксис 2
cout.precision(2);      // формат #.##
cout.setf(ios_base::fixed, ios_base::floatfield);  // формат #.##
cout.setf(ios_base::showpoint); //формат # . ##
stockl.show();
stock2.show();
stock2 = stockl;        // назначение объекта
```

```
    // использование конструктора для
    // переустановки объекта
    stockl = Stock("Nifty Foods", 10, 50.0);

    cout « "After stock reshuffle:\n" ;
    stockl.show();
    stock2 . show () ;
    return 0;
    }
```

Оператор

```
    Stock stockl("RosNano", 12, 20.0);
```

создаёт объект класса Stock под именем stockl и инициализирует его элементы данных заданными значениями.

Оператор

```
    Stock stock2 = Stock("Gazprom" , 2 , 2.0);
```

использует вторую разновидность синтаксиса для построения и инициализации объекта с именем stock2.

Можно использовать конструктор не только для инициализации нового объекта. Например, в составе функции main() имеется оператор

```
    stockl = Stock("Lukoil", 10, 50.0);
```

Объект stockl уже существует. Таким образом, вместо того чтобы инициализировать объект stockl, этот оператор присваивает новые значения данному объекту. Он выполняет это, вынуждая конструктор построить новый, временный объект и в дальнейшем осуществить копирование содержимого этого нового объекта в объект stock1.

Оператор

```
    stock2 = stockl; // присвоение объекта
```

показывает, что вы можете присвоить один объект другому объекту того же типа. Как и в случае присваивания структур, в процессе присваивания объекта типа класс по умолчанию копируются элементы одного объекта в элементы другого. В этом случае первоначальное содержимое объекта stock2 затирается.

Однако следует помнить, что когда вы присваиваете один объект другому объекту того же класса, C++ по умолчанию копирует содержимое каждого элемента исходного объекта в соответствующий элемент данных целевого объекта.

Следует отметить принципиальное различие между двумя приводимыми ниже операторами:

```
Stock stock2 = Stock("Boffo Objects",2, 2.0);
stockl = Stock("Nifty Foods", 10, 50.0);
```

Первый из представленных операторов выполняет инициализацию; он создаёт объект с заданным значением. Второй оператор является оператором присваивания. Он создаёт временный объект и затем копирует его в существующий. Этот оператор менее эффективен, чем оператор инициализации.

4.4. УКАЗАТЕЛЬ THIS

Каждая функция-элемент, включая конструкторы и деструкторы, имеет указатель this. Характерной особенностью указателя this является то, что он указывает на вызывающий объект. Если у какого-либо метода возникает необходимость ссылки на объект как на единое целое, он может воспользоваться выражением *this. Спецификатор const, помещённый сразу за скобками, в которые заключены аргументы, истолковывает в рассматриваемом случае указатель this как const; в такой ситуации вы не можете использовать this, чтобы изменить значение объекта.

Продолжим работу с классом Stock. До сих пор каждая функция-элемент этого класса имела дело с одним-единственным объектом. Этим объектом был объект, который обращался к ней. Однако иногда возникает потребность в методе, который работает с двумя объектами, и эта задача решается с использованием специального указателя this. Рассмотрим, при каких обстоятельствах может возникнуть такая потребность.

Несмотря на то, что объявление класса Stock отображает данные, ему не хватает аналитических возможностей. Например, проанализировав выходные данные функции show(), вы сможете сказать, какой из ваших вкладов превосходит по величине все остальные. Однако программа этого сделать не может, поскольку у неё нет непосредственного доступа к значению total_val. Самый прямой способ, позволяющий программе иметь сведения о хранимых данных, – предоставить в её распоряжение методы, возвращающие значения. Как правило, для этой цели используется встроенный программный код:

```
class Stock
{
private:
double total_val;
public:
double total () const
{ return total_val; }
};
```

По существу, это определение превращает total_val в область памяти только для чтения, если дело касается прямого доступа программы к данным.

Добавив эту функцию в объявление класса, вы можете предоставить программе возможность проверить ряд вкладов, чтобы найти тот из них, который превосходит все остальные по стоимости.

Однако воспользуемся другим подходом. Этот подход состоит в том, что определяется функция-элемент, которая просматривает два объекта Stock и возвращает ссылку на тот из них, который больше по величине. При попытке реализовать этот подход возникает целый ряд интересных вопросов.

Во-первых, как будет выглядеть функция-элемент, выполняющая сравнение двух объектов? Предположим, например, что вы решили назвать такой метод topval(). Далее, при вызове функции stockl.topval() обеспечивается доступ к данным объекта stockl, в то время как сообщение stock2.topval() получает доступ к данным объекта stock2. Если вы хотите, чтобы указанный выше метод провёл сравнение двух объектов, то необходимо передать второй объект как аргумент. В целях повышения производительности передайте этот аргумент по ссылке. Иначе говоря, пусть метод topval() использует аргумент типа const Stock &.

Во-вторых, как вы возвратите ответ этого метода в вызывающую программу? Наиболее простой способ – заставить метод возвратить ссылку на объект, который имеет наибольшую общую стоимость. Таким образом, метод, выполняющий сравнение, должен иметь следующий прототип:

```
const Stock & topval(const Stock & s) const;
```

Эта функция осуществляет неявный доступ к одному из объектов, к другому объекту она получает прямой доступ, при этом она возвращает ссылку на один из двух объектов. Ключевое слово const в круглых скобках означает, что функция не вносит изменений в объект, к

которому она получает прямой доступ, а ключевое слово const, которое следует сразу за скобками, означает, что функция не подвергает модификации объект, к которому она осуществляет неявный доступ. Поскольку функция возвращает ссылку на один из объектов const, возвращаемый тип также имеет ссылку const.

Предположим далее, что вы хотите сравнить объекты stockl и stock2 класса Stock и присвоить тот из них, который имеет большую общую стоимость, объекту top. Для этого можно воспользоваться любым из следующих двух операторов:

```
top = stockl.topval(stock2);
top = stock2.topval(stockl);
```

Первый из них получает прямой доступ к объекту stockl и неявный доступ к объекту stock2, в то время как второй получает прямой доступ к объекту stockl и неявный доступ к объекту stock2.

Каким бы ни был доступ, этот метод сравнивает два объекта и возвращает ссылку на тот из них, у которого общая стоимость больше.

По существу, такой способ записи может привести к путанице. Было бы намного проще, если бы вы каким-то образом могли использовать оператор сравнения > для сравнения двух объектов. Это можно сделать с помощью перегрузки операторов.

Между тем, мы ещё не всё выяснили из того, как реализовать функцию topval(). При этом возникает небольшая проблема. Ниже представлена частичная реализация, которая позволяет нам вникнуть в суть этой проблемы.

```
const Stock & Stock::topval(const Stock & s) const
{
if (s.total_val > total_val)
return s; // аргумент типа объект
else return ?????; // объект, вызывающий метод topval
}
```

Здесь s.total_val – это общее значение для объекта, переданного в качестве аргумента, а total_val – общее значение для объекта, которому было отправлено сообщение. Если значение s.total_val больше значения total_val, то функция возвращает s. В противном случае она возвращает объект, использованный для вызова метода. Проблема заключается в том, как назвать этот объект? Если вы присвоите ему имя stockl.topval(stock2), то s будет ссылкой на stock2 (т.е. псевдоним для stock2), однако псевдонима у stockl нет.

В C++ эта проблема решается путём использования специального указателя this. Указатель this ссылается на объект, который используется для вызова функции-элемента. (По существу, указатель this передаётся как скрытый аргумент рассматриваемого метода.) Следовательно, при вызове функции stockl.topval(stock2) устанавливается указатель this на адрес объекта stockl и обеспечивается доступ к этому указателю со стороны метода topval(). Аналогично при вызове функции stock2.topval(stockl) устанавливается указатель this на адрес объекта stock2. В общем случае для всех методов класса указатель this указывает на адрес объекта, который вызывает этот метод. И в самом деле, total_val в методе topval() – всего лишь сокращённое обозначение от this->total_val.

Напомним, что в результате применения оператора разыменования * к указателю получается величина, на которую указывает указатель. Теперь вы можете завершить определение метода, используя *this как псевдоним вызывающего объекта.

