ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Р.Е. АЛЕКСЕЕВА

Е.А. СУХАНОВА

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

КОМПЛЕКС УЧЕБНО-МЕТОДИЧЕСКИХ МАТЕРИАЛОВ

Часть 1

Рекомендовано Ученым советом Нижегородского государственного технического университета им. Р.Е. Алексеева в качестве учебно-методического пособия для студентов всех форм обучения, включающих элементы дистанционных технологий по специальности 230201 «Информационные системы и технологии»

Нижний Новгород 2008

УДК 681.3.06

Суханова Е.А. Программирование на языке высокого уровня: комплекс учебно-методических материалов: ч.1 / Е.А.Суханова; НГТУ им. Р.Е. Алексеева. Н.Новгород, 2008. – 144 с.

Содержит рабочую программу дисциплины, опорный конспект лекций, глоссарий и список рекомендуемой учебной литературы

Предназначен для студентов всех форм обучения, включающих элементы дистанционных технологий.

Редактор Е.В. Комарова

Подписано в печать 09.07.2008. Формат 60 x 84 $^{1}/_{16}$. Бумага офсетная. Печать офсетная. Усл. печ. л. 9,0. Уч.-изд. л. 8,4. Тираж 200 экз. Заказ .

Нижегородский государственный технический университет им. Р.Е. Алексеева. Типография НГТУ. 603950, ГСП-41, г. Нижний Новгород, ул. Минина, 24.

СОДЕРЖАНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Целью дисциплины является изучение модульного и объектно-ориентированного программирования, основных понятий: объекта, реализации объектов, построения иерархии объектов, полиморфизма, особенностей разработки современного программного обеспечения.

В процессе изучения дисциплины студенты овладевают основами объектно-ориентированного программирования, изучают основные алгоритмы, используемые в базовых задачах. Должны уметь использовать объектно-ориентированный подход при решении задач. В ходе лабораторных занятий студенты закрепляют знания, полученные на лекциях и при самостоятельном изучении рекомендованной литературы.

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

Ведомость числа часов по рабочим учебным планам

Наименование специальности и ее шифр	Объем работы студентов, ч								Распределение по семестрам			
	Аудиторной						Внеаудиторной					
	всего	лекции	лаб. занятия	практич. занятия	семинары	курс. работы	индивид. занятия	самост. работа	экзамены	зачеты	курсовой проект	курсовые работы
Информационные системы и технологии (дистанционное обучение, ИРИТ) 230201	204	51	34	-	-	+	-	119	3	-	-	3

По учебному плану на дисциплину отводится 136 часов, из которых 68 – на самостоятельную работу студента. Аудиторные занятия: 51 час лекций и 34 лабораторных занятий. Студенты, получившие зачет по лабораторным занятиям и курсовому проектированию, допускаются к экзамену.

Распределение лекционных часов по темам

№ п/п	Название темы	Курс с общим объемом 136 часов	
		Лекции 51 ч	Лаборат. занятия 34 ч.
1	2	3	4
1	**Введение в программирование. Введение в C++** Классификация языков программирования. Свойства языков программирования. История и назначения языка C/C++. Основные парадигмы программирования	2	-
2	**Типы данных** Понятие переменной. Встроенные типы данных. Операции и выражения. Служебные слова C++. Приоритет операций	2	-
3	**Операторы и выражения** Понятие оператора. Пустой оператор. Оператор `return`. Оператор условного выбора. Операторы цикла. Оператор множественного выбора	2	-
4	**Массивы** Определение и объявление массивов. Сортировка и поиск массивов в программе. Многомерные массивы	3	-
5	**Указатели** Понятие указателя. Объявление и инициализация указателей. Операции с указателями. Взаимосвязь указателей и массивов. Динамическое выделение памяти под массивы	4	4
6	**Функции** Определения функций. Классы памяти и область действия. Рекурсия. Перегрузка функций. Передача массивов в функции. Указатели на функции	4	4
7	**Введение в обработку строк** Понятие символов и строк в C/C++. Библиотечные функции для работы со строками	2	4
8	**Работа с файлами** Понятие файла. Библиотечные функции для работы с файлами	2	4
9	**Компоновка программ и препроцессор** Проблема использования общих функций и имен. Использование включаемых файлов. Препроцессор	1	-

Окончание таблицы

1	2	3	4
10	**Структуры** Определение структур. Доступ к элементам структур. Битовые поля. Объединения. Построение связных списков на основе структур с самоадресацией	4	8
11	**Классы и абстрагирование данных** Определения классов. Отделение интерфейса от реализации. Область действия класс и доступ к элементам класса. Спецификаторы доступа	2	-
12	**Специальные члены класса** Конструкторы. Деструкторы. Указатель `this`	2	-
13	**Отношения между классами** Композиция. Дружественность	2	-
14	**Константные элементы и экземпляры класса. Статические элементы класса** Константные элементы класса. Константные объекты. Статические элементы-данные и элементы-функции	2	-
15	**Одиночное наследование** Типы наследования. Вызов конструкторов и деструкторов при наследовании. Неявное преобразование объектов	2	4
16	**Множественное наследование** Понятие множественного наследования. Виртуальные базовые классы. Порядок инициализации различных частей создаваемого класса	2	-
17	**Перегрузка операций** Понятие перегрузки операций. Общие правила перегрузки	2	2
18	**Виртуальные функции и полиморфизм** Виртуальные функции. Абстрактные и конкретные классы. Полиморфизм	4	4
19	**Шаблоны** Шаблоны функций. Шаблоны классов	4	-
20	**Обработка ошибок** Введение в обработку ошибок. Генерация исключительных ситуаций с помощью конструкции `throw`. Перехват исключений с помощью конструкции `try/catch`	2	-
21	**Приведение типов** Динамическая идентификация типов. Операторы преобразования типов	1	-

ОПОРНЫЙ КОНСПЕКТ ЛЕКЦИЙ

1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ. ВВЕДЕНИЕ В С++

Программы – это алгоритмы и структуры данных. Известна формула Никлауса Вирта – разработчика языка Pascal:

алгоритмы+структуры данных = программы

Структуры данных представляют исходные данные, промежуточные и конечные результаты.

Алгоритмы – указания, какие действия и в какой последовательности необходимо применять к данным для получения требуемого конечного результата.

1.1. Классификация языков программирования

Языки программирования делятся на две группы:

1. Машинно-зависимые языки. Их можно применять на одной ЭВМ или на ограниченном подмножестве машин с одинаковой архитектурой.

2. Машинно-независимые языки. Их можно использовать на любой ЭВМ. Языки этой группы называют универсальными

Машинно-зависимые языки в зависимости от их близости к машинным языкам делятся на три группы:

- **машинные** (языки нулевого уровня),
- **ассемблерные** (языки первого уровня или языки типа 1:1, т.е. одна ассемблерная команда порождает одну машинную команду),
- **макроассемблеры** (языки второго уровня или языки типа 1:2).

Машинно-независимые языки включают следующие группы языков:

- **императивные** – это языки программирования, управляемые командами или операторами языка. Программа на императивном языке представляет собой последовательность команд (операторов), которые выполняются в порядке их написания. Выполнение каждой команды ведет к изменению состояния компьютера. Основными элементами императивных языков являются переменные и операторы присваивания. Эти языки поддерживают один или несколько итеративных циклов, а также конструкции, позволяющие программировать рекурсивные алгоритмы. *Пример – ALGOL, PL/1, Basic, C/C++, FORTRAN, Ada, Pascal, Java;*

- **функциональные (аппликативные)** – вычисления в основном производятся путем применения функций к заданному набору данных. Разработка программ заключается в создании из простых функций более сложных. Затем эти функции последовательно применяются к начальным данным до тех пор, пока не получится конечный результат. Типичная программа на функциональном языке имеет следующий вид:

$функция_n(\ldots функция_2(функция_1(данные))\ldots)$

Пример – LISP, ML, Miranda, Haskell;

-**декларативные** – это языки программирования, в которых операторы представляют собой объявления или высказывания в символьной логике. Типичный пример таких языков – языки логического программирования. В программах на языках логического программирования соответствующие действия выполняются только при наличии необходимого разрешающего условия. Программа на языке логического программирования схематично выглядит следующим образом:

разрешающее условие 1 -> последовательность операторов 1
разрешающее условие 2 -> последовательность операторов 2
разрешающее условие 3 -> последовательность операторов 3
……
разрешающее условие n -> последовательность операторов n

Порядок выполнения операторов определяется системой реализации правил. *Пример – Prolog, YACC;*

- **объектно-ориентированные** – это языки, реализующие объектно-ориентированную парадигму программирования, основанную на трех ключевых понятиях: абстракция данных (инкапсуляция), наследование и полиморфизм. В основе объектно-ориентированного программирования лежит объектно-ориентированная декомпозиция. Разработка объектно-ориентированных программ заключается в построении иерархии классов, описывающих отношения между объектами, и в определении классов. Вычисления в объектно-ориентированных программах задаются сообщениями, передаваемыми от одного объекта другому.

Пример – Smalltalk, C#, поддержка объектно-ориентированной парадигмы включена в языки *Ada95, Object Pascal, C++*.

1.2. Свойства языков программирования

1. **Понятность (удобочитаемость) конструкций языка** – это свойство, обеспечивающее легкость восприятия программ человеком. Это свойство языка зависит от целого ряда факторов, начиная с выбора ключевых слов и заканчивая возможностью построения модульных программ. Понятность конструкций языка также зависит от выбора такой нотации, которая позволяла бы при чтении текста программы легко выделять основные понятия каждой конкретной части программы, не обращаясь ни к какой другой документации на программу. Реализация требования понятности во многом зависит от программиста.

2. **Надежность** – степень автоматического обнаружения ошибок, которое может быть выполнено транслятором или операционной средой, в которой работает программа. Надежный язык позволяет выявить большинство ошибок во время трансляции программы, а не во время ее выполнения. Принципиальным средством достижения высокой надежности языка, поддерживаемым на этапе трансляции, является система типизации данных.

3. **Гибкость** – сколько возможностей язык предоставляет программисту для выражения всех операций, которые требуются в программе, не прибегая к вставкам ассемблерного кода или другим ухищрениям. Требование гибкости конфликтует с требованием надежности, поэтому необходимо понимать на какие компромиссы приходится идти при решении каждой конкретной задачи.

4. **Простота** – легкость понимания семантики языковых конструкций и запоминания их синтаксиса. Простой язык предоставляет ясный, простой и единообразный набор понятий, которые могут быть использованы в качестве базовых при разработке алгоритмов. При этом желательно иметь минимальное количество различных понятий с как можно более простыми и систематизированными правилами их комбинирования – язык должен обладать свойством **концептуальной целостности**. Концептуальная целостность включает в себя три взаимосвязанных аспекта: экономию, ортогональность и единообразие понятий. Экономия – использование минимального числа понятий. Ортогональность – между понятиями нет взаимного влияния (любые языковые конструкции можно комбинировать по определенным правилам). Единообразие понятий – согласованный, единый подход к описанию и использованию всех понятий.

5. **Естественность** – язык должен содержать такие структуры данных, управляющие структуры и операции, а также иметь такой синтаксис, которые позволяли бы отражать в программе логические структуры, лежащие в основе реализуемого алгоритма.

6. **Мобильность** – возможность переносить программы с одной платформы на другую с относительной легкостью. На мобильность в значительной степени влияет уровень стандартизации языка. Для языков, имеющих стандартное определение (Ada, Pascal, C) все реализации языка должны основываться на этом стандарте.

7. **Стоимость** – суммарная стоимость использования языка программирования складывается из нескольких составляющих: стоимости обучения языку, стоимости создания программы, стоимости трансляции программы, стоимости выполнения программы, стоимости сопровождения программы.

1.3. История и назначение языка C/C++

Язык C (произносится «си») - это универсальный язык программирования, для которого характерны экономичность выражения, современный поток управления и структуры данных, богатый набор операторов. Язык C не предназначается для некоторой специальной области применения. Но отсутствие ограничений и общность языка делают его более удобным и эффективным для многих задач, чем языки, предположительно более мощные.

Язык C, первоначально предназначавшийся для написания операционной системы UNIX на ЭВМ DEC PDP-11, был разработан и реализован на этой системе Деннисом Ритчи. Операционная система, компилятор с языка C и все прикладные программы системы UNIX написаны на C. Коммерческие компиляторы с языка C существуют также на некоторых других ЭВМ, включая IBM

SYSTEM/370, HONEYWELL 6000, INTERDATA 8/32. Язык С, однако, не связан с какими-либо определенными аппаратными средствами или системами, и на нем легко писать программы, которые можно пропускать без изменений на любой ЭВМ, имеющей С-компилятор.

Язык С является универсальным языком программирования. Он тесно связан с операционной системой UNIX, так как был развит на этой системе и UNIX и ее программное обеспечение написано на С. Сам язык, однако, не связан с какой-либо одной операционной системой или машиной; и хотя его называют языком системного программирования, так как он удобен для написания операционных систем, он с равным успехом использовался при написании больших вычислительных программ, программ для обработки текстов и баз данных.

Язык С - это язык относительно низкого уровня. В такой характеристике нет ничего оскорбительного; это просто означает, что С имеет дело с объектами того же вида, что и большинство ЭВМ, а именно, с символами, числами и адресами. Они могут объединяться и пересылаться посредством обычных арифметических и логических операций, осуществляемых реальными ЭВМ.

Аналогично, язык С предлагает только простые, последовательные конструкции потоков управления: проверки, циклы, группирование и подпрограммы, но не мультипрограммирование, параллельные операции, синхронизацию или сопрограммы.

Хотя отсутствие некоторых из этих средств может выглядеть как удручающая неполноценность, но удержание языка в скромных размерах дает реальные преимущества. Так как С относительно мал, он не требует много места для своего описания и может быть быстро выучен. Компилятор с С может быть простым и компактным. Кроме того, компиляторы легко пишутся; при использовании современной технологии можно ожидать написания компилятора для новой ЭВМ за пару месяцев и при этом окажется, что 80 процентов программы нового компилятора будет общей с программой для уже существующих компиляторов. Это обеспечивает высокую степень мобильности языка.

Из-за того, что язык С отражает возможности современных компьютеров, программы на С оказываются достаточно эффективными, так что не возникает побуждения писать вместо этого программы на языке ассемблера. Наиболее убедительным примером этого является сама операционная система UNIX, которая почти полностью написана на С. Из 13000 строк программы системы только около 800 строк самого низкого уровня написаны на ассемблере. Кроме того, по существу все прикладное программное обеспечение системы UNIX написано на С.

Многие из наиболее важных идей С происходят от гораздо более старого языка BCPL , разработанного Мартином Ричардсом. Косвенно язык BCPL оказал влияние на С через язык В, написанный Кеном Томпсоном в 1970 году для первой операционной системы UNIX на ЭВМ PDP-7.

Разработчиком языка С++ является Бьерн Страуструп. В своей работе он опирался на опыт создателей языков Симула, Модула 2, абстрактных типов

данных. Основные работы велись в исследовательском центре компании Bell Labs.