```
const Stock & Stock::topval(const Stock & s) const
{
if (s.total_val > total_val)
return s; // объект в качестве аргумента
else return *this; // вызывающий объект
}
```

Тот факт, что возвращаемый тип является ссылкой, означает, что возвращаемый объект сам является вызывающим объектом, и ни в коем случае не означает, что механизм возврата передаёт копию.

4.5. ПЕРЕГРУЗКА ОПЕРАЦИЙ

Перегрузка операций представляет собой ещё один пример полиморфизма C++. Вы уже видели, как C++ обеспечивает возможность определять несколько функций, имеющих одно и то же имя при условии, что у них разные сигнатуры (списки аргументов). Это и есть перегрузка функций, или функциональный полиморфизм. Он предназначен для того, чтобы предоставить вам возможность использовать одно и то же имя функции для выполнения одних и тех же базовых операций, даже если вы применяете такую операцию по отношению к различным типам данных. Перегрузка операций (операторов) позволяет вам рассматривать операции (операторы) C++ сразу в нескольких смыслах. Фактически многие операции языка C++ уже перегружены изначально. Например, операция *, будучи применённой к адресу, даёт значение,

которое хранится по этому адресу. Однако, применяя операцию * к паре чисел, мы получаем произведение этих значений. C++ использует число и типы операндов, чтобы решить, какое действие предпринять.

C++ позволяет распространить перегрузку операций (операторов) на типы, определённые пользователем. Благодаря этому появляется возможность, например, использовать символ + для сложения двух объектов. Опять-таки, компилятор воспользуется числом и типом операндов, чтобы определить, каким определением операции сложения воспользоваться. Перегруженные операции часто делают вид программного кода более привычным.

Чтобы перегрузить какую-либо операцию, лучше воспользоваться функцией специальной формы, получившей название операторной. Операторная функция имеет вид:

```
operatorop(список_аргументов)
```

где op – это символ операции, подвергаемой перегрузке. Другими словами, operator+() перегружает операцию + (здесь op – это +), а operator*() перегружает операцию * (здесь op – это *). Операцией op может быть только полноценная операция C++; для этого недостаточно ввести новое обозначение и этим ограничиться. Например, вы не можете пользоваться функцией operator@(), поскольку в C++ нет операции @. Но функция operator[]() перегрузит операцию [], так как [] – это операция, выполняющая индексацию массива. Предположим, у вас имеется класс Salesperson, для которого даётся определение функции-элемента operator+(), предназначенной для выполнения перегрузки операции +, чтобы она могла выполнять сложение цифр, отражающих данные о продаже одного объекта класса Salesperson (продавец) с цифрами другого объекта этого же класса. Далее, если district2, sid и sara – это объекты класса Salesperson, вы можете записать такое уравнение:

```
district2 = sid + sara;
```

Компилятор, убедившись в том, что оба операнда принадлежат одному и тому же классу Salesperson, заменят эту операцию соответствующей операторной функцией:

```
district2 = sid.operator+(sara) ;
```

Эта функция использует объект sid неявно (поскольку она вызывает метод) и объект sara явно (поскольку он передаётся как аргумент) для вычисления суммы, которую она возвращает. Разумеется, наибо-

лее привлекательной стороной является то, что вы можете использовать привычную запись операции + вместо неуклюжей формы записи функции.

Например, Вы работали с отчётом в течение 2 часов 35 минут до обеда и в течение 2 часов 40 минут после обеда, то возникает вопрос, как долго вы работали над отчётом в течение всего дня? Это один из примеров, в котором понятие сложения имеет смысл, однако величины, которые вы складываете (сочетание часов и минут) не соответствуют встроенному типу данных. Можно решить задачу путём объявления структуры time и функции sum(), выполняющей сложение подобных структур. Сейчас у нас появилась возможность реализовать это в виде класса Time, выполняя для этой цели сложение. Начнём с обычного метода, а затем посмотрим, как вместо него использовать операцию перегрузки.

```
Программа mytime0.h
// mytime0.h – класс Time перед выполнением
// перегрузки операции
#ifndef _MYTIME0_H_
#define _MYTIME0_H_
#include <iostream>
using namespace std;
class Time
{
private:
int hours;
int minutes;
public:
Time();
Time(int h, int m = 0) ;
void AddMin(int m);
void AddHr(int h);
void Reset(int h = 0, int m = 0) ;
Time Sum (const Time & t) const;
void Show() const;
};
#endif
```

В рассматриваемом классе имеются методы для преобразования и возврата в исходное положение показаний времени, для отображения временных значений и для сложения двух таких значений. В листинге представлены определения этих методов.

```
//mytime0.cpp – реализация методов класса Time
#include "mytime0.h"
Time::Time()
{
hours = minutes = 0;
}
Time: :Time (int h, int m )
{
hours = h ; minutes = m;
}
void Time::AddMin(int m)
{
minutes += m;
hours += minutes / 60;
minutes %= 60;
}
void Time::AddHr(int h)
{
hours += h;
}
void Time ::Reset (int h, int m)
{
hours = h;
minutes = m;
}
Time Time::Sum(const Time & t) const
{
Time sum;
sum.minutes = minutes + t.minutes;
sum.hours = hours + t.hours + sum.minutes / 60;
sum.minutes %= 60;
return sum;
}
void Time : : Show () const
{
cout << hours << " hours, " << minutes << " minutes"<< ' \n' ;
}
```

Рассмотрим программный код функции Sum(). Обратите внимание на тот факт, что аргумент представляет собой ссылку, однако возвращаемый тип не является ссылкой. Причина выбора ссылки в каче-

стве аргумента заключается в достижении более высокой эффективности. Программный код должен обеспечить получение такого же результата, как если бы объект Time был передан по значению. Однако в большинстве случаев передача по ссылке обеспечивает более высокое быстродействие и более эффективное использование памяти, чем при передаче по значению.

Возвращаемое значение, однако, не может быть ссылкой. Это объясняется тем, что данная функция создаёт новый объект (sum) класса Time, представляющий собой сумму двух других объектов Time. Возврат объекта, как это имеет место в рассматриваемом программном коде, приводит к созданию копии объекта, которой может воспользоваться вызывающая функция. Однако, если возвращаемым типом будет Time &, ссылка будет указывать на объект sum. Но объект sum – это локальная переменная, которая уничтожается, как только функция перестаёт существовать, следовательно, эта ссылка будет относиться к несуществующему объекту. Вместе с тем использование возвращаемого типа Time означает, что программа создаёт копию объекта sum, прежде чем уничтожить его, и вызывающая функция получит эту копию.

И наконец, следующий программный код позволяет выполнить проверку того фрагмента класса, который выполняет суммирование.

```
// usetime0.cpp – использует первый вариант
// класса Time
// компилирует файлы usetime0.cpp и
// mytime0.cpp в один программный модуль
#include <iostream>
#include "mytime0.h"
using namespace std;
int main()
{
Time A;
Time B(5, 40);
Time C (2, 55);
A.Show() ;
B.Show();
C.Show();
A = B.Sum (C) ;
A.Show();
return 0;
}
```

Результаты выполнения программы:

0 hours, 0 minutes;
5 hours, 40 minutes;
2 hours, 55 minutes;
8 hours, 35 minutes.

Достаточно просто преобразовать класс Time таким образом, чтобы можно было пользоваться перегруженной операцией сложения. Для этого достаточно поменять имя Sum() на имя operator+().

```
// mytime1.h – класс Time после выполнения перегрузки операции
#ifndef _MYTIMS1_H_
#define _MYTIMS1_H_
#include <iostream>
using namespace std;
class Time
{
private:
int hours;
int minutes;
public:
Time();
Time(int h, int m = 0) ;
void AddMin(int m);
void AddHr(int h);
void Reset (int h = 0, int m = 0) ;
Time operator+(const Time S t) const;
void Show () const;
};
#endif

// mytimel.cpp – реализация методов класса Time
#include "mytime1.h"
Time::Time()
{
hours = minutes = 0;
}
Time: :Time (int h, int m )
{
hours = h ;
minutes = m;
}
```

```
void Time::AddMin(int m)
{
minutes += m;
hours += minutes / 60;
minutes %= 60;
}
void Time::AddHr(int h)
{
hours += h ;
}
void Time::Reset (int h, int m)
{
hours = h;
minutes = m;
}
Time Time: :operator+(const Time & t) const
{
Time sum;
sum.minutes = minutes + t.minutes;
sum.hours = hours + t.hours + sum.minutes / 60;
sum.minutes %= 60;
return sum;
}
void Time::Show() const
{
cout << hours << " hours, " << minutes << " minutes";
cout << ' \n ' ;
}
```

Подобно функции Sum(), operator+() вызывается объектом класса Time, принимает второй объект Time в качестве аргумента и возвращает объект Time. Таким образом, можно вызывать метод operator+(), используя для этой цели тот же синтаксис, который применялся функцией Sum():

```
A = B.operator+(C); // обозначение функции
```

Присваивая этому методу имя operator+(), вы получаете также возможность пользоваться операторной формой записи:

```
A = B + C; // операторная форма записи
```

Этот выражение обеспечивает вызов метода operator+(). Обратите внимание на тот факт, что объект в левой части операции (в рассматриваемом случае это B) является вызывающим, а объект в правой части (в рассматриваемом случае это C) – это объект, переданный в качестве аргумента. Следующий листинг иллюстрирует этот момент.

```
// usetimel.cpp – использует второй вариант
// класса Time
// компилирует файлы usetimel.cpp и
// mytimel.cpp в один программный модуль

#include <iostream>
#include "mytimel.h"
int main()
{
Time A;
Time B (5, 40);
Time C (2, 55);
A.Show() ;
B.Show();
C.Show();
A = B.operators-(C) ; // функциональная форма записи
A.Show () ;
B = A + C; // операторная форма записи
A.Show();
return 0;
}
```

Результаты выполнения программы:

0 hours, 0 minutes;
5 hours, 40 minutes;
2 hours, 55 minutes;
8 hours, 35 minutes;
11 hours, 30 minutes.

Большая часть операций C++ может быть перегружена таким же образом. Перегруженные операции (за некоторым исключением) не обязательно должны быть функциями-элементами. Однако, по меньшей мере, один из операндов должен иметь тип, определённый пользователем. Рассмотрим ограничения, которые C++ накладывает на перегрузку операций, определённых пользователем, несколько более подробно:

1. Перегруженная операция должна иметь, по меньшей мере, один операнд, имеющий тип, определённый пользователем. Это не позволит вам осуществлять перегрузку операций для стандартных типов данных. Следовательно, вы не сможете переопределить операцию вычитания (–) так, чтобы вместо разности двух значений типа double вычислялась их сумма. Это ограничение способствует обеспечению надёжной работы программы, но в то же время может повредить творческому подходу к использованию системных ресурсов.

2. Вы не можете использовать операцию таким образом, чтобы она нарушала правила синтаксиса, которым подчиняется исходная операция. Например, невозможно перегрузить операцию деления по модулю (%) так, чтобы её можно было использовать с одним операндом.

Аналогично вы не можете изменить приоритеты операций. Следовательно, если перегружается операция сложения в целях получения возможности складывать два класса, новая операция получает тот же приоритет, что и обычное сложение.

3. Невозможно создавать новые символы операций. Например, нельзя сделать так, чтобы функция operator**() обозначала возведение в степень.

4. Не подлежат перегрузке следующие операции:

sizeof	(операция sizeof)
.	операция принадлежности
.*	операция указатель на элемент класса
::	операция определения диапазона доступа
?:	условная операция
typeid	операция RTTI
constcast	операция преобразования типа
dynamic_cast	операция преобразования типа
reinterpret_cast	операция преобразования типа
static_cast	операция преобразования типа

5. Для перегрузки следующих операторов необходимо использовать только функции-элементы:

=	оператор присваивания
()	оператор вызова функции
[]	оператор индексации
->	доступ к элементу класса посредством операции указателя

В дополнение к этим формальным ограничениям нужно разумно подходить к проблеме перегрузки операций. Например, не следует перегружать операцию * таким образом, чтобы она производила обмен элементами данных между двумя объектами Time. Ничего в этой записи не говорит о том, что выполняется такая операция, так что будет лучше, если конкретный метод класса будет определён с именем, несущем в себе информацию, например, Swap().

Имеет смысл ввести некоторые другие операции для класса Time. Например, вам может понадобиться вычесть одно временное значение из другого или умножить его на некоторый множитель. Для этого требуется перегрузка операций вычитания и умножения. Метод тот же, что и применённый выше для операции сложения, – создаются методы operator-() и operator*(). Иначе говоря, в объявление класса следует добавить следующие прототипы:

```
Time operator-(const Time & t) const
Time operator*(double n) const
```

Далее в файл реализации добавьте следующие определения методов:

```
Time Time::operator-(const Time & t) const
{
Time diff; int totl, tot2 ;
totl = t.minutes + 60 * t.hours;
tot2 = minutes + 60 * hours;
diff. minutes = (tot2 – totl) % 60;
diff.hours = (tot2 – totl) / 60;
return diff;
}

Time Time::operator* (double mult) const
{
Time result;
long totalminutes = hours * mult * 60 + minutes * mult;
result.hours = totalminutes / 60;
result.minutes = totalminutes % 60;
return result;
}
```

После того как эти изменения будут выполнены, можно проверить новые определения с помощью программного кода (здесь предполагается, что файлы модифицированных классов изменились и вместо версии mytimel появилась версия mytime2)

```
// usetime2.cpp – используется третий вариант
// класса Time
// файлы usetime2.cpp и mytime2.cpp
// компилируются в единый программный модуль

#include <iostream>
#include "mytime2.h"
using namespace std;
int main()
{
Time A;
Time B (5, 40);
Time C (2, 55);
Show () ;
Show () ;
Show();
A = B + C;  // operator+()
A.Show();
A = B – C;  // operator-()
A.Show () ;
A = B * 2.75; // operator* ()
A.Show() ;
return 0;
}
```

Результаты выполнения программы:
0 hours, 0 minutes;
5 hours, 40 minutes;
2 hours, 55 minutes;
8 hours, 35 minutes;
2 hours, 45 minutes;
15 hours, 35 minutes.

4.6. ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

C++ управляет доступом к приватным разделам объекта класса. Обычно общедоступные методы класса служат единственным средством доступа, однако иногда такое ограничение оказывается чрезмерно жёстким и не позволяет решать некоторые конкретные проблемы в процессе создания программ. Для таких случаев C++ предусматривает другую форму доступа – дружественные структуры. Дружественные структуры принимают формы:

– дружественных функций;
– дружественных классов;
– дружественных функций-элементов.

Делая функцию дружественной по отношению к какому-либо классу, вы наделяете её такими же привилегиями доступа, какими обладает функция-элемент этого класса.

Довольно часто перегрузка бинарной операции (операции, имеющей два аргумента) класса порождает необходимость в дружественных структурах. Умножение объекта Time на вещественное число может служить примером подобного рода ситуации, поэтому перейдём к её изучению.

В примере с классом Time перегруженная операция умножения отличается от остальных двух перегруженных операций тем, что она работает с двумя различными типами. Иначе говоря, каждая из операций сложения и вычитания выполняется над двумя величинами типа Time, в то время как операция умножения использует сочетание значения Time со значением типа double. Это обстоятельство ограничивает область применения этой операции. Напомним, что левый операнд представляет собой вызывающий объект. Иначе говоря,

```
A = B*2.75
```

преобразуется в следующее обращение к функции-элементу:

```
A = B.operator*(2.75)
```

Что можно сказать о таком операторе?