Непосредственный предшественник C++ – язык C с классами – появился в 1979 году, а в 1997 году был принят международный стандарт C++, который фактически подвел итоги его 20-летнего развития. Принятие стандарта обеспечило единообразие всех реализаций языка C++. Не менее важным результатом стандартизации стало то, что в процессе выработки и утверждения стандарта язык был уточнен и дополнен рядом существенных возможностей.

На сегодня стандарт утвержден Международной организацией по стандартизации ISO. Его номер ISO/IEC 14882.

Язык C++ является универсальным языком программирования, в дополнение к которому разработан набор разнообразных библиотек. Поэтому он позволяет решить практически любую задачу программирования. Тем не менее, в силу разных причин (не всегда технических) для каких-то типов задач он употребляется чаще, а для каких-то – реже.

C++ как преемник языка C широко используется в системном программировании. На нем можно писать высокоэффективные программы, в том числе операционные системы, драйверы и т.п. Язык C++ – один из основных языков разработки трансляторов.

Поскольку системное программное обеспечение часто бывает написано на языке C или C++, то и программные интерфейсы к подсистемам операционных систем тоже часто пишут на C++. Соответственно, те программы, даже и прикладные, которые взаимодействуют с операционными системами, написаны на языке C++.

Распределенные системы, функционирующие на разных компьютерах, также разрабатываются на языке C++. Этому способствует то, что у широко распространенных компонентных моделей CORBA и COM есть удобные интерфейсы на языке C++.

Обработка сложных структур данных – текста, бизнес-информации, Internet-страниц и т.п. – одна из наиболее распространенных возможностей применения языка. В прикладном программировании проще назвать те области, где язык C++ применяется мало.

Разработка графического пользовательского интерфейса на языке C++ выполняется в основном тогда, когда необходимо разрабатывать сложные, нестандартные интерфейсы. Простые программы чаще пишутся на языках Visual Basic, Java и т.п.

Программирование для Internet в основном производится на языках Java, VBScript, Perl.

В целом следует отметить, что язык C++ в настоящее время является одним из наиболее распространенных языков программирования в мире.

1.4. Основные парадигмы программирования

Процедурное программирование

Первоначальная (и, возможно, наиболее используемая) парадигма программирования имела вид:

ОПРЕДЕЛИТЕ, КАКИЕ ПРОЦЕДУРЫ ВАМ НУЖНЫ; ИСПОЛЬЗУЙТЕ ЛУЧШИЕ ИЗ ИЗВЕСТНЫХ ВАМ АЛГОРИТМОВ

Ударение делалось на обработку данных с помощью алгоритма, производящего нужные вычисления. Для поддержки этой парадигмы языки предоставляли механизм передачи параметров и получения результатов функций. Литература, отражающая такой подход, заполнена рассуждениями о способах передачи параметров, о том, как различать параметры разных типов, о различных видах функций (процедуры, подпрограммы, макрокоманды) и т.д. Первым процедурным языком был Фортран, а Алгол60, Алгол68, Паскаль и С продолжили это направление.

Модульное программирование

Со временем при проектировании программ акцент сместился с организации процедур на организацию структур данных. Помимо всего прочего это вызвано и ростом размеров программ. Модулем обычно называют совокупность связанных процедур и тех данных, которыми они управляют. Парадигма программирования приобрела вид:

ОПРЕДЕЛИТЕ, КАКИЕ МОДУЛИ НУЖНЫ; ПОДЕЛИТЕ ПРОГРАММУ ТАК, ЧТОБЫ ДАННЫЕ БЫЛИ СКРЫТЫ В ЭТИХ МОДУЛЯХ

Эта парадигма известна также как "принцип сокрытия данных". Если в языке нет возможности сгруппировать связанные процедуры вместе с данными, то он плохо поддерживает модульный стиль программирования. Теперь метод написания «хороших» процедур применяется для отдельных процедур модуля.

Язык Модула-2 прямо поддерживает эту парадигму, тогда как С только допускает такой стиль.

Поскольку данные есть единственная вещь, которую хотят скрывать, понятие скрытия данных тривиально расширяется до понятия скрытия информации, т.е. имен переменных, констант, функций и типов, которые тоже могут быть локальными в модуле. Хотя С++ и не предназначался специально для поддержки модульного программирования, классы поддерживают концепцию модульности.

Абстракция данных

Модульное программирование предполагает группировку всех данных одного типа вокруг одного модуля, управляющего этим типом. Тип, реализуемый управляющим им модулем, по многим важным аспектам существенно отличается от встроенных типов. Такие типы не получают той поддержки со стороны транслятора (разного вида контроль), которая обеспечивается для встроенных типов. Проблема здесь в том, что программа формулируется в терминах небольших (одно-два слова) дескрипторов объектов, а не в терминах самих объектов. Это означает, что транслятор не сможет отловить глупые, очевидные ошибки.

В языках Ада, C++ и подобных им эта трудность преодолевается благодаря тому, что пользователю разрешается определять свои типы, которые трактуются в языке практически так же, как встроенные. Такие типы обычно называют **абстрактными типами данных**, хотя лучше, пожалуй, их называть просто пользовательскими. Более строгим определением абстрактных типов данных было бы их математическое определение. Если бы удалось его дать, то, что называется в программировании типами, было бы конкретным представлением действительно абстрактных сущностей. Парадигму программирования можно выразить теперь так:

ОПРЕДЕЛИТЕ, КАКИЕ ТИПЫ ВАМ НУЖНЫ; ПРЕДОСТАВЬТЕ ПОЛНЫЙ НАБОР ОПЕРАЦИЙ ДЛЯ КАЖДОГО ТИПА

Если нет необходимости в разных объектах одного типа, то стиль программирования, суть которого сводится к скрытию данных, и следование которому обеспечивается с помощью концепции модульности, вполне адекватен этой парадигме.

Абстрактный тип данных определяется как некий «черный ящик». После своего определения он никак не взаимодействует с программой. Его никак нельзя приспособить для новых целей, не меняя определения. В этом смысле это негибкое решение

Объектно-ориентированное программирование

Пусть, например, нужно определить для графической системы тип `shape` (фигура). Пока считаем, что в системе могут быть такие фигуры: окружность (`circle`), треугольник (`triangle`) и квадрат (`square`). Пусть необходимо создать в программе функцию `draw()` для рисования любой фигуры. В ней нужно учесть все возможные фигуры, какие только есть. Поэтому она дополняется новыми операторами, как только в системе появляется новая фигура. Плохо то, что после определения новой фигуры нужно проверить и, возможно, изменить все старые операции класса. Поэтому, если исходный текст каждой операции класса недоступен, то ввести новую фигуру в систему просто

невозможно. Появление любой новой фигуры приводит к манипуляциям с текстом каждой существенной операции класса. Требуется достаточно высокая квалификация, чтобы справиться с этой задачей, но все равно могут появиться ошибки в уже отлаженных частях программы, работающих со старыми фигурами. Возможность выбора представления для конкретной фигуры сильно сужается, если требовать, чтобы все ее представления укладывались в уже заданный формат, специфицированный общим определением фигуры (т.е. определением типа shape).

Проблема состоит в том, что в программе не различаются общие свойства фигур (например, фигура имеет цвет, ее можно нарисовать и т.д.) и свойства конкретной фигуры (например, окружность - это такая фигура, которая имеет радиус, она изображается с помощью функции, рисующей дуги и т.д.). Суть **объектно-ориентированного программирования** в том, что оно позволяет выражать эти различия и использует их. Язык, который имеет конструкции для выражения и использования подобных различий, поддерживает объектно-ориентированное программирование. Все другие языки не поддерживают его. Здесь основную роль играет механизм наследования, заимствованный из языка Симула.

Теперь парадигма программирования формулируется так:

ОПРЕДЕЛИТЕ, КАКОЙ КЛАСС ВАМ НЕОБХОДИМ; ПРЕДОСТАВЬТЕ ПОЛНЫЙ НАБОР ОПЕРАЦИЙ ДЛЯ КАЖДОГО КЛАССА; ОБЩНОСТЬ КЛАССОВ ВЫРАЗИТЕ ЯВНО С ПОМОЩЬЮ НАСЛЕДОВАНИЯ

Если общность между классами отсутствует, вполне достаточно абстракции данных. Насколько применимо объектно-ориентированное программирование для данной области приложения определяется степенью общности между разными типами, которая позволяет использовать наследование и виртуальные функции. В некоторых областях, таких, например, как интерактивная графика, есть широкий простор для объектно-ориентированного программирования. В других областях, в которых используются традиционные арифметические типы и вычисления над ними, трудно найти применение для более развитых стилей программирования, чем абстракция данных. Здесь средства, поддерживающие объектно-ориентированное программирование, очевидно, избыточны.

Нахождение общности среди отдельных типов системы представляет собой нетривиальный процесс. Степень такой общности зависит от способа проектирования системы. В процессе проектирования выявление общности классов должно быть постоянной целью. Она достигается двумя способами: либо проектированием специальных классов, используемых как "кирпичи" при построении других, либо поиском похожих классов для выделения их общей части в один базовый класс.

1.5. Первая программа на C++

Далее представлена первая программа на C++, которая выводит на экран строчку «Hello, world!» (рис. 1.1)

```
//Первая программа на C++

#include <iostream.h>
main()
{
     cout << "Hello, World!\n";
     return 0;
}
```

Рис. 1.1. Первая программа на C++

Строка

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

является директивой препроцессора, т.е. сообщением препроцессору C++. Строки, начинающиеся с #, обрабатываются препроцессором перед компиляцией программы. Данная строка дает указание препроцессору включить в программу содержание заголовочного файла потока ввода/вывода `iostream.h`.

Остальная часть программы

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

определяет функцию, названную `main`. Каждая программа должна содержать функцию с именем `main`, и работа программы начинается с выполнения этой функции, даже если это не первая функция программы.

Круглые скобки после `main` показывают, что `main` – программный блок, называемый функцией.

Левая фигурная скобка `{` должна начинать тело каждой функции. Соответствующая правая фигурная скобка должна заканчивать каждую функцию.

Строка

```
cout << "Hello, World!\n";
```

является командой компьютеру напечатать на экране строку символов, заключенную в кавычки. Полная строка, включающая `cout`, операцию `<<`, строку `"Hello, World!\n"` и точку с запятой (`;`), называется оператором. Каждый оператор должен заканчиваться точкой с запятой. Операция `<<` («поместить в») пишет свой первый аргумент во второй (в данном случае, строку `"Hello, world\n"` в стандартный поток вывода `cout`). Строка - это последовательность символов, заключенная в двойные кавычки. В строке символ обратной

косой \, за которым следует другой символ, обозначает один специальный символ; в данном случае, `\n` является символом новой строки. Таким образом, выводимые символы состоят из `Hello, world` и перевода строки.

Некоторые неграфические символы, одиночная кавычка `'` и обратная косая `\`, могут быть представлены в соответствии с таблицей escape-последовательностей (табл. 1).

Операция `<<` называется «поместить в поток». При выполнении этой программы значение справа от оператора помещается в поток вывода.

Строка

```
return 0;
```

показывает, что программа успешно закончена. Включается в конце каждой функции main. Ключевое слово return – один из нескольких способов выхода из функции.

Таблица 1

Непечатные символы

Описание символа	**Код символа**	**Специальная последовательность**
Символ новой строки	**NL(LF)**	**\n**
Горизонтальная табуляция	**NT**	**\t**
Вертикальная табуляция	**VT**	**\v**
Возврат на шаг	**BS**	**\b**
Возврат каретки	**CR**	**\r**
Перевод формата	**FF**	**\f**
Обратная косая	****	****
Одиночная кавычка (апостроф)	**'**	**\'**
Двойные кавычки	**“**	**\”**
Звонок		**\a**

2. ТИПЫ ДАННЫХ

2.1. Понятие переменной и объявление переменных

Имя (идентификатор) – это строка символов, используемая для обозначения некоторой сущности в программе. Такими сущностями могут быть переменные, типы, метки, функции и т.д. В общем случае идентификаторы не имеют смысла, а используются только в качестве имен программных объектов или их атрибутов.

В C/C++ идентификатор должен начинаться с латинской буквы или нижнего подчеркивания, и содержать латинские буквы, цифры и знаки подчеркивания.

Переменная – это объект данных, который явным образом определен и именован в программе. Переменные характеризуются с помощью следующих атрибутов: имя, адрес, значение, тип, время жизни, область видимости.

Имя переменной – это идентификатор, используемый в программах для ссылки на значение переменной.

Адрес переменной – это адрес области памяти, с которой связана данная переменная.

Значение переменной – это содержимое ячейки или совокупности ячеек памяти, связанных с данной переменной.

Тип переменной связывает переменную с множеством значений, которые она может принимать.

Время жизни – это время, в течение которого переменная связана с определенной областью памяти.

Область видимости – это блок программы, из которого можно обратиться к этой переменной.

Объявление переменных

Перед использованием в программе любой объект данных должен быть объявлен. Оператор объявления сообщает компилятору информацию об идентификаторах и типах данных, назначаемых объектам данных, и о предполагаемом времени жизни этих объектов (глобальная или локальная переменная, переменная - член класса, статическая переменная и т.п.).

В языке C++ для объявления новых имен в текущей области видимости предназначаются **операторы объявления.**

Например:

```
int i, j; float m, n;
```

Оператор объявления в языке C++ может указываться в любом допустимом месте программы.

В языке C++ каждый оператор объявления завершается символом конца оператора (точка с запятой). В операторе объявления может объявляться несколько объектов данных одного типа, перечисляемых через запятую. Любой

оператор объявления начинается с ключевого слова или идентификатора, указывающего тип объявляемого объекта.

Встроенные типы данных

Каждое имя и каждое выражение обязаны иметь тип. Именно тип определяет операции, которые могут выполняться над ними. Например, в описании

```
int inch;
```

говорится, что `inch` имеет тип `int`, т.е. `inch` является целой переменной.

Таблица 2

Размер памяти, выделяемой под встроенные типы данных

Тип данных	Байт	Диапазон
long double	?	3.4e-4932..3.4e+4932
double	8	1.7e-308..1.7e+308
float	4	3.4e-38..3.4e+38
unsigned long long	8	0..18 446 744 073 709 551 615
long long int	8	-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807
unsigned long int	4	0..4 294 967 295
long int	4	-2 147 483 648 .. 2 147 483 647
int	?	?
unsigned short int	2	0..65535
short int	2	-32 768 .. 32 767
unsigned char	1	0..255
char	1	-128 .. 127

Описание - это оператор, который вводит имя в программу. В описании указывается тип имени. Тип, в свою очередь, определяет, как правильно использовать имя или выражение.

Типы данных в порядке следования от высших типов к низшим с количеством занимаемых типом байт (в 32-разрядных системах) приведены в табл. 2.

2.2. Константы и перечисления

Константные переменные

Если переменная объявлена с ключевым словом `const`, значит, она не должна меняться. После определения константной переменной уже нельзя изменить ее значение или передать ее в качестве аргумента функции, которая не гарантирует ее неизменности. Рассмотрим простой пример с константной целой переменной (рис. 2.1.).

```
const int j = 17; // Целая константа
j = 29; // Нельзя, значение не должно меняться
const int i; // Нельзя, отсутствует начальное значение
```

Рис. 2.1. Работа с константой

Третья строка неверна, поскольку в ней компилятору предлагается определить случайную переменную, которую никогда не удастся изменить, - довольно странный генератор случайных целых констант.

Целое число может быть непосредственно записано в программе в виде константы. Запись чисел соответствует общепринятой нотации. Примеры целых констант: `0, 125, -37`. По умолчанию целые константы принадлежат к типу `int`. Если необходимо указать, что целое число — это константа типа `long` , можно добавить символ `L` или `l` после числа. Если константа беззнаковая, т.е. относится к типу `unsigned long` или `unsigned int` , после числа записывается символ `U` или `u`. Например:

`34U, 700034L, 7654ul`.

Кроме стандартной десятичной записи, числа можно записывать в восьмеричной или шестнадцатеричной системе счисления. Признаком восьмеричной системы счисления является цифра `0` в начале числа. Признаком шестнадцатеричной — `0x` или `0X` перед числом. Для шестнадцатеричных цифр используются латинские буквы от `A` до `F` (неважно, большие или маленькие).

Таким образом, фрагмент программы

```
const int x = 240;
const int y = 0360;
const int z = 0xF0;
```

определяет три целые константы `x`, `y` и `z` с одинаковыми значениями

Перечисления

Достаточно часто в программе вводится тип, состоящий лишь из нескольких заранее известных значений. Например, в программе используется переменная, хранящая величину, отражающую время суток, и решено, что будем различать ночь, утро, день и вечер. Конечно, можно договориться обозна-

чить время суток числами от 1 до 4. Но, во-первых, это не наглядно. Во-вторых, что более существенно, очень легко сделать ошибку и, например, использовать число 5, которое не соответствует никакому времени дня. Гораздо удобней и надежнее определить набор значений с помощью типа `enum` языка C++:

```
enum DayTime { morning, day, evening, night };
```

Теперь можно определить переменную

```
DayTime current;
```

которая хранит текущее время дня, а затем присваивать ей одно из допустимых значений типа `DayTime`:

```
current = day;
```

Контроль, который осуществляет компилятор при использовании в программе этой переменной, гораздо более строгий, чем при использовании целого числа.

Для наборов определены операции сравнения на равенство (==) и неравенство (!=) с атрибутами этого же типа, т.е.

```
if (current != night)
     // выполнить работу
```

Внутреннее представление значений набора – целые числа. По умолчанию элементам набора соответствуют последовательные целые числа, начиная с 0. Этим можно пользоваться в программе. Во-первых, можно задать, какое число какому атрибуту набора будет соответствовать (рис. 2.2).

```
     //каждому атрибуту задается число
enum { morning = 4, day = 3, evening = 2, night = 1 };
     // последовательные числа начиная с 1
enum { morning = 1, day, evening, night };
     // используются числа 0, 2, 3 и 4
enum { morning, day = 2, evening, night };
```

Рис. 2.2. Различные наборы чисел в перечислении

Во-вторых, атрибуты наборов можно использовать в выражениях вместо целых чисел. Преобразования из набора в целое и наоборот разрешены (рис. 2.3).

```
int main(){
     enum season{spring=1, summer, autumn=1, winter};
     season s;
     s = season(4);
     cout<< s<<endl;
     s = winter;
     cout<< s<<endl;
     if(s==winter)  cout<<"Winter!"<<endl;
     return 0;}
```

Рис. 2.3. Программа, иллюстрирующая работу с перечислениями

2.3. Операции и выражения

Основные операции языка C++ приведены в табл. 3. Использование этих операций более подробно будет указано далее. Операции располагаются в порядке убывания приоритета.

Таблица 3

Операции языка C++

Операция	Арность	Ассоциативность	Описание	Использование	Приоритет
1	2	3	4	5	6
`::`	**унарная**	**правая**	*получение доступа к глобальной переменной*	`:: имя`	**17**
`::`	**бинарная**	**левая**	*разрешение области видимости*	`имя_класса :: имя_члена`	**17**
`->`	**бинарная**	**левая**	*селектор члена*	`указатель->член`	**16**
`.`	**бинарная**	**левая**	*селектор члена*	`объект.член`	**16**
`[]`	**бинарная**	**левая**	*индекс элемента массива*	`имя_массива[выражение]`	**16**
`()`	**бинарная**	**левая**	*вызов функции*	`имя_функции (список_параметров)`	**16**
`()`	**бинарная**	**левая**	*создание типа*	`type(список_выражений)`	**16**
`++`	**унарная**	**правая**	*постфиксная инкрементация*	`левое_значение++`	**16**
`--`	**унарная**	**правая**	*постфиксная декрементация*	`левое_значение--`	**16**
`typeid`	**унарная**	**правая**	*идентификация типов*	`typeid(тип)` `typeid(выражение)`	**16**

Продолжение табл. 3

1	2	3	4	5	6
dynamic_cast	унарная	правая	*контролируемое преобразование типов*	dynamic_cast <тип> (выражение)	16
static_cast	унарная	правая	*контролируемое преобразование типов*	static_cast <тип> (выражение)	16
reinterpret_cast	унарная	правая	*неконтролируемое преобразование типов*	reinterpret_cast <тип> (выражение)	16
const_cast	унарная	правая	*преобразование константы*	const_cast <тип> (выражение)	16
sizeof	унарная	правая	*получение размера типа*	sizeof(тип)	15
++	унарная	правая	*префиксная инкрементация*	++правое_значение	15
--	унарная	правая	*префиксная декрементация*	--правое_значение	15
~	унарная	правая	*побитовое дополнение (побитовое отрицание)*	~выражение	15
!	унарная	правая	*логическое отрицание*	!выражение	15
+	унарная	правая	*унарный плюс*	+выражение	15
-	унарная	правая	*унарный минус*	-выражение	15
*	унарная	правая	*косвенная адресация или разыменование*	*указатель	15
&	унарная	правая	*взятие адреса*	&значение	15

1	2	3	4	5	6
( )	**унарная**	**правая**	***явное приведение типа***	**`(тип)выражение`**	**15**

Продолжение табл. 3

1	2	3	4	5	6
new	унарная	правая	*выделение памяти*	new типа new тип (список_выражений) new (список_выражений) тип new (список_выражений) тип (список_выражений)	15
delete	унарная	правая	*очистка памяти*	delete указатель delete [] указатель	15
->*	бинарная	левая	*селектор члена*	указатель->*-указатель_на_член	14
.*	бинарная	левая	*селектор члена*	объект.*-указатель_на_член	14
*	бинарная	левая	*умножение*	выражение*выражение	13
/	бинарная	левая	*деление*	выражение/выражение	13
%	бинарная	левая	*остаток от деления*	выражение%выражение	13
+	бинарная	левая	*сложение*	выражение+выражение	12
-	бинарная	левая	*вычитание*	выражение-выражение	12
<<	бинарная	левая	*побитовый сдвиг влево*	выражение<<выражение	11

Продолжение табл. 3

1	2	3	4	5	6
>>	**бинарная**	**левая**	***побитовый сдвиг вправо***	**выражение>>выражение**	**11**
<	**бинарная**	**левая**	***меньше***	**выражение<выражение**	**10**
<=	**бинарная**	**левая**	***меньше или равно***	**выражение<=выражение**	**10**
>	**бинарная**	**левая**	***больше***	**выражение>выражение**	**10**
>=	**бинарная**	**левая**	***больше или равно***	**выражение>=выражение**	**10**
==	**бинарная**	**левая**	***равно***	**выражение==выражение**	**9**
!=	**бинарная**	**левая**	***не равно***	**выражение!=выражение**	**9**
&	**бинарная**	**левая**	***побитовое логическое И***	**выражение&выражение**	**8**
^	**бинарная**	**левая**	***побитовое логическое исключающее ИЛИ***	**выражение^выражение**	**7**
|	**бинарная**	**левая**	***побитовое логическое ИЛИ***	**выражение|выражение**	**6**
&&	**бинарная**	**левая**	***логическое И***	**выражение&&выражение**	**5**
||	**бинарная**	**левая**	***логическое ИЛИ***	**выражение||выражение**	**4**
?:	**тернарная**	**левая**	***условная операция***	**выражение?выражение:выражение**	**3**

Окончание табл. 3

1	2	3	4	5	6
=	бинарная	левая	*присваивание*	значение=выражение	2
*=	бинарная	левая	*умножение и присваивание*	значение*=выражение	2
/=	бинарная	левая	*деление и присваивание*	значение/=выражение	2
%=	бинарная	левая	*вычисление остатка от деления и присваивание*	значение%=выражение	2
+=	бинарная	левая	*сложение и присваивание*	значение+=выражение	2
-=	бинарная	левая	*вычитание и присваивание*	значение-=выражение	2
<<=	бинарная	левая	*сдвиг влево и присваивание*	значение<<=выражение	2
>>=	бинарная	левая	*сдвиг вправо и присваивание*	значение>>=выражение	2
&=	бинарная	левая	*логическое побитовое И и присваивание*	значение&=выражение	2
|=	бинарная	левая	*логическое побитовое ИЛИ и присваивание*	значение|=выражение	2
^=	бинарная	левая	*логическое побитовое исключающее ИЛИ и присваивание*	значение^=выражение	2
,	бинарная	левая	*запятая*	выражение,выражение	1

Побитовые логические операции

В битовых операциях целое число рассматривается как строка битов (нулей и единиц при записи числа в двоичной системе счисления или разрядов машинного представления).

Таблица 4

Побитовые операции

Операция	Результат	Объяснение операции
`3 | 1`	3	побитовое ИЛИ
`4 & 7`	4	побитовое И
`4 ^ 7`	3	побитовое исключающее ИЛИ
`0 & 0xF`	0	побитовое И
`~0x00F0`	0xFF0F	поразрядное дополнение (отрицание)
`8<<2`	32	побитовый сдвиг влево
`192>>4`	12	побитовый сдвиг вправо

Поразрядная операция ИЛИ, обозначаемая знаком `|`, выполняет операцию ИЛИ над каждым индивидуальным битом двух своих операндов. Например, 1 `|` 2 в результате дают 3, поскольку в двоичном виде 1 это 01, 2 – это 10, соответственно операция ИЛИ дает 11 или 3 в десятичной системе (нули слева опущены).

Аналогично выполняются поразрядные операции И(&), ИСКЛЮЧАЮЩЕЕ ИЛИ(^) и отрицание или дополнение (~).

Операция сдвига (<< - сдвиг влево или >> - сдвиг вправо) перемещает двоичное представление левого операнда на количество битов, соответствующее значению правого операнда. Например, двоичное представление короткого целого числа 3 – 0000000000000011. Результатом операции 3 << 2 (сдвиг влево) будет двоичное число 0000000000001100 или, в десятичной записи, 12. Аналогично, сдвинув число 9 (в двоичном виде 0000000000001001) на 2 разряда вправо (записывается 9 >> 2) получим 0000000000000010, т.е. 2.

При сдвиге влево число дополняется нулями справа. При сдвиге вправо бит, которым дополняется число, зависит от того, знаковое оно или беззнаковое. Для беззнаковых чисел при сдвиге вправо они всегда дополняются нулевым битом. Если же число знаковое, то значение самого левого бита числа используется для дополнения. Это объясняется тем, что самый левый бит как раз и является знаком — 0 означает плюс и 1 означает минус. Таким образом, если

```
short x = 0xFF00;
unsigned short y = 0xFF00;
```

то результатом x >> 2 будет 0xFFC0 (двоичное представление 1111111111000000), а результатом y >> 2 будет 0x3FC0 (двоичное представление 0011111111000000).

Операция сдвига влево аналогична умножению на соответствующую степень двойки, а сдвиг влево – делению.

Комментарии

Хорошая программа должна содержать комментарии для документирования. Комментарии могут начинаться и заканчиваться в любом месте программы. В C++ используется два вида комментариев: однострочные и многострочные.

Однострочные комментарии начинаются с символа «`//`» и заканчиваются в конце текущей строки.

Многострочные комментарии начинаются с символов «`/*`» и заканчиваются парой «`*/`». Вложенных комментариев не бывает. Независимо от количества символов «`/*`», следующих друг за другом, их действие заканчивается на ближайшем символе «`*/`».

3. ОПЕРАТОРЫ И ВЫРАЖЕНИЯ

Понятие оператора

Пустой оператор состоит из одного символа – «`;`». Он не выполняет никаких действий. Используется как заполнитель в других, более сложных операторах.

Операторы выражения представляют собой просто выражения, за которыми следуют точка с запятой, или запятая, или некоторый другой контекст.

Составной оператор (блок) состоит из определений объектов (данных), за которыми следует последовательность операторов. Блок заключается в фигурные скобки.

Оператор `return`

Имеет две формы:

```
return;
return выражение;
```

Первая обеспечивает передачу управления из текущей функции, не имеющей возвращаемого значения, на оператор, непосредственно следующий за вызовом функции.

Вторая форма кроме этого еще и возвращает значение в место вызова. Тип выражения в операторе return и тип возвращаемого значения должны совпадать.

Оператор return может отсутствовать в функции, но лучше его явно указывать.

Оператор `if` (ЕСЛИ)

Этот оператор позволяет изменять порядок выполнения программы в зависимости от истинности условия. Структура `if` является структурой с одним входом и одним выходом. Структура `if/else (ЕСЛИ/ИНАЧЕ)` предусматривает действия в случае ложного условия (см. рис. 3.1)

```
if (x>y)
          cout << 1 << endl;
     else if (x>z)
          cout << 2 << endl;
     else if (y>z)
          cout << 3 << endl;
     else
          cout << 4 << endl;
```

Рис. 3.1 Использование структуры if/else

Структуры выбора `if` обычно предполагает наличие в своем теле только одного оператора. Чтобы включить несколько операторов в тело структуры, заключите их в фигурные скобки `{}`

Оператор `while` (ПОКА)

Задача. Найти среднее арифметическое оценок группы по предмету. Пользователь вводит оценки с клавиатуры, признак конца ввода оценок - -1. (рис 3.2.)

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
     int counter, grade, total;
     float average;
     total = 0;
     counter = 0;
     cout << "Enter the grade or -1 for stopping: ";
     cin >> grade;
     while (grade != -1)
     {
          total = total + grade;
          counter++;
          cout << "Enter the grade or -1 for stopping: ";
          cin >> grade;
     }
     average = (float)total / counter;
     cout << "The average grade is " << average << endl;
     return 0;
```

```
}
```

Рис. 3.2. Поиск среднего арифметического оценок

Целые значения не всегда выражаются целыми числами. Часто среднее значение имеет величину типа `7,2` или `-93,5`. Подобные числа записываются как числа с плавающей точкой и представляются типом данных `float`. Переменная `average` объявлена как переменная типа `float`, чтобы учесть дробную часть результатов вычислений. Однако результат вычисления `total/counter` является целым числом, поскольку и `total` и `counter` – переменные целого типа. Деление двух целых чисел осуществляется как целочисленное деление, при котором любая дробная часть-результат теряется. Поскольку сначала осуществляется деление, дробная часть потеряется прежде, чем результат будет записан в переменную `average`. Чтобы осуществлять над целыми числами вычисления с плавающей запятой, надо создавать для вычислений временные величины с плавающей запятой. В C++ для решения этой задачи используется **унарная операция приведения к типу**. Оператор

```
average = (float)total / counter;
```

включает операцию приведения к типу, которая создает временную копию с плавающей запятой своего оператора `total`. Величина, сохраняемая в `total`, остается целой. А вычисления теперь сводятся к делению величины с плавающей запятой на целую величину.

Оператор `do/while` (ВЫПОЛНЯТЬ/ПОКА)

Структура `do/while` похожа на структуру `while`. В структуре `while` условие продолжения цикла проверяется в начале цикла, до того, как выполняется тело цикла. В структуре `do/while` проверка условия продолжения циклов производится после того, как тело цикла выполнено, т.е. тело цикла будет выполнено по крайней мере один раз.

Задача. Определить количество цифр в числе, введенном пользователем. (рис. 3.3.)