```
A = 2.75*B;  // не соответствует функции элементу
```

В принципе, для 2.75*B результат должен быть таким же, как и для B*2.75, однако первое выражение не может соответствовать функции-элементу, поскольку 2.75 не является объектом типа Time. Напомним, что левый операнд – это вызывающий объект, но 2.75 не является объектом. Следовательно, компилятор не может заменить это выражение обращением к функции-элементу.

Один из способов, позволяющих обойти эту трудность, заключается в том, чтобы уведомить каждого (и помнить об этом самому), что следует использовать только B*2.75 и ни в коем случае не 2.75*B. Это и есть пример дружественного отношения к пользователю, что не всегда характерно для ООП.

Существует и другая возможность – функция, не являющаяся элементом класса. (Напомним, что большая часть операций может

быть перегружена с помощью функции-элемента либо посредством функции, не являющейся элементом какого-либо класса.) Функция, не являющаяся элементом какого-либо класса, не вызывается объектом. Причина состоит в том, что любые значения, которые она использует, в том числе и объекты, являются явными аргументами. Следовательно, компилятор может сопоставить выражение

```
A = 2.75*B; // не может соответствовать функции-элементу
```

со следующим вызовом функции, не являющейся элементом какого-либо класса:

```
A = operator* (2.75, B);
```

Эта функция будет иметь такой прототип:

```
Time operator* (double mult, const Time & t);
```

В функции перегруженной операции, не являющейся элементом класса, левый операнд выражения операции соответствует первому аргументу операторной функции, а правый операнд соответствует второму аргументу.

Использование функции, не являющейся элементом класса, решает проблему расстановки операндов в нужном порядке (сначала double, а затем Time), но при этом возникает новая проблема: функции, не являющиеся элементами класса, не могут осуществить прямой доступ к приватным данным этого класса. Итак, по меньшей мере обычные функции, не являющиеся элементами соответствующего класса, не получают доступа к данным этого класса. Однако имеется специальная категория функций, не являющихся членами класса, получивших название дружественных, которые способны обеспечить доступ к приватным элементам класса.

Первым шагом к созданию дружественной функции является размещение прототипа в объявлении класса, при этом объявлению предпосылается ключевое слово friend:

```
// размещается в объявлении класса
friend Time operator*(double mult, const Time & t);
```

Что касается этого прототипа, то следует отметить две его особенности. Несмотря на то, что функция operator*() объявлена в объявлении класса, она не является функцией-элементом. Хотя функция operator*() не является функцией-элементом, она получает те же права доступа, что и функция-элемент.

Вторым шагом является формулировка определения функции. Поскольку такая функция не является функцией-элементом, не следует прибегать к помощи спецификатора Time::. Кроме того, нельзя пользоваться в определении ключевым словом friend:

```
Time operator* (double mult, const Time & t)
{
Time result;
long totalminutes = t.hours * mult * 60 + t.minutes * mult;
result.hours = totalminutes / 60; result.minutes = totalminutes % 60;
return result;
}
```

При наличии такого объявления оператор

```
A = 2.75*B;
```

преобразуется в оператор

```
A = operator*(2.75, B);
```

и при этом вызывается дружественная функция, не являющаяся функцией-элементом, которую мы только что определили.

Одним словом, функция, дружественная по отношению к какому-либо классу, не являясь его элементом, имеет те же права доступа, что и функция-элемент.

На первый взгляд может показаться, что дружественные конструкции нарушают принцип сокрытия данных ООП, поскольку механизм дружественных конструкций позволяет функциям, не являющимся элементами конкретного класса, осуществлять доступ к приватным данным этого класса. Тем не менее, такая точка зрения поверхностна. Вместо этого, считайте дружественные функции частью расширенного интерфейса класса. Например, с концептуальной точки зрения значение Time, умноженное на double, во многом совпадает со значением double, умноженным на Time. Тот факт, что в первом случае требуется дружественная функция, в то время как во втором случае умножение может быть выполнено при участии функции-элемента, является следствием особенностей синтаксиса языка C++. Пользуясь как дружественной функцией, так и тем или иным методом класса, вы можете задействовать и ту, и другую операцию с одним и тем же интерфейсом пользователя. При этом следует иметь в виду, что только определение класса может отличить дружественные функции от прочих, следовательно, за объявлением класса сохраняется право определять, каким

функциям можно разрешить доступ к приватным данным. Итак, методы класса и дружественные средства – это два различных механизма, выражающие интерфейс класса.

По существу, эта конкретная дружественная функция может быть записана не как дружественная конструкция путём внесения в определение функции следующих изменений:

```
Time operator* (double mult, const Time & t)
{
return t * mult;  // используется функция-элемент
}
```

Первоначальная версия этой функции получает прямой доступ к t.minutes и t.hours, следовательно, она должна быть дружественной конструкцией. Эта версия использует только объект t класса Time как единое целое, оставляя функции-элементу манипулирование приватными значениями, так что эта версия не обязательно должна быть дружественным средством. Тем не менее, идея превратить эту версию в дружественную функцию также вполне оправдана. Самое главное, она связывает функцию таким образом, что та становится частью официального интерфейса класса. Если в дальнейшем у вас появится необходимость в том, чтобы такая функция осуществляла прямой доступ к приватным данным, достаточно изменить только определение функции, но не прототип класса.

При выполнении перегрузки операций во многих случаях вы будете поставлены перед выбором: использовать для этой цели функцию-элемент или функцию, не являющуюся таковой. Обычно вариант функции, не являющейся элементом класса, может быть представлен дружественной функцией, которая осуществляет прямой доступ к приватным данным соответствующего класса. В качестве примера рассмотрим операцию сложения для класса Time. У неё есть прототип в объявлении класса Time:

```
// вариант с использованием функции-элемента
Time operator+ (const Time & t) const;
```

Вместо этого класс мог бы воспользоваться следующим прототипом:

```
// вариант с использованием функции,
// не являющейся членом класса
friend Time operator+(const Time & t1, const Time & t2) ;
```

Операция сложения требует двух операндов. В случае использования функции-элемента один операнд передаётся неявно с помощью указателя this, а второй – явно как аргумент функции. Если речь идёт о дружественной функции, оба операнда передаются как аргументы.

Любой из этих двух прототипов соответствует выражению B + C, в котором B и C являются объектами класса Time. Иначе говоря, компилятор может преобразовать оператор

```
A = B + C;
```

в какой-либо из двух следующих:

```
A = B.operator+(C); // функция-элемент
A = operator+(B, C); // не является функцией-элементом
```

Имейте в виду, что обязательно следует выбрать какую-либо одну из форм при определении заданной операции, но не обе сразу. Поскольку обе формы соответствуют одному и тому же выражению, определение одновременно обеих форм рассматривается как неопределённость.

4.7. НАСЛЕДОВАНИЕ КЛАССОВ

Одна из основных целей объектно-ориентированного программирования – обеспечение кода многократного использования. При разработке нового проекта, особенно если проект большой, хорошо иметь возможность повторно использовать проверенный код, а не изобретать его снова. Использование старого кода позволяет экономить время и, поскольку он уже использовался и проверен, может помочь избежать появления ошибок в программе. Кроме того, чем меньше приходится вникать в детали, тем больше можно сконцентрироваться на общей стратегии программы.

Классы C++ обеспечивают высокий уровень возможностей многократного использования. В настоящее время многие поставщики предлагают библиотеки классов, которые состоят из объявлений и реализаций классов. Поскольку класс объединяет представление данных с методами класса, он предоставляет более комплексный пакет, чем библиотека функций. Наследование классов позволяет производить новые классы из старых, причём производный класс наследует свойства, включая методы, старого класса, называемого базовым. Вот некоторые действия, которые можно выполнять с наследованием: добавлять функциональные возможности к существующему классу; выпол-

нять добавление элементов к данным, представляющим класс; изменять поведение метода класса.

Конечно, этих же целей можно достигнуть, дублируя код исходного класса и изменяя его, но механизм наследования позволяет выполнить задачу, обеспечивая только новые свойства. Для получения производного класса даже не требуется доступ к исходному коду. Таким образом, при приобретении библиотеки классов, которая предоставляет только заголовочные файлы и скомпилированный код для методов класса, всё же можно производить новые классы, основанные на библиотечных классах. И наоборот, можно предоставлять свои собственные классы другим пользователям, храня части своей реализации в секрете и всё же предоставляя клиентам возможность добавлять свойства к вашим классам.

Когда один класс наследуется из другого, исходный класс называется базовым, а наследующий – производным. Таким образом, чтобы проиллюстрировать наследование, следует начать с базового класса. Например, необходим класс, представляющий основной расчётный счёт в банке. В листинге показан заголовок для упрощённого класса BankAccount, удовлетворяющего этой потребности. Он включает элементы данных, представляющие имя клиента, номер счёта и баланс. Класс содержит методы для создания счёта, внесения вкладов, снятия со счётов и отображения данных счёта.

```
// bankacct.h – простой класс BankAccount.

#ifndef _BANKACCT_H_
#define _BANKACCT_H_
class BankAccount
{
private:
enum {MAX = 35};
char fullName[MAX];
long acctNum; double balance;
public:
BankAccount(const char *s = "Nullbody", long an = -1,
double bal = 0.0); void Deposit(double amt);
void Withdraw(double amt);
double Balance() const;
void ViewAcct() const;
};
#endif
```

Класс использует методику создания константы области класса путём использования функции enum. Далее следуют методы класса.

```
// bankacct.cpp – методы для класса BankAccount
#include <iostream>
#include "bankacct.h"
#include <string.h>

BankAccount::BankAccount(const char *s, long an, double bal)
{
strncpy (fullName, s, MAX – 1) ;
fullName [MAX – 1] = ' \0' ;
acctNum = an ;
balance = bal;
}
void BankAccount::Deposit(double amt)
{
balance += amt;
}
void BankAccount::Withdraw(double amt)
{
if (amt <= balance) balance -= amt;
else
cout <<"Withdrawal amount of $" <<amt << " exceeds your
balance.\n"
<<"Withdrawal canceled.\n";
}


double BankAccount::Balance() const
{
return balance;
}
void BankAccount::ViewAcct() const
{
cout << "Client: " << fullName << endl;
cout << "Account Number: " << acctNum << endl ;
cout << "Balance: $" << balance << endl;
}
```

Проиллюстрируем короткий список свойств класса.

```
// usebank.cpp компилируется с bankacct.cpp

#include <iostream>
#include <string.h>
#include "bankacct.h"
int main()
{
BankAccount IPet("Ivan Petrov", 381299, 4000.00);
IPet.ViewAcct() ;
IPet.Deposit(5000.00);
cout << "New balance: $"<<IPet.Balance() << endl;
IPet.Withdraw(8000.00);
cout << "New balance: $"<< IPet.Balance()<< endl;
IPet.Withdraw(1200.00) ;
cout << "New balance: $"<< IPet.Balance() << endl ;
return 0;
}
```

Ниже приведён вывод программы:
Client: Ivan Petrov.
Account Number: 381299.
Balance: $4000.00.
New balance: $9000.00.
New balance: $1000.00.
Withdrawal amount of $1200.00 exceeds your balance.
Withdrawal canceled.
New balance: $1000.00.

Наследование – отношение is-a. Теперь, когда BankAccount является классом, с которым можно работать, из него можно получить новый класс. Однако перед этим давайте рассмотрим модель, лежащую в основе наследования C++. Фактически C++ имеет три разновидности наследования: общедоступную, защищённую и приватную. Общедоступное наследование – наиболее общая форма, и она моделируется отношением is-a («является объектом»). Это сокращённая форма выражения того, что объект производного класса должен также быть объектом базового класса. Предположим, например, что имеется класс Fruit. В нём можно было бы задавать, скажем, вес и энергетическую ценность плода. Поскольку банан – конкретный вид плода, класс Banana можно было бы получить из класса Fruit. Новый класс насле-

довал бы все элементы данных исходного класса, так что объект Banana содержал бы элементы, представляющие вес и энергетическую ценность банана. Новый класс Banana мог бы также добавлять элементы, которые присущи именно бананам, а не плодам вообще.

Чтобы выяснить, что собой представляет отношение is-a, давайте рассмотрим несколько примеров, которые не соответствуют этой модели. Общедоступное наследование не моделирует отношение has-a («имеет объект»). Например, завтрак мог бы содержать плод. Но завтрак вообще – это не плод. Следовательно, предпринимая попытку поместить плод в завтрак, не следует получать класс Lunch из класса Fruit. Правильный способ обработки помещения плода в завтрак состоит в рассмотрении задачи в качестве отношения has-a: завтрак содержит плод. Это легче всего смоделировать путём включения объекта Fruit в качестве элемента данных класса Lunch.

Общедоступное наследование не моделирует отношение is-like-a («подобный объекту»), т.е. оно не создаёт подобия. Часто говорят, что адвокаты подобны акулам. Но это не означает буквально, что адвокат – акула. Следовательно, вам не требуется получать класс Lawyer из класса Shark. Путём наследования можно добавлять свойства к базовому классу; но не удалять свойства из него. В некоторых случаях общие характеристики могут обрабатываться путём разработки класса, определяющего эти характеристики, и последующего использования этого класса в отношении is-a или has-a для определения связанных классов.

Общедоступное наследование не моделирует отношение is-implemented-as-a («реализован в качестве объекта»). Например, стек можно было бы реализовать путём использования массива. Однако было бы неверным получать класс Stack из класса Array. Стек – это не массив. Например, индексация массива не является свойством стека. Кроме того, стек мог бы быть реализован каким-либо другим способом, например, путём использования связанного списка. Правильным подходом было бы скрытие реализации массива, присвоение стеку элемента приватного объекта Array.

Общедоступное наследование не моделирует отношение uses-a («использует объект»). Например, компьютер может использовать лазерный принтер, но нет смысла получать класс Printer из класса Computer или наоборот. Однако можно было бы создать удобные функции или классы для выполнения обмена данными между объектами Printer и Computer.

Ничто в языке C++ не препятствует использованию метода общедоступного наследования для моделирования отношений has-a, is-implemented-as-a или uses-a. Но обычно это приводит к проблемам программирования, поэтому желательно придерживаться отношений is-a.

Для демонстрации примера наследования представим, что банк также предоставляет текущий счёт. Этот счёт имеет все свойства регулярного счёта, а также обеспечивает блокировку от превышения кредита. Так, если клиент выписывает чек на сумму, превышающую (но не слишком) имеющуюся на текущем счету, банк покроет чек, выставив счёт на оплату излишка и наложив определённый штраф.

В этом случае определим новый класс, который наследует все свойства класса BankAccount и дополнительно имеет необходимые новые функциональные возможности. Создавая новый класс на основе класса BankAccount, воспользуемся результатами работы, которая уже была проделана при разработке класса BankAccount. Другими словами, в данном случае повторно используется проверенный код.

Новый класс назовем Overdraft. Удовлетворяет ли он условиям отношения is-a? Безусловно. Все, что является истинным для объекта BankAccount, будет истинным и для объекта Overdraft. Это значит, что можно вносить вклады, снимать деньги со счёта и отображать информацию о счёте. Обратите внимание, что в общем случае отношение is-a не является обратимым. В общем случае плод не всегда будет бананом. Объект BankAccount не будет иметь всех возможностей объекта Overdraft.

Производный класс должен идентифицировать класс, из которого он произведен. Метод реализации этого в C++ заключается во включении имени базового класса в объявление производного класса. Если класс Overdraft производится из класса BankAccount, объявление класса начиналось бы подобно следующему:

```
class Overdraft : public BankAccount
{
```

Двоеточие указывает, что класс Overdraft основан на классе BankAccount. Этот конкретный заголовок указывает, что BankAccount – общедоступный базовый класс; такой процесс называется общедоступным произведением. Объект производного класса включает в себя объект базового класса. При осуществлении общедоступного произведения общедоступные элементы базового класса станут общедоступными элементами производного класса. Приватные разделы базового класса станут частью производного класса, но к ним можно обращаться только посредством общедоступных и защищённых методов базового класса.

Например, функция Deposit() становится также общедоступной функцией класса Overdraft. Элемент balance объекта BankAccount становится частью объекта Overdraft, но к нему можно обращаться только

посредством таких методов BankAccount, как Deposit() и конструкторов BankAccount. Одним словом, класс Overdraft наследует общедоступные элементы из базового класса наряду с доступом к ним. Их не нужно переопределять для нового класса. Производный класс содержит приватные элементы базового класса, но не может обращаться к ним иначе, кроме как используя общедоступные и защищёные методы базового класса.

C++ также поддерживает защищёные и частные произведения:

```
class Computer : protected HardDisk
{…} ;
class House : private Study
{ … } ;
```

Отметим, что, если опустить ключевое слово доступа, C++ сделает произведение приватным:

```
class House : Study   // то же, что и private Study
{…};
```

После получения производного класса к нему можно добавлять новые элементы. Фактически необходимо обеспечить новые конструкторы. Дело в том, что имя конструктора совпадает с именем класса:

```
BankAccount A; // требуется конструктор BankAccount()
Overdraft B; // требуется конструктор Overdraft()
```

При создании объекта производного класса вначале программа вызывает конструктор для базового класса, а затем конструктор для производного класса. В этом есть смысл, поскольку конструктор для производного класса может строить поверх элементов данных из базового класса; следовательно, объект базового класса должен быть создан первым. Таким образом, при определении новых конструкторов они не должны дублировать работу базовых конструкторов. Вместо этого необходимо обрабатывать только дополнительные детали, которые требуются производному классу. Например, конструктор мог бы инициализировать новые элементы данных. В общем случае конструктор производного класса должен также передавать информацию конструктору базового класса; вскоре мы рассмотрим методику для выполнения этого.

Новый деструктор не нужно добавлять явно, если только новый класс не требует очистки, кроме выполняемой базовым деструктором. Когда срок существования объекта истекает, программа вначале вызы-

вает деструктор для производного класса, если таковой существует, а затем базовый деструктор.

Вернёмся к разработке класса Overdraft и определим его следующие свойства. Счёт ограничивает денежные суммы, предоставляемые банком для покрытия превышения кредита. Значение, принятое по умолчанию, – $500, но некоторые клиенты могут начинать с другого ограничения. Банк может изменять ограничение на превышение кредита клиента. Обеспечивается возможность выставления счёта оплаты процента по ссуде. Значение, принятое по умолчанию, 10%, но некоторые клиенты могут начинать с другого тарифа. Банк может изменять величину процентной ставки. Счёт отслеживает задолженность клиента банку (ссуды на превышение кредита плюс проценты за предоставление ссуды). Пользователь не может оплачивать эту сумму путём обычного вклада на счёт или переводом с другого счёта. Вместо этого он должен оплатить сумму наличными специальному сотруднику банка, который в случае необходимости разыщет клиента. Как только долг оплачен, счёт может сбрасывать задолженность до 0.

В листинге показано объявление класса, соответствующее этим условиям.

```
// overdrft.h –объявление класса Overdraft
#ifndef _OVERDRFT_H_
#define _OVERDRFT_H_
#include "bankacct.h"
class Overdraft : public BankAccount
{
private:
double maxLoan;
double rate;
double owesBank;
public:
Overdraft(const char *s = "Nullbody", long an = -1, double bal = 0.0,
double ml = 500, double r = 0.10);
Overdraft(const BankAccount & ba, double ml = 500, double r = 0.1);
void ViewAcct () const;
void Withdraw(double amt);
void ResetMax(double m) { maxLoan = m; };
void ResetRate (double r) { rate = r; };
void ResetOwes() { owesBank = 0; }
};
#endif
```

Давайте исследуем, как реализовать производный класс, и рассмотрим некоторые методы, начиная с конструкторов. Вначале давайте подумаем о процессе конструирования. Программа не может создать объект Overdraft до тех пор, пока вначале не создаст объект BankAccount. Поэтому базовый конструктор должен быть вызван прежде, чем программа введёт код для производного конструктора. С другой стороны, базовый конструктор не может быть вызван до тех пор, пока не вызван производный конструктор, поскольку именно при обращении к производному конструктору программе сообщается о потребности в базовом конструкторе. Давайте рассмотрим следующий конструктор Overdraft.

```
    Overdraft (const char *s = "Nullbody", long an = 1, double bal =
0.0,
    double ml = 500, double r = 0.10);
```

При этом имеется пять аргументов, три из которых обеспечивают значения для раздела BankAccount, а два значения для раздела Overdraft. Последние два аргумента использовать достаточно просто:

```
    // незавершенная версия
    Overdraft::Overdraft(const char *s , long an, double bal, double ml,
double r)
    {
    maxLoan = ml ;
    owesBank =0.0;       // начало без задолженности
    rate = r;
    }
```

А как дела обстоят с компонентом BankAccount? Вначале давайте рассмотрим, что произошло бы в случае использования этой незавершённой версии конструктора. Объект базового класса создаётся прежде, чем добавляется производный компонент. С точки зрения синтаксиса данного конструктора это означает, что объект базового класса создаётся прежде, чем выполнение программы передаётся оператору в теле конструктора. Поскольку никакой конструктор не упоминается явно, это означает, что конструктор BankAccount по умолчанию используется для создания компонента базового класса. Давайте перефразируем это: если не указано иное, конструктор производного класса вызывает заданный по умолчанию базовый конструктор перед вхождением в тело конструктора производного класса.

Однако в данном случае заданный по умолчанию конструктор некорректен, поскольку он использует значения, принятые по умолчанию, вместо требуемых значений. C++ предлагает специальный синтаксис, который позволяет определять, какой конструктор должен использоваться.