```
#include <iostream.h>
int main()
{
     int num =0, step = 0, del=1;
     cout<<"Введите число:\t";
     cin>> num;
     do
     {
          del*=10;
          step++;
     } while(num/del);
     cout<<"Количество цифр:\t"<< step<<endl;
```

```
        return 0;
}
```

Рис. 3.3. Определение количества цифр в числе

Оператор `for` (ЦИКЛ)

Структура повторения `for` содержит все элементы, необходимые для повторения, управляемого счетчиком. Инструкция цикла состоит из трех операторов. Первый оператор задает начальное значение переменной цикла, второй содержит условие продолжения цикла, а третий - шаг цикла.

Задача. Вывести на экран значения степеней двойки от 0 до 16.

```
#include <iostream.h>
int main()
{
     int step = 1;
     cout<<"Степень:\t\tЗначение:"<<endl;
     for(int i = 0; i<=16; i++)
     {
          cout<<i<<"\t\t"<<step<<endl;
          step*=2;
     }
     return 0;
}
```

Рис. 3.4. Вывод на экран таблицы степеней двойки

Оператор множественного выбора `switch`

Программа в следующем примере осуществляет преобразование дюймов в сантиметры и сантиметров в дюймы; предполагается, что будут указаны единицы измерения вводимых данных, добавляя `i` для дюймов и с для сантиметров (через пробел). (рис. 3.5.)

```
#include <iostream.h>
int main(){
     const float fac = 2.54;
     float x, in, cm; char ch = 0;
     cout << "введите длину: "; cin >> x >> ch;
     switch (ch) {
     case 'i':
          in = x;
          cm = x*fac; break;
     case 'c':
          in = x/fac;
          cm = x; break;
     default:
          in = cm = 0;
          break;
```

```
        }
    }
```

Рис. 3.5. Программа, выполняющая преобразования дюймов в сантиметры и наоборот

Оператор `break` вызывает передачу программного управления на первый оператор после структуры `switch`. Если не использовать `break`, тогда каждый раз, когда одно из условий `case` удовлетворяется, будут выполняться операторы всех последующих меток `case`. Если ни одно условие не выполнено, то выполняются действия после метки `default`.

Структура `switch` отличается от всех других структур тем, что при нескольких действиях после `case` не требуется заключать их в фигурные скобки.

Операторы `break` и `continue`

Операторы `break` и `continue` изменяют поток управления. Когда оператор `break` выполняется в структурах `while`, `for`, `do/while` или `switch`, происходит немедленный выход из структуры. Программа продолжает выполнение с первого оператора после структуры. Обычное назначение оператора `break` – досрочно прерывать цикл или пропустить оставшуюся часть структуры `switch`.

Оператор `continue` в структурах `while`, `for` или `do/while` вызывает пропуск оставшейся части тела структуры и начинается выполнение последующей итерации цикла. В структурах `while` и `do/while` немедленно после выполнения оператора `continue` производится проверка условия продолжения цикла. В структуре `for` выполняется выражение приращения, а затем осуществляется проверка условия продолжения.

4. МАССИВЫ

4.1.Определение, объявление и инициализация массивов

Массив – это последовательная группа ячеек памяти, имеющих одинаковое имя и одинаковый тип. Чтобы сослаться на элемент массива, необходимо указать имя массива и номер позиции отдельного элемента массива.

На любой элемент массива можно сослаться, указывая имя массива и номер позиции элемента, заключенный в квадратные скобки. Первый элемент каждого массива – нулевой. Номер позиции, указанный в квадратных скобках, называют **индексом**. Индекс должен быть целым числом или целым выражением. Если программа использует выражение в качестве индекса, то выражение вычисляется с целью определения индекса.

Важно понимать различие между «седьмым элементом массива» и «элементом массива семь». Седьмой элемент массива имеет индекс 6, тогда как элемент массива семь имеет индекс 7.

Чтобы разделить значение седьмого элемента массива `arr` на 2 и записать результат в переменную `x`, надо записать:

```
x = arr[6] / 2;
```

Объявления и инициализация массивов в программе

Массивы занимают область в памяти. Программист указывает тип каждого элемента, количество элементов, требуемое каждым массивом, и компилятор может зарезервировать соответствующий объем памяти.

```
int c[12];
int b[100], x[27];
```

Элементам массива должны быть заданы какие-то начальные значения:

```
int n[10];
for (int i = 0; i<10; i++) n[i] = 0;
```

Элементам массива можно присваивать начальные значения (инициализировать их) в объявлении массива с помощью следующего за объявлением списка (заключенного в фигурные скобки):

```
int n[5] = {5,4,3,2};
```

Если начальных значений меньше, чем элементов в массиве, оставшиеся элементы автоматически получают нулевые значения.

Если размер массива не указан в объявлении со списком инициализации, то количество элементов массива будет равно количеству элементов в списке начальных значений:

```
int n[ ] = {5,4,3,2,1};
const int arraySize = 10;
int s[arraySize];
```

При объявлении массивов можно использовать только константы.

Символьному массиву можно также задать в качестве начального значения список отдельных символьных констант, указанных в списке инициализации. Например,

```
char string1 [ ] = “first”;
char string2 [ ] ={‘f’, ‘i’, ‘r’, ‘s’, ‘t’, ‘\0’}
```

Эти объявления эквивалентны. Оба массива содержат ШЕСТЬ символов, потому что последним символом в строке всегда является нулевой символ – признак завершения строки.

4.2. Сортировка массивов

Пузырьковая сортировка

Задача. В программе задан массив из 10 элементов. Необходимо отсортировать его по возрастанию и вывести отсортированный массив на экран. Использовать пузырьковую сортировку (см. рис. 4.1.).

Данная сортировка получила название **пузырьковая сортировка** или сортировка погружением, потому что наименьшее значение постепенно

«всплывает», продвигаясь к вершине (началу) массива, подобно пузырьку воздуха в воде, тогда как наибольшее значение погружается на дно (конец массива).

```
int main(){
     const int arraySize = 10;
     int a[arraySize] = {2,6,4,8,10,12,89,68,45,37};
     int hold;
     cout << "In natural order" << endl;
     for (int i = 0; i< arraySize; i ++)
          cout << a[i] << "\t";
     cout << endl;
     //Bubbble-sort**************************************
     for( int pass = arraySize-1; pass >0; pass --)
          for (i = 0; i < pass; i++)
               if (a[i] > a[i+1])
               {
                    hold = a[i];
                    a[i] = a[i+1];
                    a[i+1] = hold;
               }
     //***********************************************
     cout << "In right order" << endl;
     for (i = 0; i< arraySize; i ++)
          cout << a[i] << "\t";
     cout << endl;
}
```

Рис. 4.1. Программа, иллюстрирующая работу пузырьковой сортировки

Сортировка осуществляется с помощью вложенного цикла `for`. Перестановка выполняется тремя присваиваниями, с помощью переменной `hold`. Двумя присваиваниями сортировку выполнить нельзя, потому что после первого присваивания одно из значений будет утеряно, а обе переменные будут иметь одинаковое значение.

Сортировка вставками

Задача. В программе задан массив из 10 элементов. Необходимо отсортировать его по возрастанию и вывести отсортированный массив на экран. Для сортировки использовать алгоритм сортировки массива вставками (см. рис. 4.2.).

Вставка происходит следующим образом: в конце нового массива выделяется свободная ячейка, далее анализируется элемент, стоящий перед пустой ячейкой (если, конечно, пустая ячейка не стоит на первом месте), и если этот элемент больше вставляемого, то элемент сдвигается в свободную ячейку (при этом на том месте, где он стоял, образуется пустая ячейка) и затем сравнивается

следующий элемент. Так можно получить ситуацию, когда элемент перед пустой ячейкой меньше вставляемого, или пустая ячейка стоит в начале массива. Помещаем вставляемый элемент в пустую ячейку. Таким образом, по очереди вставляем все элементы исходного массива.

```
int main()
{
     const int arraySize = 10;
     int a[arraySize] = {2,6,4,8,10,12,89,68,45,37};
     int b[arraySize];
     cout << "In natural order" << endl;
     for (int i = 0; i< arraySize; i ++)
          cout << a[i] << "\t";
     cout << endl;
     //Insert sorting**********************
     for (i = 0; i< arraySize; i++)
     {
          int j = i;
          while ((j>0) && (b[j-1]>a[i]))
          {
               b[j]=b[j-1];
               j=j-1;
          }
          b[j] = a[i];
     }
//****************************************
     cout << "In right order" << endl;
     for (i = 0; i< arraySize; i ++)
          cout << b[i] << "\t";
     cout << endl;
}
```

Рис. 4.2. Программа, иллюстрирующая работу сортировки вставками

4.3. Поиск в массивах

Линейный поиск

```
int main(){
     const int arraySize = 10;
     int a[arraySize] = {13,6,4,8,10,12,89,68,45,37};
     int searchKey, element;
     cout << "Enter search key: ";
     cin >> searchKey;
     element = -1;
     for (i = 0; i< arraySize; i ++)
          if(a[i]==searchKey)
               element = i;
     if (element != -1)
          cout << "Number is " << element << endl;
```

```
else
      cout << "No such element" << endl;}
```

Рис. 4.3. Линейный поиск в массиве

Линейный поиск сравнивает каждый элемент массива с ключом поиска. Поскольку массив может быть неупорядочен, вполне вероятно, что отыскиваемое значение окажется первым же элементом массива. Но в среднем программа должна сравнить с ключом поиска половину элементов массива (рис. 4.3.).

Метод линейного поиска хорошо работает для небольших или для несортированных массивов. Однако для больших массивов линейный поиск неэффективен. Если массив отсортирован, можно использовать высокоэффективный метод двоичного поиска.

Двоичный поиск

Алгоритм двоичного поиска исключает половину еще непроверенных элементов массива после каждого сравнения. Алгоритм определяет местоположение среднего элемента массива и сравнивает его с ключом поиска. Если они равны, то ключ поиска найден, и выдается индекс этого элемента. В противном случае задача сокращается на половину элементов массива. Если ключ поиска меньше, чем средний элемент массива, то дальнейший поиск осуществляется в первой половине массива, а если больше, то во второй половине.

4.4. Многомерные массивы

Массивы в C++ могут иметь много индексов. Обычным представлением многомерных массивов являются таблицы значений, содержащие информацию в строках и столбцах. Чтобы определить отдельный табличный элемент, нужно указать два индекса: первый (по соглашению) показывает номер строки, а второй (по соглашению) - номер столбца. Таблицы или массивы, которые требуют двух индексов для указания отдельного элемента, называются двумерными.

Каждый элемент в двумерном массиве `arr` определяется именем элемента в форме `arr[i][j]`; `arr` – это имя массива, а `i` и `j` – индексы, которые однозначно определяют каждый элемент в `arr`. Имена элементов первой строки имеют первый индекс 0, имена элементов четвертого столбца имеют второй индекс 3.

Многомерные массивы могут получать начальные значения в своих объявления точно так же, как массивы с единственным индексом. Значения группируются в строки, заключенные в фигурные скобки:

```
int b [2][3] = {{1,2,3,}, {4,5,6}};
```

Если начальных значений в данной строке не хватает для их присвоения всем элементам строки, то остающимся элементам строки присваиваются нулевые значения

```
int b [2][3] = {{1,}, {4,5,6}};
```

```
int Array [2][3] = {{1,2,3,4,5};
```

Объявление массива `Array` содержит пять начальных значений. Начальные значения присваиваются первой строке, затем второй строке. Любые элементы, которые не имеют явно заданных начальных значений, автоматически получают нулевые значения.

5. УКАЗАТЕЛИ

Объявления и инициализация переменных указателей

Указатели – это переменные, которые содержат в качестве своих значений адреса памяти. С другой стороны, указатель хранит адрес переменной, которая содержит определенное значение. В этом смысле имя переменной отсылает к значению прямо, а указатель – косвенно. Ссылка на значение посредством указателя называется **косвенной адресацией.**

Указатели, подобно любым другим переменным, перед своим использованием должны быть объявлены. Следующая строка

```
int *countPtr, count;
```

объявляет переменную `countPtr` типа `int*` (т.е. указатель на целое число) и читается как «`countPtr` является указателем на целое число» или «`countPtr` указывает на объект типа `int`». Однако переменная `count` объявлена как целое число, а не как указатель на целое число. Символ `*` в объявлении относится только к `countPtr`. Каждая переменная, объявляемая как указатель, должна иметь перед собой знак звездочки (`*`). Например, объявление

```
float *xPtr, *yPtr;
```

указывает, что и `xPtr` и `yPtr` являются указателями на значения типа `float`. Использование `*` подобным образом в объявлении показывает, что переменная объявляется как указатель. Указатели можно объявлять, чтобы указывать на объекты любого типа данных.

Указатели должны инициализироваться либо при своем объявлении, либо с помощью операции присваивания. Указатель может получить в качестве начального значения 0 или адрес. Указатель с начальным значением 0 ни на что не указывает.

5.1. Операции над указателями

1. ***Операция & или операция адресации*** – унарная операция, которая возвращает адрес своего операнда. Например, если имеются объявления

```
int y = 5;
int *yPtr;
```

то оператор

```
yPtr = &y;
```

присваивает адрес переменной `y` указателю `yPtr`. Говорят, что переменная `yPtr` указывает на `y`. Операнд операции адресации должен быть `L`-величиной

(т.е. чем-то таким, чему можно присвоить значение так же, как переменной); операция адресации не может быть применена к константам, к выражениям, не дающим результат, на который можно сослаться.

2. ***Операция **** , называемая **операцией косвенной адресации** или **операцией разыменования**, возвращает значение объекта, на который указывает ее операнд (т.е. указатель). Например, оператор

```
cout << *yPtr << endl;
```

выведет на экран значение, на которое ссылается указатель (число 5).

Программа на рис. 5.1. демонстрирует операции с указателями.