```
Overdraft::Overdraft(const char *s , long an, double bal,
double ml, double r) : BankAccount(s, an, bal)
{
maxLoan = ml;
owesBank = 0.0;
rate = r;
}
```

Здесь часть

```
: BankAccount(s, an, bal)
```

означает: «Вызовите конструктор BankAccount(const char *, double, double) для создания части базового класса объекта Overdraft». Путём использования этого механизма значения первых трёх аргументов конструктора Overdraft передаются конструктору BankAccount. Таким образом, конструктор BankAccount устанавливает унаследованные элементы, а тело конструктора Overdraft устанавливает новые элементы.

Итак, конструктор производного класса всегда вызывает конструктор базового класса перед выполнением операторов в теле конструктора производного класса. Программа использует заданный по умолчанию базовый конструктор, если только другой конструктор не указан явно путём использования синтаксиса списка инициализатора. В этом случае аргументы из конструктора производного класса можно использовать в качестве аргументов конструктора базового класса.

Конструктор для производного класса может использовать механизм списка инициализатора для передачи значений конструктору базового класса.

```
derived::derived(typel x, type2 y):base(x, y)
{…}
```

Здесь derived – производный класс, base – базовый класс, а x и y – переменные, используемые конструктором базового класса. За исключением случая виртуальных базовых классов, класс может передавать значения обратно только непосредственному базовому классу. Однако этот класс может использовать тот же самый механизм для передачи

возвращаемой информации своему непосредственному базовому классу и т.д. Если конструктор базового класса отсутствует в списке инициализатора, программа будет использовать заданный по умолчанию конструктор базового класса. Список инициализатора может использоваться только с конструкторами.

Рассмотрим второй конструктор класса Overdraft:

```
Overdraft(const BankAccount & ba, double ml = 500, double r = 0.1);
```

Он предназначен для того, чтобы позволить преобразование от счёта BankAccount в счёт Overdraft. Здесь аргумент ba обеспечивает информацию старого счёта, а остальные аргументы обеспечивают информацию для новых элементов данных. Вопрос состоит в том, как использовать аргумент BankAccount для инициализации раздела BankAccount. Поскольку это действие создаёт копию объекта BankAccount, необходимо использовать конструктор копии:

```
Overdraft::Overdraft(const BankAccount & ba,
double ml, double r): BankAccount(ba)
{
maxLoan = ml ; owesBank = 0.0; rate = r ;
}
```

Действительно, объявление BankAccount не определяет конструктор копии явно. Однако вспомните, что компилятор обеспечивает заданный по умолчанию конструктор копии, если конструктор копии необходим и ни один не определен. Он выполняет копирование с учётом элементов, которые прекрасно подходит для объекта BankAccount.

Класс Overdraft не определяет функцию Deposit(), и, следовательно, объект Overdraft будет использовать BankAccount::Deposit(). Такое же поведение сохраняется для функции Balance(). Но новый класс определяет метод ViewAcct(), значит, объект Overdraft будет использовать Overdraft::ViewAcct(). Давайте посмотрим, как это реализовать. Во-первых, кое-что не будет работать:

```
void Overdraft::ViewAcct() const // НЕВЕРНАЯ ВЕРСИЯ
{
cout << "Client: " <<fullName << endl; // неверно
cout << "Account Number: " << acctNum << endl;        // неверно
cout << "Balance: $" << balance << endl;          // неверно
cout << "Maximum loan: $" << maxLoan << endl;
cout << "Owed to bank: $" << owesBank << endl;
}
```

Проблема такова, что должно стать очевидным следующее: производный класс не может непосредственно обращаться к приватным данным и методам базового класса. Так, объект Overdraft содержит объект BankAccount с элементами fullName, acctNum и balance, но он не может обращаться к ним по имени. Дело в том, что общедоступный раздел базового класса определяет интерфейс для того класса, а остальная часть программы, включая производные классы, должна использовать этот интерфейс. В данном случае класс Overdraft может использовать общедоступный интерфейс класса BankAccount, чтобы получить доступ к данным BankAccount. Например, метод Overdraft может использовать метод BankAccount::ViewAcct():

```
void Overdraft::ViewAcct() const
{
BankAccount::ViewAcct(); // отображение базовой части
cout << "Maximum loan: $" << maxLoan << endl;
cout << "Owed to bank: $" << owesBank << endl;
}
```

Здесь обязательно нужно было использовать оператор определения диапазона. Если его пропустить, метод Overdraft::ViewAcct() будет выполнять рекурсивный вызов самого себя:

```
void Overdraft::ViewAcct () const
{
…
ViewAcct(); // ОШИБКА!
//рекурсивное обращение к Overdraft::ViewAcct()
BankAccount::ViewAcct(); // вызов версии базового класса
…
}
```

Подведём итог, если производный класс не переопределяет метод базового класса, объект производного класса использует метод базового класса. Если производный класс переопределяет метод, объекты производного класса используют новое определение.

Рассмотрим полную реализацию класса Overdraft.

```
// overdrft.cpp – методы класса Overdraft
#include <iostream>
#include "overdrft.h"
```

```
Overdraft::Overdraft(const char *s, long an, double bal,
double ml, double r) : BankAccount(s, an, bal)
{
maxLoan = ml ;
owesBank = 0.0;
rate = r;
}
// использует заданный по умолчанию конструктор копии
Overdraft::Overdraft(const BankAccount & ba,
double ml, double r): BankAccount(ba)
{
maxLoan = ml ;
owesBank = 0.0;
rate = r ;
}
// переопределение работы ViewAcct()
void Overdraft::ViewAcct() const
{
BankAccount::ViewAcct(); // отображение базовой части
cout << "Maximum loan: $" << maxLoan << endl;
cout << "Owed to bank: $" << owesBank << endl;
}
// переопределение работы Withdraw()
void Overdraft::Withdraw(double amt)
{
double bal = Balance();
if (amt <= bal) BankAccount::Withdraw(amt);
else if ( amt <= bal + maxLoan – owesBank)
{
double advance = amt – bal;
owesBank += advance * (1.0 + rate);
cout << "Bank advance: $" << advance << endl;
cout << "Finance charge: $" << advance * rate << endl;
Deposit(advance);
BankAccount::Withdraw(amt);
}
else cout << "Credit limit exceeded. Transaction cancelled.\n";
}
```

Следующий шаг заключается в проверке производного класса. Короткая программа, приведённая ниже, выполняет эту операцию. Она

должна быть скомпилирована с файлами overdrft.cpp и bankacct.cpp, потому что производный класс использует определения базового класса.

```
// useover.cpp--проверяет класс Overdraft
// компилируется с файлами bankacct.cpp и overdrft.cpp
#include <iostream>
#include "overdrft.h"
int main ()
{
BankAccount IPet ("Ivan Petrov", 381299, 4000.00);
// преобразование в новый тип счёта
Overdraft IPet2 (IPet);
IPet2.ViewAcct() ;
cout << "Depositing $5000:\n”;
IPet2.Deposit(5000.00);
cout << "New balance: $" << IPet2.Balance() << "\n\n";
cout << "Withdrawing $8000:\n";
IPet2.Withdraw(8000.00);
cout << "New balance: $" << IPet2.Balance() << "\n\n";
cout << "Withdrawing $1200:\n";
IPet2.Withdraw(1200.00);
IPet2.ViewAcct();
cout << "\nWithdrawing $500:\n";
IPet2.Withdraw(500.00);
IPet2.ViewAcct();
return 0;
}
```

Результаты выполнения программы:

```
Client: Ivan Petrov
Account Number: 381299
Balance: $4000.00
Maximum loan: $500.00
Owed to bank: $0.00
Depositing $5000:
New balance: $9000.00
Withdrawing $8000:
New balance: $1000.00
Withdrawing $1200:
Bank advance: $200.00
Finance charge: $20.00
```