```
#include <iostream.h>

int main()
{
     int a;
     int *aPtr;
     a= 7;
     aPtr = &a;

     cout << "Address a is: " << &a << endl;      // 0x0012FF7C
     cout << "Value aPtr is: " << aPtr << endl;   // 0x0012FF7C
     cout << "Value a is: " << a << endl;         // 7
     cout << "Value *aPtr is:" << *aPtr << endl; // 7
     cout << "Proof of adding & and *" << endl;
     cout << "&*aPtr is " << &*aPtr <<  endl;     // 0x0012FF7C
     cout << "*&aPtr ia " << *&aPtr <<  endl;     // 0x0012FF7C
     return 0;
}
```

Рис. 5.1. Операции с указателями

Ячейки памяти выводятся как шестнадцатеричные целые с помощью `cout`. Адрес `a` и значение `aPtr` идентичны – это подтверждает, что адрес `a` действительно присвоен переменной указателю `aPtr`. Операции `&` и `*` взаимно дополняют друг друга – при их поочередном применении к `aPtr` в любой последовательности будет напечатан один и тот же результат.

Выражения и арифметические действия с указателями

С указателями может выполняться ограниченное количество арифметических операций. Указатель можно увеличивать (`++`), уменьшать (`--`), складывать с указателем целые числа (`+` или `+=`), вычитать из него целые числа (`-` или `-=`) или вычитать один указатель из другого.

Допустим, что объявлен массив `int v[10]` и его первый элемент находится в памяти в ячейке 3000. Допустим, что указателю `vPtr` было присвоено начальное значение путем указания на `v[0]`, т.е. начальное значение `vPtr` равно 3000 (см. рис. 5.2).

Указателю `vPtr` можно было дать начальное значение указанием на массив `v` с помощью одного из следующих операторов:

```
vPtr = v;
vPtr = &v[0];
```

В общепринятой арифметике сложение 3000 + 2 дает значение 3002. Это нормально, но не в случае арифметических действий с указателями. Когда целое складывается или вычитается из указателя, указатель не просто увеличивается или уменьшается на это целое, но это целое предварительно умножается на размер объекта, на который ссылается указатель. Количество байтов зависит от типа данных. Например, оператор

```
vPtr +=2;
```

выработал бы значение 3008 (3000 + 2*4) в предположении, что целое хранится в 4 байтах памяти. В массиве `v` указатель `vPtr` теперь указал бы на `v[2]`. Если целое хранится в 2 байтах памяти, то тот же самый оператор дал бы результат в памяти ячейку 3004 (3000+2*2). И т.д.

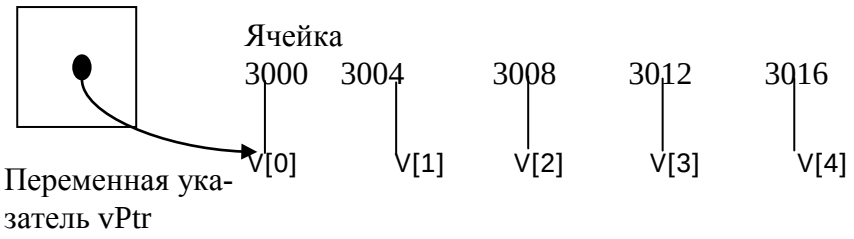


Рис. 5.2. Указатель на массив целых чисел

Если указатель `vPtr` был увеличен до значения 3016, указывающего на v[4], оператор

```
vPtr -= 4;
```

вернул бы `vPtr` обратно к значению 3000 – к началу массива. Если указатель увеличивается или уменьшается на 1, можно использовать операции инкремента (`++`) или декремента (`--`).

Переменные указатели можно вычитать друг из друга. Например, если `vPtr` содержит ячейку 3000, `v2Ptr` содержит адрес 3008, оператор

```
x = v2Ptr – vPtr;
```

присвоит x значение разности номеров элементов массива, на которые указывают vPtr и v2Ptr, в данном случае 2.

Арифметика указателей теряет смысл, если она выполняется не над массивами. Мы не можем предполагать, что две переменные одинакового типа хранятся в памяти вплотную друг к другу, если они не соседствуют в массиве.

5.2. Использование спецификатора const с указателями

Спецификация const дает возможность программисту информировать компилятор о том, что значение данной переменной не должно изменяться.

В случае попытки изменить значение константного параметра, компилятор отлавливает ее и выдает сообщение об ошибке.

Существуют четыре способа передачи в функцию указателя:

1) неконстантный указатель на неконстантные данные;
2) неконстантный указатель на константные данные;
3) константный указатель на неконстантные данные;
4) константный указатель на константные данные.

Каждая комбинация обеспечивает доступ с разным уровнем привилегий.

Наивысший уровень доступа предоставляется неконстантным указателям на неконстантные данные – данные можно модифицировать посредством разыменования указателя, а сам указатель может быть модифицирован, чтобы он указывал на другие данные. Объявление неконстантного указателя на неконстантные данные не содержит const.

Неконстантный указатель на константные данные – это указатель, который можно модифицировать, чтобы указывать на любые элементы данных подходящего типа, но данные, на которые он ссылается, не могут быть модифицированы. Такой указатель можно было бы использовать, чтобы передать в качестве аргументов массивы функции, которая будет обрабатывать каждый элемент массива без модификации данных (рис. 5.3).

```
#include <iostream.h>
void CubeByReference (const int*);
int main()
{    int number2 = 5;
     cout << "number2 before " << number2 << endl;
     CubeByReference (&number2);
     cout << "number2 after " << number2 << endl;
     return 0;
}

void CubeByReference (const int* nPtr)
{
     int number = 20;
     nPtr = &number;
     cout << *nPtr**nPtr**nPtr << endl;
```

```
    //*nPtr= *nPtr**nPtr**nPtr; нельзя изменять данные с помощью
    //операции разыменования указателя
    number = 1;
    cout << *nPtr**nPtr**nPtr << endl;
}
```

Рис. 5.3. Использование неконстантного указателя на константные данные

Константный указатель на неконстантные данные – это указатель, который всегда указывает на одну и ту же ячейку памяти, данные в которой можно модифицировать посредством указателя. Этот вариант реализуется по умолчанию для имени массива. **Имя массива – это константный указатель на начало массива**. Используя имя массива и индексы массива можно обращаться ко всем данным в массиве и изменять их. Указатели, объявленные как `const`, должны получить начальные значения при своем объявлении (если указатель является параметром функции, он получает начальное значение, равное указателю, который передается в функцию) (рис. 5.4.)

```
void CubeByReference (int* const);
int main()
{
    int number2;
    number2 = 5;
    cout << "number2 before " << number2 << endl;  // 5
    CubeByReference (&number2);
    cout << "number2 after " << number2 << endl;   // 125
    number2 = 10;
    cout << "number2 before " << number2 << endl;  // 10
    CubeByReference (&number2);   // 1000 потом 1000000000
    cout << "number2 after " << number2 << endl;   // 1000
    return 0;
}

void CubeByReference (int* const nPtr)
{
    int number = 20;
    //nPtr = &number; нельзя изменять значение указателя
    cout << *nPtr**nPtr**nPtr << endl;              //125
    *nPtr= *nPtr**nPtr**nPtr;
    number = 1;
    cout << *nPtr**nPtr**nPtr << endl;            //1953125
}
```

Рис. 5.4. Использование константного указателя на неконстантные данные

Наименьший уровень привилегий доступа предоставляет константный указатель на константные данные. Такой указатель всегда указывает на одну и ту же ячейку памяти и данные в этой ячейке нельзя модифицировать. Это выглядит так, как если бы массив нужно было передать функции, которая только просматривает массив, использует его индексы, но не модифицирует сам массив (рис. 5.5.).

```
int number1, number2;
number1 = 10;
number2 = 5;
const int * const ptr = &number2;
cout << *ptr << endl;;
//*ptr = number1; нельзя изменять данные через операцию разыменования
//ptr = &number1; нельзя изменять значение самого указателя
number1 = 10;
cout << *ptr << endl
```

Рис. 5.5. Использование неконстантного указателя на неконстантные данные

Взаимосвязи между указателями и массивами

Массивы и указатели в C++ тесно связаны и могут быть использованы почти эквивалентно. Имя массива можно понимать как константный указатель. Указатели можно использовать для выполнения любой операции, включая индексирование массива.

Предположим, объявлены массив целых чисел `b[5]` и целая переменная-указатель `bPtr`. Поскольку имя массива (без индекса) является указателем на первый элемент массива, то можно задать указателю `bPtr` адрес первого элемента массива `b` с помощью оператора

```
bPtr = b;
```

это эквивалентно присваиванию адреса первого элемента массива следующим образом:

```
bPtr = &b[0];
```

Сослаться на элемент массива `b[3]` можно с помощью выражения указателя

```
*(bPtr + 3)
```

В приведенном выражении цифра 3 является **смещением** указателя. Когда указатель указывает на начало массива, смещение показывает, на какой элемент массива должна быть ссылка, так что значение смещения эквивалентно индексу массива. Предыдущую запись называют записью **указатель-смещение**. Скобки необходимы, потому что приоритет `*` выше, чем приоритет. `+`. Без скобок это выражение прибавило бы число 3 к значению выражения `*bPtr` (т.е. 3 было бы прибавлено к `b[0]` в предположении, что `bPtr` указывает на начало массива).

Сам массив можно рассматривать как указатель и использовать в арифметике указателей.

Например, выражение

```
*(b + 3)
```

тоже ссылается на элемент массива `b[3]`. Вообще все выражения с индексами массива могли бы быть записаны с помощью указателей и смещений. В этом случае запись указатель-смещение применялась бы к имени массива как к указателю.

Указатели можно индексировать точно так же, как и массивы. Например, выражение

```
bPtr[1]
```

ссылается на элемент массива `b[1]`, это выражение рассматривается как запись указатель индекс.

Имя массива является **константным** указателем, оно всегда указывает на начало массива. Поэтому выражение

```
b += 3;
```

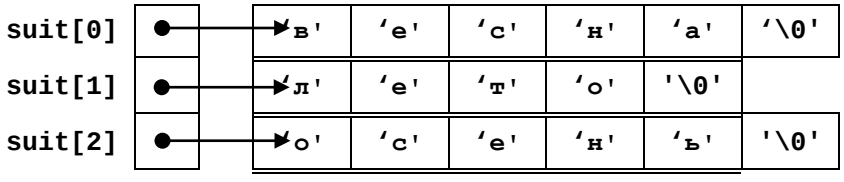
не разрешено, потому что оно пытается модифицировать значение имени массива с помощью арифметической операции над указателем.

5.3. Массивы указателей

Массивы могут содержать указатели. Типичным использованием такой структуры данных является формирование массива строк. Каждый элемент такого массива – строка, но в C++ строка является, по существу, указателем на ее первый символ. Таким образом, каждый элемент в массиве строк в действительности является указателем на первый символ строки (см. рис. 5.6.). Рассмотрим объявление массива строк `suit`

```
char *suit[4] = {“весна”, “лето”, “осень”, “зима”};
```

Элемент объявления `suit[4]` указывает массив из четырех элементов. Элемент объявления `char*` указывает, что тип каждого элемента массива `suit` – «указатель на `char`». Четыре значения, размещаемые в массиве – это «весна», «лето», «осень», «зима». Каждое из них хранится в памяти как строка, завершающаяся нулевым символом, которая на один символ длиннее, чем число символов текста, указанного в кавычках. Эти четыре строки имеют длину 6, 5, 6 и 5 символов соответственно. Хотя это и выглядит так, словно эти строки помещены в массив `suit`, на самом деле в массиве хранятся лишь указатели. Таким образом, хотя размер массива `suit` фиксирован, он обеспечивает доступ к строкам символов любой длины.



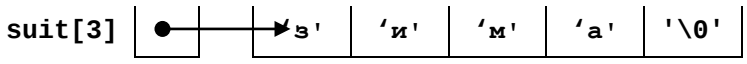


Рис. 5.6. Выделение памяти под массивы указателей

Строки символов могут быть размещены в двумерном массиве, в котором каждая строка представляет одно время года, а каждый столбец представляет одну букву имени времени года. Такая структура должна иметь фиксированное количество столбцов на строку и это количество должно быть таким большим, как самая длинная строка. Поэтому затраты памяти на хранение большого количества строк, большинство из которых короче, чем самая длинная строка, будут значительными.

5.4. Динамическое выделение памяти под массивы

Иногда заранее неизвестны размеры массивов. Тогда необходимо применять **динамические массивы**. При использовании массивов в предыдущих программах на этапе компиляции определялось, сколько места в памяти займет этот массив. И память под массив выделялась до выполнения самой программы, как под любые другие переменные.

```
int i,k;
char c;
char * MyArr;
cout << "Enter a number" << endl;  cin >> i;
//выделение памяти под массив символьного типа
MyArr = new char[i];
for (k=0;k<i;k++){
     MyArr[k] = k+60;
     c = MyArr[k];
     printf("%c\n", c);  }
delete [] MyArr;
```

Рис. 5.7. Выделение памяти под одномерный динамический массив

В случае если заранее неизвестны размеры массивов, тогда необходимо выделять память под массивы уже в процессе выполнения программы, получив от пользователя или рассчитав размер массива (рис. 5.7, 5.8 и 5.9).

```
int i,j, k, c;
int ** MyArr;
cout << "Enter two numbers" << endl;
cin >> i >> j;
//выделение памяти под двумерный массив
//сначала под массив указателей
MyArr = new int*[i];
//потом под каждый из подмассивов
for (k=0;k<i;k++)
```

```
{
     MyArr[k] = new int[j];
}
//****************************************
```

Рис. 5.8. Выделение памяти под двумерный динамический массив

```
//вывод на экран элементов массива
for (c=0;c<i;c++)
{
     for (k=0; k<j;k++)
     {
          MyArr[c][k] = c+k;
          cout << MyArr[c][k] << '\t';
     }
     cout << endl;
}
//************************************
//освобождение памяти
//сначала у каждого подмассива
for (k=0; k<i;k++)
     delete []MyArr[k];
//потом у массива указателей
delete [] MyArr;
//**********************************
```

Рис. 5.9. Работа с двумерным массивом и освобождение памяти из-под двумерного динамического массива

6. ФУНКЦИИ

6.1. Программные модули в C++

Большинство компьютерных программ, решающих реальные практические задачи, намного превышают те программы, которые были представлены ранее. Экспериментально доказано, что наилучшим способом создания и поддержки больших программ является их конструирование из маленьких фрагментов или **модулей**, каждый из которых более управляем, чем сложная программа. Эта техника называется «разделяй и властвуй».

Модули в C++ называются **функциями** и **классами**.

Функция – это именованный блок программы, созданный для решения одной небольшой задачи. ***Класс*** - это абстрактный тип данных. Обычно программы на C++ пишутся путем объединения новых функций, которые пишет сам программист, с функциями, уже имеющимися в стандартной библиотеке C, и путем объединения новых классов, которые пишет сам программист, с классами, уже имеющимися в различных библиотеках классов.

Стандартная библиотека C обеспечивает широкий набор функций для выполнения типовых математических расчетов, операций со строками, с символами, ввода-вывода, проверки ошибок и многих других полезных операций. Если возможно, нужно использовать стандартную библиотеку ANSI C вместо того, чтобы писать новые функции..

Программист может написать функции, чтобы определить, какие-то специфические задачи, которые можно использовать в разных местах программы. Эти функции называют **функциями, определенными пользователем**. Операторы, реализующие данную функцию, пишутся только один раз и скрыты от других функций.

Функция **активизируется** (т.е. начинает выполнять запроектированную для нее задачу) путем **вызова функции.** В вызове функции указывается ее имя и дается информация (в виде **аргументов**), необходимая вызываемой функции для ее работы.

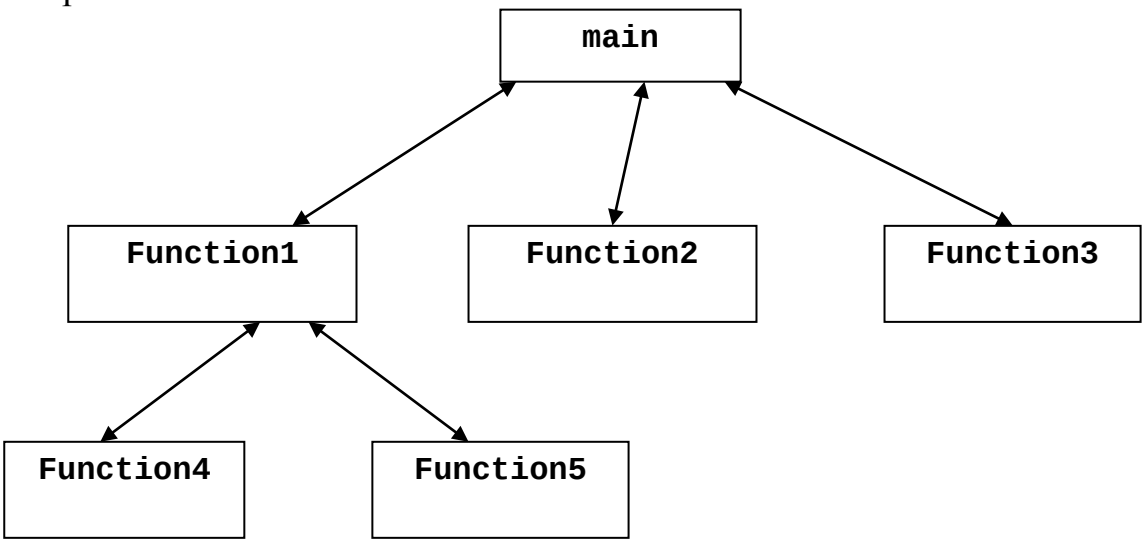


Рис. 6.1. Вызовы функций

Начальник (вызывающая функция или вызывающий оператор) просит подчиненного (вызываемую функцию) выполнить задание и возвратить (т.е. сообщить) результаты после того, как задание выполнено (рис. 6.1.). Функция-начальник не знает, как функция-подчиненный выполняет порученное ей задание.

6.2. Определения функций

Рассмотрим программу, которая использует функцию `square` для вычисления квадратов целых чисел от 1 до 10 (см. рис. 6.2.).

Функция создает копию значения `x` в параметре `y`. Затем `square` вычисляет `y*y`. Результат передает в ту точку `main`, из которой была вызвана `square`, и затем этот результат выводится на экран.

Описание square показывает, что эта функция ожидает передачи в нее целого параметра y. Ключевое слово int, предшествующее имени функции указывает, что square возвращает целый результат. Оператор return в square передает результат вычислений обратно в вызывающую функцию.

Функция не может быть определена внутри другой функции.

```
#include <iostream.h>
int square (int);

int main()
{
    for (int x = 1; x<=10; x++)
         cout << square (x) << " ";
    cout << endl;
    return 0;
}

int square (int y)
{
    return y*y;
}
```

Рис. 6.2. Программа, вычисляющая квадраты целых чисел

Строка

```
int square (int);
```

называется **прототипом** функции. Прототип функции указывает компилятору тип данных, возвращаемых функцией, количество параметров, которое ожидает функция, тип параметров и ожидаемый порядок их следования. Компилятор использует прототип функции для проверки правильности вызовов функции. Вызов функции, который не соответствует прототипу функции, ведет к синтаксической ошибке.

Другой важной особенностью прототипов функций является **приведение типов аргументов**, т.е. задание аргументам подходящего типа. Например, оператор

```
cout << sqrt (4);
```

правильно вычисляет и печатает значение квадратного корня 4. Функция sqrt библиотеки математических функций принимает значение типа double. Прототип функции заставляет компилятор преобразовать целое значение 4 в значение 4,0 типа double, прежде чем значение будет передано в sqrt.

Преобразования типов могут привести к неверным результатам, если не руководствоваться **правилами приведения типов** C++. Правила приведения определяют, как типы могут быть преобразованы в другие типы без потерь.

Правила приведения типов применяются к выражениям, содержащим значения двух или более типов данных; такие выражения относятся к **выраже-**

ниям смешанного типа. Тип каждого значения в выражениях смешанного типа приводится к «наивысшему» типу, имеющемуся в выражении (на самом деле создается и используется временная копия выражения – истинные значения остаются неизменными).

Задача. Определить максимальное из трех чисел (рис. 6.3.).

```
#include <iostream.h>
int maximum (int, int, int);
int main()
{
    int a,b,c;
    cout << "Enter 3 integers: ";
    cin >> a >> b >>c;
    cout << "Maximum is: " << maximum(a,b,c) << endl;
    return 0;
}

int maximum (int x, int y, int z)
{
    int max = x;
    if (y > max)   max = y;
    if (z>max)     max = z;
    return max;
}
```

Рис. 6.3. Программа, вычисляющая максимальное из трех чисел

Генерация случайных чисел

Программа моделирует бросание игральной кости (рис. 6.4.)

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

int main()
{
    srand(time(0));
    for (int i = 1; i <=20; i++)
         cout << 1+ rand() % 6 << endl;
    return 0;
}
```

Рис. 6.4. Программа, моделирующая бросание игральной кости

Функция `rand()` генерирует псевдослучайное целое число в диапазоне между 0 и `RAND_MAX` (константа, определенная в заголовочном файле

<stdlib.h>). Для данной программы необходимы только числа, лежащие в диапазоне от 1 до 6.

`rand() % 6` – эту операцию называют **масштабированием**. Число 6 называется **масштабирующим коэффициентом**. В результате этой операции получаются числа от 0 до 5 включительно. После масштабирования нужно **сдвинуть** диапазон чисел, добавляя 1 к каждому полученному результату.

Так как функция `rand()` генерирует псевдослучайные числа, то при втором запуске программы будут получены те же числа, что и при первом. Для получения разных последовательностей необходимо применить операцию **рандомизации**. Она реализуется с помощью библиотечной функции `srand`. Она получает в качестве аргумента беззнаковое целое и при каждом выполнении программы задает начальное число, которое функция `rand` использует для генерации последовательности квазислучайных чисел. Функция `time` стандартной библиотеки <time.h> возвращает наилучшее приближение текущего календарного времени, обеспечиваемое реализацией. Таким образом, при каждом запуске программы в генератор псевдослучайных чисел будет передаваться разное начальное число.

6.3. Классы памяти и область действия

Классы памяти

В C++ имеется четыре **спецификации класса памяти**: `auto`, `register`, `extern` и `static`. Спецификация класса памяти идентификатора помогает определить его класс памяти, область действия и пространство имен.

Класс памяти идентификатора определяет его **время жизни** - период, в течение которого этот идентификатор существует в памяти. Одни идентификаторы существуют недолго, другие - неоднократно создаются и уничтожаются, третьи - существуют на протяжении всего времени выполнения программы.

Спецификации класса памяти могут быть разбиты на два класса: **автоматический класс памяти с локальным временем жизни** и **статический класс памяти с глобальным временем жизни**. Ключевые слова `auto` и `register` используются для объявления переменных с локальным временем жизни. Такие переменные создаются при входе в блок, в котором они объявлены, они существуют лишь во время активности блока и исчезают при выходе из блока.

К классу с локальным временем жизни могут относиться только переменные. К этому классу относятся локальные переменные функций и параметры функций. Спецификация `auto` явно объявляет переменные автоматического класса памяти, т.е. с локальным временем жизни. Например, следующее объявление указывает, что переменные `x` и `y` типа `float` являются временными с локальным временем жизни, т.е. они существуют только в теле функции, в которой появляется объявление:

```
auto float x, y;
```

Локальные переменные являются переменными с локальным временем жизни по умолчанию, так что ключевое слово `auto` используется редко. Далее будем ссылаться на переменные автоматического класса памяти просто как на автоматические переменные.

Обычно данные в версии программы на машинном языке загружаются для расчетов и другой обработки в регистры.

Спецификация класса памяти `register` может быть помещена перед объявлением автоматической переменной, чтобы компилятор сохранял переменную не в памяти, а в одном из высокоскоростных аппаратных регистров компьютера. Если интенсивно используемые переменные, такие как счетчики или суммы могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память могут быть исключены.

Компилятор может проигнорировать объявления `register`. Например, может оказаться недостаточным количество регистров, доступных компилятору для использования. Приведенное далее объявление определяет, что целая переменная `counter` должна быть помещена в один из регистров компьютера; независимо от того, сделает это компилятор или нет, `counter` получит начальное значение 1:

```
register int counter = 1;
```

Ключевое слово `register` может применяться только к локальным переменным и параметрам функций. Часто объявления `register` не являются необходимыми. Современные оптимизирующие компиляторы способны распознавать часто используемые переменные и решать, помещать их в регистры или нет, не требуя от программиста объявления `register`

Ключевые слова `extern` и `static` используются, чтобы объявить идентификаторы переменных и функций как идентификаторы статического класса памяти с глобальным временем жизни. Такие переменные существуют с момента начала выполнения программы. Для таких переменных память выделяется и инициализируется сразу после начала выполнения программы. Имена функций тоже существуют с самого начала выполнения программы. Однако это не означает, что эти идентификаторы могут быть использованы по всей программе. Класс памяти и область действия (где можно использовать имя) — это разные вещи.

Существуют два типа идентификаторов статического класса памяти: внешние идентификаторы (такие, как глобальные переменные и имена функций), и локальные переменные, объявленные спецификацией класса памяти `static`. Глобальные переменные и имена функций по умолчанию относятся к классу памяти `extern`. Глобальные переменные создаются путем размещения их объявлений вне описания какой-либо функции. Глобальные переменные сохраняют свои значения в течение всего времени выполнения программы. На глобальные переменные и функции может ссылаться любая функция, которая расположена после их объявления или описания в файле.

Локальные переменные, объявленные с ключевым словом `static`, известны только в той функции, в которой они определены, но в отличие от автоматических переменных локальные переменные `static` сохраняют свои значения в течение всего времени существования функции. При каждом следующем вызове функции локальные переменные содержат те значения, которые они имели при предыдущем вызове. Следующий оператор объявляет локальную переменную `count` как `static` и присваивает начальное значение 1

```
static int count = 1;
```

Все числовые переменные статического класса памяти принимают нулевые начальные значения, если программист явно не указал другие начальные значения.

Спецификации класса памяти `extern` и `static` имеют специальное значение, когда они применяются явно к внешним идентификаторам.

Область действия

Область действия идентификатора – это часть программы, в которой на идентификатор можно ссылаться. Например, когда объявляется локальная переменная в блоке, на нее можно ссылаться только в этом блоке или в блоке, вложенном в этот блок. Существуют четыре области действия идентификатора – **область действия функция, область действия файл, область действия блок и область действия прототип функции.**

Идентификатор, объявленный **вне любой функции** (на внешнем уровне), имеет **область действия файл**. Такой идентификатор известен всем функциям от точки его объявления до конца файла.

Метки (идентификаторы с последующим двоеточием, например, `start:`) – единственные идентификаторы, имеющие **областью действия функцию**. Метки можно использовать всюду в функции, в которой они появились, но на них нельзя ссылаться вне тела функции.

Идентификаторы, объявленные **внутри блока** (на внутреннем уровне), имеют **областью действия блок**. Область действия блок начинается объявлением идентификатора и заканчивается конечной правой скобкой. Локальные переменные, объявленные в начале функции, имеют областью действия блок так же, как и параметры функции, являющиеся локальными переменными. Если блоки вложены и идентификатор во внешнем блоке имеет такое же имя, как идентификатор во внутреннем блоке, идентификатор внешнего блока «невидим» (скрыт) до момента завершения работы внутреннего блока. Это означает, что пока выполняется внутренний блок, он видит значение своих собственных идентификаторов, а не значения идентификаторов с идентичными именами в охватывающем блоке.

Единственными идентификаторами с **областью действия прототип функции** являются те, которые используются в **списке параметров прототипа функции**. Если в списке параметров прототипа функции используется имя, компилятор его просто игнорирует. Идентификатор, используемый в прототипе

функции, можно повторно использовать где угодно в программе, не опасаясь двусмысленности.

6.4. Рекурсия

Рекурсивная функция – это функция, которая вызывает сама себя либо непосредственно, либо косвенно с помощью другой функции.

Рекурсивная задача в общем случае разбивается на ряд этапов. Для решения задачи вызывается рекурсивная функция. Эта функция знает, как решать только простейшую часть задачи – так называемую базовую задачу. Если эта функция вызывается для решения базовой задачи, она просто возвращает результат. Если функция вызывается для решения более сложной задачи, она делит эту задачу на две части: одну часть, которую функция решать умеет, и другую, которую функция решать не умеет. Чтобы сделать рекурсию выполнимой, последняя часть должна быть похожа на исходную задачу, но быть по сравнению с ней несколько проще или несколько меньше. Поскольку эта новая задача подобна исходной, функция вызывает новую копию самой себя, чтобы начать работать над меньшей проблемой – это называется **рекурсивным вызовом** или **шагом рекурсии**. Шаг рекурсии включает ключевое слово `return`, так как в дальнейшем его результат будет объединен с той частью задачи, которую функция решать умеет, и сформируется конечный результат, который будет передан обратно в исходное место вызова.