```
Client: Ivan Petrov
Account Number: 381299
Balance: $0.00
Maximum loan: $500.00
Owed to bank: $220.00

Withdrawing $500:
Credit limit exceeded. Transaction cancelled.
Client: Ivan Petrov
Account Number: 381299
Balance: $0.00
Maximum loan: $500.00
Owed to bank: $220.00
```

Управление доступом – protected. До сих пор в примерах классов для управления доступом к элементам класса использовались ключевые слова public и private. Существует ещё одна категория доступа, обозначаемая ключевым словом protected. Ключевое слово protected подобно ключевому слову private тем, что внешний мир может получать доступ к элементам класса в защищённом разделе только путём использования элементов класса. Различие между private и protected выступает на сцену только внутри классов, производных от базового класса. Элементы производного класса могут обращаться к защищенным элементам базового класса непосредственно, но они не могут непосредственно обращаться к приватным элементам базового класса. Таким образом, элементы в защищённой категории ведут себя подобно приватным элементам до тех пор, пока дело касается внешнего мира, но действуют подобно общедоступным элементам, когда речь идёт о производных классах.

Например, предположим, что класс BankAccount объявил элемент balance в качестве защищённого (protected):

```
class BankAccount
{
protected:
double balance;
…
} ;
```

В этом случае класс Overdraft мог бы обращаться к balance непосредственно, не используя методы класса BankAccount. Например, ядро метода Overdraf::Withdraw() могло бы быть записано следующим образом:

```
void Overdraft::Withdraw(double amt)
{
if (amt <= balance) // обращается к  balance непосредственно
balance -= amt;
else if ( amt <= balance + maxLoan – owesBank)
{
double advance = amt – balance;
owesBank += advance * (1.0 + rate);
cout << "Bank advance: $" << advance << endl;
cout << "Finance charge: $" << advance * rate << endl;
Deposit(advance);
balance -= amt;
}
else cout << "Credit limit exceeded. Transaction cancelled.\n";
}
```

Отношение is-a, ссылки и указатели. Одним из способов, с помощью которого общедоступное наследование позволяет моделировать отношение is-a, заключается в том, как при этом обрабатываются указатели и ссылки на объекты. Обычно C++ не позволяет присваивать адрес одного типа указателю другого типа. Он также не позволяет ссылке на один тип ссылаться на другой тип:

```
double x = 2.5;
int * pi = &x; // недопустимое присвоение, несоответствующие
// типы указателей
long & rl = x; // недопустимое назначение, несоответствующие
// типы ссылок
```

Однако ссылка или указатель на базовый класс может ссылаться на объект производного класса без выполнения явного приведения типа. Например, следующие инициализации являются допустимыми:

```
Overdraft IPet ("Ivan Petrov", 493222, 2000)
BankAccount * pIP = &IPet; // верно
BankAccount & rIP = IPet; // верно
```

Преобразование ссылки или указателя производного класса в ссылку или указатель базового класса называется приведением вверх, и оно всегда допускается для общедоступного наследования, не требуя явного приведения типа. Это правило – часть выражения отношения is-a. Объект Overdraft является объектом BankAccount в том смысле,

что он наследует все элементы данных и функции-элементы объекта BankAccount. Следовательно, объектом Overdraft можно манипулировать так же, как и объектом BankAccount. Так, функция, созданная для обработки ссылки BankAccount, может без особых проблем выполнять те же самые действия по отношению к объекту Overdraft. Эта же идея применяется при передаче указателя на объект в качестве аргумента функции.

Противоположный процесс – преобразование указателя или ссылки базового класса в указатель или ссылку производного класса – называется приведением вниз, и он не допускается без явного приведения типа. Причина этого ограничения заключается в том, что в общем случае отношение is-a необратимо. Производный класс мог бы добавлять новые элементы данных, а использующие эти элементы данных функции-элементы класса не были бы применимы к базовому классу. Например, предположим, что вы производите класс Singer из класса Employee, добавляя элемент данных, представляющий вокальный диапазон певца, и функцию-элемент, названную range(), которая сообщает значение вокального диапазона. Было бы бессмысленным применять метод range() к объекту Employee. Но если бы неявное приведение вниз было допустимым, можно было бы случайно установить указатель-на-Singer на адрес объекта Employee и использовать этот указатель для вызова метода range().

ЗАКЛЮЧЕНИЕ

Авторы постарались отразить основные существующие подходы в технологии программирования, применяемые при разработке программного обеспечения. Следует отметить, в пособии не нашёл отражение ряд важных вопросов, таких как логическое и функциональное программирование, описание стандартных библиотек, подходов и методов тестирования программных продуктов и т.д. Все эти средства необходимы программисту, работающему на современном уровне, однако при написании пособия авторы руководствовались разумным, по их мнению, компромиссом между требованием полноты представляемого материала, с одной стороны, и громоздкостью изложения – с другой.

Большое внимание авторы уделили описанию основ языка C++, владение которыми является необходимой базой для разработки программного обеспечения, так как создание эффективной, легко читаемой и расширяемой программы невозможно без знания механизмов реализации возможностей языка и способов их взаимодействия. Неполное же понимание функционирования языка, напротив, приводит к созданию программ, полных ошибок и не поддающихся сопровождению.

Изучение дисциплины «Технология программирования» является одним из первых шагов на пути освоения языков программирования и подходов к созданию программ. Знания и умения, приобретённые при этом, являются необходимой основой для дальнейшего изучения современных библиотек и технологий разработки сложных программных продуктов.

СПИСОК ЛИТЕРАТУРЫ

1. Жоголев, Е.А. Технология программирования / Е.А. Жоголев. – М. : Научный мир, 2004. – 216 с.
2. Иванова, Г.С. Технология программирования : учебник для вузов / Г.С. Иванова. – М. : Изд-во МГТУ им. Н.Э. Баумана, 2002. – 320 с.
3. Павловская, Т.А. С/С++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб. : Питер, 2003. – 461 с.
4. Громов, Ю.Ю. Языки СИ и СИ++ для решения инженерных и экономических задач : учебное пособие / Ю.Ю. Громов, С.И. Татаренко ; Тамб. гос. техн. ун-т. – Тамбов : Изд-во ТГТУ, 2001. – 190 с.
5. Липпман, С.Б. Основы программирования на С++ / С.Б. Липпман. – М. : Вильямс, 2002. – 256 с.
6. Шилдт, Г. С/С++. Справочник программиста / Г. Шилдт. – М. : Вильямс, 2000. –- 448 с.
7. Прата, С. Язык программирования С++. Лекции и упражнения / С. Прата. – М. : ДиаСофт, 2005. – 645 с.
8. Подбельский, В.В. Язык С++ / В.В. Подбельский. – М. : Финансы и статистика, 2003. – 562 с.
9. Страуструп, Б. Язык программирования С++. Специальное издание / Б. Страуструп. – М. : Бином-Пресс, 2008. – 1104 с.
10. Давыдов, В.Г. Технологии программирования С++ / В.Г. Давыдов. – СПб. : БХВ-Петербург, 2005. – 672 с.
11. Культин, Н. С/С++ в задачах и примерах / Н. Культин. – СПб. : BHV-СПб, 2005. – 288 с.
12. Труб, И. Объектно-ориентированное моделирование на С++ / И. Труб. – СПб. : Питер, 2006. – 416 с.
13. Дейтел, Х.М. Как программировать на С++ / Х.М. Дейтел, П.Дж. Дейтел. – М. : Бином, 2008. – 1454 с.

ОГЛАВЛЕНИЕ