Шаг рекурсии выполняется до тех пор, пока исходное обращение к функции не закрыто, т.е. пока еще не закончено выполнение функции. Шаг рекурсии может приводить к большому числу таких рекурсивных вызовов, поскольку функция продолжает деление каждой новой подзадачи на две части. Чтобы завершить процесс рекурсии, каждый раз, как функция вызывает саму себя с несколько упрощенной версией исходной задачи, должна формироваться последовательность все меньших и меньших задач, в конце концов, сходящаяся к базовой задаче. В этот момент функция распознает базовую задачу, возвращает результат предыдущей копии функции и последовательность возвратов повторяет весь путь назад, пока не дойдет до первоначального вызова и не возвратит конечный результат в `main` (см. рис. 6.6)

Факториал *n!*:

`n! = n*(n-1)*(n-2)*…*1`

Факториал может быть вычислен не рекурсивно, а итеративно с помощью оператора `for`:

```
factorial = 1;
for(int counter = number; counter >=1; counter--)
     factorial *= counter;
```

Рис. 6.5. Итеративное вычисление факториала

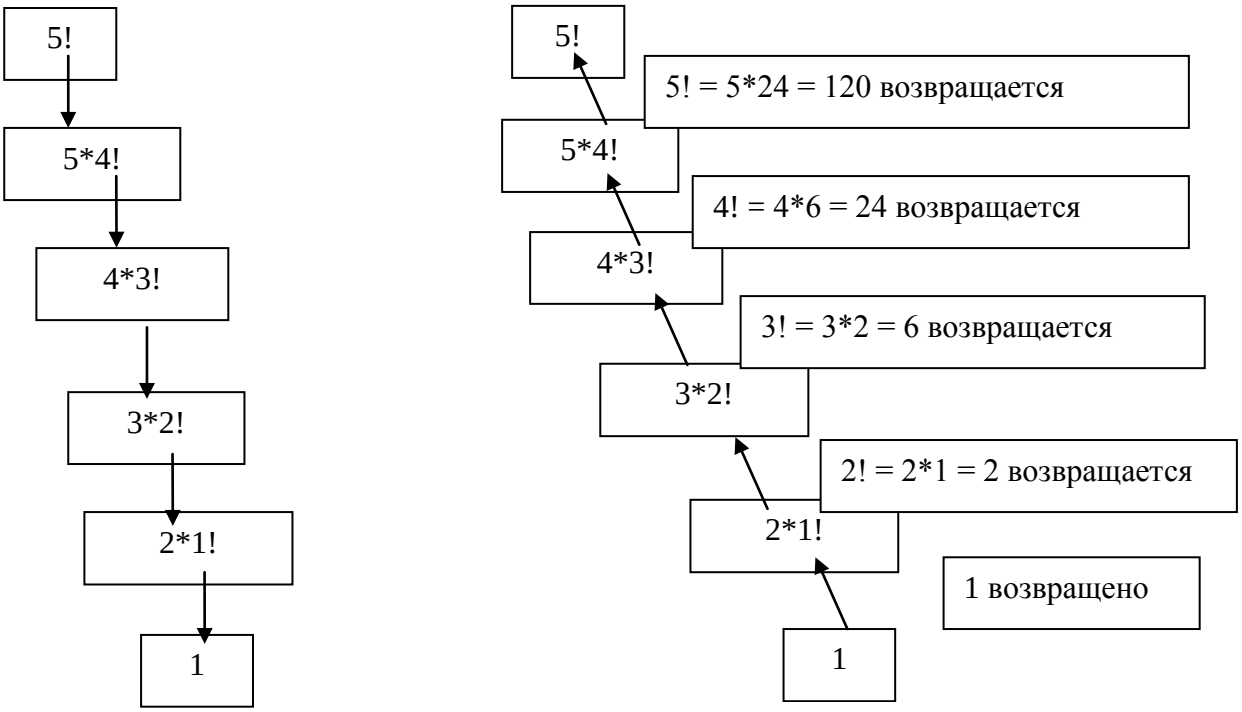


Рис. 6.6. Рекурсивное дерево для вычисления факториала

Рекурсивное определение функции факториал дается следующим соотношением: `n! = n*(n-1)!`

```
unsigned long factorial(unsigned long number)
{
if (number <=1)
    return 1;
else
    return number * factorial (number-1);
}
```

Рис. 6.7. Рекурсивное вычисление факториала

Последовательность чисел Фибоначчи

Это пример как не надо использовать рекурсию.

Последовательность чисел Фибоначчи рекурсивно можно определить следующим образом:

```
fibonacci(0) = 0
fibonacci(1) = 0
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

Рис. 6.8. Формула для числа Фибоначчи

Функция, рекурсивно вычисляющая число Фибоначчи:

```
unsigned long fibonacci (insigned long n)
{
     if(n == 0 || n == 1)
          return n;
     else
     return fibonacci (n-1) + fibonacci (n-2);
}
```

Рис.6.9. Рекурсивное вычисление числа Фибоначчи

При n>1 шаг рекурсии генерирует два рекурсивных вызова, каждый из которых представляет собой несколько упрощенную задачу по сравнению с исходным вызовом `fibonacci.`

Количество рекурсивных вызовов, которое должно быть выполнено для вычисления n-го числа Фибоначчи, оказывается порядка 2^n. Поэтому поиск числа Фибоначчи обычно осуществляют итеративным путем

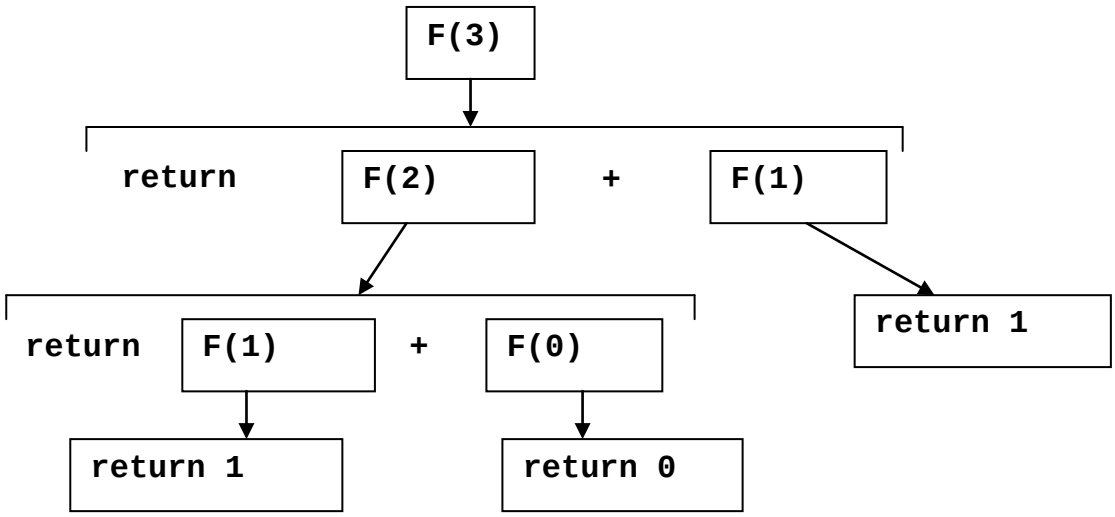


Рис. 6.10. Рекурсивное дерево для вычисления числа Фибоначчи

6.5. Ссылки и ссылочные параметры

Во многих языках программирования имеются два способа обращения к функциям – вызов по значению и вызов по ссылке.

В C++ существуют три способа передачи аргументов в функции:

1) вызов по значению;

2) вызов по ссылке с аргументами ссылками;

3) вызов по ссылке с аргументами указателями.

Когда аргумент передается вызовом по значению, создается копия аргумента, и она передается вызываемой функции. Изменения копии не влияет на значение оригинала в операторе вызова. Это предотвращает случайный побоч-

ный эффект, который так сильно мешает развитию надежного и корректного программного обеспечения. Все аргументы, которые передавались в функции до этого момента, передавались вызовом по значению. Один из недостатков вызова по значению состоит в том, что если передается большой элемент данных, это может привести к значительным потерям времени выполнения.

Ссылочный параметр – это скрытый указатель, псевдоним соответствующего аргумента. Чтобы показать, что параметр функции передан по ссылке, после типа параметра в прототипе функции ставится символ амперсанда (&); такое же обозначение используется в списке типов параметров в заголовке функции. Например, объявление `int &count` в заголовке функции может читаться как «`count` является ссылкой на `int`». В вызове функции достаточно указать имя переменной, и она будет передана по ссылке. Тогда упоминание в теле вызываемой функции переменной по имени ее параметра в действительности является обращением к исходной переменной в вызывающей функции, и эта исходная переменная может быть изменена непосредственно вызываемой функцией (рис. 6.11.)

```
#include <iostream.h>

int squareByValue(int);
void squareByReference(int &);

int main()
{
    int x = 2, z = 4;
    cout << "Before:" << endl;
    cout << "x= " << x << endl;
    cout << "z= " << z << endl;
    squareByValue(x);
    squareByReference(z);
    cout << "After:" << endl;
    cout << "x= " << x << endl;
    cout << "z= " << z << endl;
    return 0;
}

int squareByValue(int a)
{
    return a *=a;
}

void squareByReference(int &cRef)
{
    cRef *=cRef;
}
```

Рис. 6.11. Передача параметра по значению и по ссылке

Ссылки можно также использовать как псевдонимы для других переменных внутри функций

```
int count = 1;      //объявление целой переменной count
int &cRef = count; //создание cRef как псевдонима для count
++cRef;
```

Рис. 6.12. Создание ссылки внутри функции

Этот фрагмент дает приращение переменной `count`, используя ее псевдоним `cRef`. Ссылочные переменные должны получать начальные условия в их объявлениях. Как только ссылка объявляется как псевдоним другой переменной, все операции, выполняемые с псевдонимом (т.е. ссылкой), на самом деле будут выполняться с самой истинной переменной. Для псевдонима не резервируется никакого места в памяти.

Вызов функций по ссылке с аргументами указателями

Ранее было проведено сравнение и сопоставление вызовов по значению и по ссылке с аргументами ссылки. Сейчас рассмотрим на вызове по ссылке с аргументами указателями.

`return` можно использовать для возвращения одного значения из вызываемой функции вызывающему оператору (или для передачи управления из вызываемой функции без возвращения какого-либо значения). Из изложенного, что аргументы могут быть переданы функции с использованием аргументов ссылок, чтобы дать возможность функции модифицировать исходные значения аргументов (таким образом, из функции может быть «возвращено» более одного значения), или чтобы передать функции большие объекты и избежать накладных расходов, сопутствующих передаче объектов вызовом по значению. Указатели, подобно ссылкам, тоже можно использовать для модификации одного или более значений переменных в вызывающем операторе, или передавать указатели на большие объекты данных, чтобы избежать накладных расходов, сопутствующих передаче объектов по значению.

В C++ программисты могут использовать указатели и операции косвенной адресации для моделирования вызова по ссылке. При вызове функции с аргументами, которые должны быть модифицированы, передаются их адреса. Это обычно сопровождается операцией адресации(&) переменной, которая должна быть модифицирована. Массивы не передаются с помощью &, потому что имя массива – это начальный адрес массива в памяти. При передаче функции адреса переменной может использоваться операция косвенной адресации (*) для модификации значения (если значение не объявлено как `const`) ячейки в памяти вызывающего оператора.

На рис. 6.13 представлена программы с двумя вариантами функции, которая возводит в куб целое число – `CubeByValue` и `CubeByReference`. Первая функция передает переменную `number` функции `CubeByValue` вызо-

вом по значению. Функция CubeByValue возводит в куб свой аргумент и возвращает новое значение в main, используя оператор return. Новое значение присваивается переменной number в main. Ключевой момент вызова по значению состоит в том, что программисту предоставляется возможность анализировать результат вызова функции перед модификацией значения переменной. Например, в первой программе можно было бы сохранить результат, возвращаемый CubeByValue в другой переменной, исследовать его значение, и после этого присвоить результат переменной number.

Вторая функция передает переменную number по ссылке – в функцию CubeByReference передается адрес number. Функция CubeByReference в качестве аргумента получает nPtr (указатель на int). Функция разыменовывает указатель и возводит в куб значение, на которое указывает nPtr. Это изменяет значение number в main.

```
int CubeByValue (int);
void CubeByReference (int*);

int main()
{
    int number1;
    int number2;
    number1 = 5;
    number2 = 5;
    cout << "number1 before " << number1 << endl;
    number1 = CubeByValue (number1);
    cout << "number1 after " << number1 << endl;;
    cout << "number2 before " << number2 << endl;
    CubeByReference (&number2);
    cout << "number2 after " << number2 << endl;
    return 0;
}

int CubeByValue (int n)
{
    return n*n*n;
}

void CubeByReference (int* nPtr)
{
    *nPtr = *nPtr **nPtr**nPtr;
}
```

Рис. 6.13. Программа, иллюстрирующая передачу параметра по значению и по ссылке с аргументом указателем

Функция, принимающая адрес в качестве аргумента, должна определить параметр как указатель, чтобы принять адрес.

6.6. Перегрузка функций

C++ позволяет определить несколько функций с одним и тем же именем, если эти функции имеют разные наборы параметров (по меньшей мере, разные типы параметров). Эта особенность называет **перегрузкой функции.** При вызове перегруженной функции компилятор C++ определяет соответствующую функцию путем анализа количества, типов и порядка следования аргументов в вызове. Перегрузка функций обычно используется для создания нескольких функций с одинаковым именем, предназначенных для выполнения сходных задач, но с разными типами данных (рис. 6.14.)

```
#include <iostream.h>

int square (int x)
{
    return x*x;
}
double square (double y)
{
    return y*y;
}

int main()
{
    cout << "Integer square is " << square(7)<<endl;
    cout << "Double square is " << square(7.1)<<endl;
    return 0;
}
```

Рис. 6.14. Перегрузка функции вычисления квадрата числа

Аргументы по умолчанию

Обычно при вызове функции в нее передается конкретное значение каждого аргумента. Но программист может указать, что аргумент является **аргументом по умолчанию** и задать для этого аргумента значение по умолчанию. Если аргумент по умолчанию не указан в вызове функции, то в вызов автоматически передается значение этого аргумента по умолчанию. Аргументы по умолчанию должны быть самыми **правыми** (последними) в списке параметров функции (см. рис. 6.15.).

Если вызывается функция с двумя или более аргументами по умолчанию и если пропущенный аргумент не является самым правым в списке аргументов, то все аргументы справа от пропущенного также пропускаются. Аргументы по умолчанию должны быть указаны при первом упоминании имени функции – обычно в прототипе. Повторное указание значения аргумента по умолчанию не допускается. Аргументы по умолчанию могут быть константами, глобальными переменными или вызовами функций.

```
#include <iostream.h>

int test (double a, int b);
int test (int a, double b);
int test (int a, int b, int c=3);
int test (int a, int b = 4);
int test (int a=7);

int main()
{
     test(3.5, 6);       //1
     test(6,3.5);        //2
     test();             //5
     test(3,4,5);        //3
     return 0;
}

int test (double a, int b) {cout<<1<<endl; return 0;}
int test (int a, double b) {cout<<2<<endl; return 0;}
int test (int a, int b, int c) {cout<<3<<endl; return 0;}
int test (int a, int b) {cout<<4<<endl; return 0;}
int test (int a) {cout<<5<<endl; return 0;}
```

Рис. 6.15. Перегрузка функции `test` с аргументами по умолчанию

6.7. Передача массивов в функции

C++ автоматически передает массивы функциям, используя моделируемый вызов по ссылке – вызываемые функции могут изменять значения элементов в исходных массивах источника вызова. Значение имени массива является адресом первого элемента массива. Поскольку в функцию передается начальный адрес массива, вызываемая функция знает, где хранится массив. Поэтому, когда вызываемая функция модифицирует элементы массива в теле функции, она модифицирует реальные элементы массива в их истинных ячейках памяти.

Хотя массивы передаются моделируемым вызовом по ссылке, отдельные элементы массива передаются вызовом по значению подобно простым переменным. Такие отдельные простые элементы данных называются скалярами, или скалярными переменными. Чтобы передать в функцию элемент массива, необходимо использовать индексированное имя элемента массива как аргумент в вызове функции.

Задача. Известны данные наблюдений среднесуточной температуры первой недели января за 10 лет. Определить среднюю температуру первой недели января для каждого года (см. рис. 6.16).

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

const int N = 10;
const int M = 7;

void print(float av[]) //пустые квадратные скобки
{
     for(int j = 0; j<N; j++) //N видна во всем файле
          cout<<av[j]<<endl;
}

void set_temp(int t[][M]) //вторые и следующие скобки заполнены
{
     srand(time(0));
     rand();
     for(int i=0;i<N;i++)     {
          for(int j = 0; j<M; j++)
          {
               t[i][j] = rand()%41-30;
               cout<<t[i][j]<<"\t";
          }    cout<<endl;
     }
}

void calculate(int t[][M], float av[])
{
     int temp;
     for(int i=0;i<N;i++)
     {
          temp=0;
          for(int j = 0; j<M; j++)
               temp+= t[i][j];
          av[i]=float(temp)/M;
     }
}

int main()
{

     int temperature[N][M];
     float average[N];
     set_temp(temperature);
     calculate(temperature, average);
     print(average);
     return 0;
}
```

Рис. 6.16. Вычисление средненедельной температуры января

6.8. Указатель на функцию

Возможны только две операции с функциями:

1) вызов,
2) взятие адреса.

Указатель, полученный с помощью последней операции, можно впоследствии использовать для вызова функции (рис. 6.17.)

```
void error(char* p) { /* ... */ }

void (*efct)(char*); // указатель на функцию

void f() {
     efct = &error; // efct настроен на функцию error
     (*efct)("error");} // вызов error через указатель efct
```

Рис. 6.17. Использование указателя на функцию

Для вызова функции с помощью указателя (`efct` в нашем примере) надо вначале применить операцию разыменования к указателю - `*efct`. Поскольку приоритет операции вызова `()` выше, чем приоритет разыменования `*`, нельзя писать просто `*efct("error")`. Это будет означать `*(efct("error"))`, что является ошибкой. По той же причине скобки нужны и при описании указателя на функцию. Однако писать просто `efct("error")` можно, так как транслятор понимает, что `efct` является указателем на функцию, и создает команды, делающие вызов нужной функции.

Формальные параметры в указателях на функцию описываются так же, как и в обычных функциях. При присваивании указателю на функцию требуется точное соответствие типа функции и типа присваиваемого значения (рис. 6.18.)

```
void (*pf)(char*); // указатель на void(char*)
void f1(char*);
int f2(char*);
void f3(int*);

void f() {
     pf = &f1; // нормально
     pf = &f2; // ошибка: не тот тип возвращаемого значения
     pf = &f3; // ошибка: не тот тип параметра
     (*pf)("asdf"); // нормально
     (*pf)(1); // ошибка: не тот тип параметра
     int i = (*pf)("qwer"); // ошибка: void присваивается int
}
```

Рис. 6.18. Ошибки, возникающие при использовании указателя на функцию

6.9. Командная строка аргументов

Системные средства, на которые опирается реализация языка С, позволяют передавать командную строку аргументов или параметров начинающей выполняться программе. Когда функция `main` вызывается к исполнению, она вызывается с двумя аргументами. Первый аргумент (условно называемый `argc`) указывает число аргументов в командной строке, с которыми происходит обращение к программе; второй аргумент (`argv`) является указателем на массив символьных строк, содержащих эти аргументы, по одному в строке. Работа с такими строками - это обычное использование многоуровневых указателей.

Самую простую иллюстрацию этой возможности и необходимых при этом описаний дает программа `echo`, которая просто печатает в одну строку аргументы командной строки, разделяя их пробелами (рис. 6.19). Таким образом, если дана команда

```
echo Hello, world
```

то выходом будет

```
Hello, world
```

По соглашению `argv[0]` является именем, по которому вызывается программа, так что `argc` по меньшей мере равен 1. В приведенном выше примере `argc` равен 3, а `argv[0]`, `argv[1]` и `argv[2]` равны соответственно «`echo`», «`Hello,`» и «`world`». Первым фактическим аргументом является `argv[1]`, а последним - `argv[argc-1]`. Если `argc` равен 1, то за именем программы не следует никакой командной строки аргументов.

```
#include <iostream.h>

int main(int argc, char* argv[])
{
      int i;
//первый вариант*********************************
     for(i = 1; i < argc; i++)
          cout << argv[i] << (i<argc-1) ? ' ' : '\n';
//*****************************************
//второй вариант*********************************
      while (--argc > 0)
          cout<<*++argv << (argc > 1) ? ' ' : '\n';
//*****************************************
return 0;
}
```

Рис. 6.19. Работа команды `echo`

7. ВВЕДЕНИЕ В ОБРАБОТКУ СТРОК

Понятие символов и строк в С++

Символы – это фундаментальные стандартные блоки исходных программ на С++. Каждая программа составлена из последовательности символов, которые, если их объединение в группу имеет смысл, интерпретируются компилятором как последовательности инструкций, используемых для выполнения задачи. Программа может содержать символьные константы. **Символьная константа** – это целое значение, представленное как символ в одинарных кавычках. Значение символьной константы – это целочисленное значение в наборе машинных символов. Например, `'z'` представляет собой целое значение `z`, а `'\n'` представляет собой целое значение символа перехода на новую строку.

Строка - это последовательность символов, заключенная в двойные кавычки. Строка имеет тип "массив символов" и класс памяти `static`. Строка инициализирована указанными в ней символами. Компилятор помещает в конец каждой строки нулевой байт `\0`, с тем, чтобы просматривающая строку программа могла определить ее конец. Перед стоящим внутри строки символом двойной кавычки должен быть поставлен символ обратной косой черты `\`; кроме того, могут использоваться те же условия последовательности, что и в символьных константах.

Строка может включать буквы, цифры, и разнообразные символы, например, такие как +, -, *, / , $ и другие.

Строка доступна через указатель на первый символ в строке, значением строки является адрес ее первого символа. Таким образом можно сделать вывод, что в С++ **строка является указателем на первый символ строки.** Поэтому все строки, даже идентично записанные, считаются различными.

Функции для работы со строками

Функции для работы со строками находятся в стандартной библиотеке `string.h`

Работа с функцией printf (библиотека stdio.h)

Функция printf используется для вывода констант и переменных на экран. На рис. 7.1 представлена программа, работающая с этой функцией.

```
#include <stdio.h>
void main(){
   char   ch = 'h', *string = "computer";
   int    count = -9234; double fp = 251.7366;
   wchar_t wch = L'w', *wstring = L"Unicode";
   /* Display integers. */
   printf( "Integer formats:\n"
           "\tDecimal: %d  Justified: %.6d  Unsigned: %u\n",
           count, count, count, count );
   printf( "Decimal %d as:\n\tHex: %Xh C hex: 0x%x Octal: %o\n",
            count, count, count, count );
   /* Display in different radixes. */
   printf("Digits 10 equal:\n\tHex: %i Octal: %i Decimal: %i\n",
            0x10, 010, 10 );
   /* Display characters. */
   printf("Characters in field:\n%10c%5hc%5C%5lc\n", ch, ch, wch,
            wch);
   /* Display strings. */
   printf("Strings in field:\n%25s\n%25.4hs\n\t%S%25.3ls\n",
            string, string, wstring, wstring);
   /* Display real numbers. */
   printf( "Real numbers:\n\t%f %.2f %e %E\n", fp, fp, fp, fp );
   /* Display pointer. */
   printf( "\nAddress as:\t%p\n", &count);
   /* Count characters printed. */
   printf( "\nDisplay to here:\n" );
   printf( "1234567890123456%n78901234567890\n", &count );
   printf( "\tNumber displayed: %d\n\n", count );}
```

Рис. 7.1. Работа с функцией `printf`

```
Integer formats:
   Decimal: -9234  Justified: -009234  Unsigned: 4294958062
Decimal -9234 as:
   Hex: FFFFDBEEh  C hex: 0xffffdbee  Octal: 37777755756
Digits 10 equal:
   Hex: 16  Octal: 8  Decimal: 10
Characters in field:
         h    h    w    w
Strings in field:
                computer
                    comp
   Unicode                      Uni
Real numbers:
   251.736600  251.74  2.517366e+002  2.517366E+002
Address as:   0012FFAC
Display to here:
12345678901234567890123456789O
   Number displayed: 16
```

Рис. 7.2. Результат работы функции `printf`

Определение длины строки

Для определения длины строки используется функция `strlen`. Она возвращает длину строки без учета символа конца строки. (рис. 7.3.)

```
#include <string.h>
#include <stdio.h>
int main(void){
   char *string = "Borland International";
   printf("%d\n",strlen(string));
   return 0;}
```

Рис. 7.3. Функция определения длины строки

Сложение двух строк (конкатенация)

Сложение двух строк. Программист должен быть уверен, что массив, используемый для хранения первой строки, достаточно велик для того, чтобы хранить комбинацию первой строки, второй строки и завершающего нулевого символа (рис. 7.4.).

```
#include <string.h>
#include <stdio.h>
int main(void){
   char destination[25];
   char *blank = " ", *c = "C++", *turbo = "Turbo";
   strcat(destination,turbo);
   strcat(destination,blank);
   strcat(destination,c);
   printf("%s\n",destination); //Turbo C++
   return 0;}
```

Рис. 7.4. Функция сложения двух строки

Добавление к исходной строке указанного количества символов.

К результату добавляется завершающий нулевой символ (рис. 7.5.)

```
#include <string.h>
#include <stdio.h>
void main(){
   char string[80] = "This is the initial string!";
   char suffix[] = " extra text to add to the string...";
/* Объединение строки с не более чем 19 символами suffix: */
   printf( "Before: %s\n", string );
   strncat( string, suffix, 19 );
   printf( "After:  %s\n", string );}

Before: This is the initial string!
After:  This is the initial string! extra text to add
```

Рис. 7.5. Функция добавления указанного количества символов к строке

Копирование строки в другую строку

При копировании строки программист должен быть уверен, что массив, используемый для хранения первой строки, достаточно велик для того, чтобы хранить вторую строку и завершающий нулевой символа. (рис. 7.6.)

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char destination[25];
   char *blank = " ", *c = "C++", *turbo = "Turbo";
   strcpy(destination,turbo);
   strcpy(destination,blank);
   strcpy(destination,c);
   printf("%s\n",destination); //C++
   return 0;
}
```

Рис. 7.6. Функция копирования строки в другую

Сравнение строк

Что значит строка «больше» или «меньше» другой строки? Все буквы алфавита идут в определенном порядке. Это значит, что `'Z'` – это не просто буква английского алфавита, это 26-я буква английского алфавита. Все символы представляются внутри компьютера как численные коды, когда компьютер сравнивает две строки, он на самом деле сравнивает численные коды символов в строке (численные коды символов упорядочены только для латинских букв, это не касается кириллицы).

В попытке стандартизации представления символов большинство производителей компьютеров придерживаются одной из популярных кодирующих схем – `ASCII` «Американский стандартный код для информационного обмена» (`American Standard Code for Information Interchange`) или `EBCDIC` «Расширенный двоичный код закодированного десятичного обмена» (`Extended Binary Code Decimal Interchange Code`). Манипуляции со строками и символами на самом деле подразумевают манипуляцию с соответствующими численными кодами, а не с самими символами. Это объясняет взаимозаменяемость символов и целых в C++. Так как имеет смысл утверждать, что один численный код больше, меньше или равен другому численному коду, стало возможным сопоставлять различные строки и символы друг с другом путем ссылки на коды символов (см. рис. 7.7.).

Функция `strcmp` возвращает 1, если первая строка больше второй, -1, если первая строка меньше второй, и 0, если строки эквивалентны.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
   int ptr;
   ptr = strcmp(buf2,buf1);
   if(ptr>0)
      printf("buf2 больше чем buf1\n");
   else
      printf("buf2 меньше чем buf1\n");
   ptr = strcmp(buf2,buf3);
   if(ptr>0)
      printf("buf2 больше чем buf3\n");
   else
      printf("buf2 меньше чем buf3\n");
   return 0;
}
```

Рис. 7.7. Функция сравнения двух строк

Между собой можно сравнивать не строки целиком, а только части двух строк (рис. 7.8.). В этом случае будут сравниваться только первые n символов. При сравнении будут учитываться и регистры символов.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *buf1 = "aaabbb", *buf2 = "bbbccc", *buf3 = "ccc";
   int ptr;
   ptr = strncmp(buf2,buf1,3);
   if(ptr>0)
      printf("buf2 больше чем buf1\n");
   else
      printf("buf2 меньше чем buf1\n");
   ptr = strncmp(buf2,buf3,3);
   if(ptr>0)
      printf("buf2 больше чем buf3\n");
   else
      printf("buf2 меньше чем buf3\n");
   return 0;
}
```

Рис. 7.8. Функция сравнения части двух строк

8. РАБОТА С ФАЙЛАМИ

Все до сих пор написанные программы читали из стандартного ввода и писали в стандартный вывод, относительно которых предполагалось, что они магическим образом предоставлены программе местной операционной системой.

Теперь научимся писать программы, которые считывают информацию из текстового файла и записывают информацию в текстовый файл. Используем функции библиотеки `stdio.h`.

Открытие файла

Для начала работы с файлом его необходимо открыть. Делает это функция `fopen` библиотеки `stdio.h`

```
FILE * fopen(char * filename, char * type);
```

Функция `fopen` открывает файл, именованный параметром `filename` и связывает его с соответствующим потоком `stream`. Функция `fopen` в качестве результата возвращает адресный указатель, который будет идентифицировать поток `stream` в последующих операциях. Строка `type`, используемая в функции `fopen`, может принимать следующие значения:

`r` - открытие файла только для чтения;

`w` - создание файла для записи;

`a` - присоединение; открытие для записи в конец файла или создание для записи, если файл не существует;

`r+` - открытие существующего файла для обновления (чтения и записи);

`w+` - создание нового файла для изменения;

`a+` - открытие для присоединения; открытие (или создание, если файл не существует) для обновления в конец файла.

Если данный файл открывается или создается в текстовом режиме, то можно приписать символ `t` к значению параметра `type` (`rt`, `w+t`, и т.д.); аналогично, для спецификации бинарного режима можно к значению параметра `type` добавить символ `b` (`wb`, `a+b`, и т.д.). Если в параметре `type` отсутствуют символы `t` или `b`, режим будет определяться глобальной переменной `_fmode`. Если переменная `_fmode` имеет значение `O_BINARY`, файлы будут открываться в бинарном режиме, иначе, если `_fmode` имеет значение `O_TEXT`, файлы открываются в текстовом режиме. Данные константы `O_...` определены в файле `fcntl.h`.

При успешном завершении `fopen` возвращает указатель на открытый поток `stream`. В случае ошибки функция возвращает нуль (`NULL`).

Чтение из файла символа или строки символов

На рис. 8.1. Представлена программа, выводящая на экран содержимое текстового файла

```
#include <stdio.h>
#include <conio.h>

 void main(void)
 {
    FILE *stream;
    char ch;
    /* создать файл для его изменения */
    stream = fopen("proba.txt","r");
    do
    {
       /* прочитать символ из файла */
       ch = fgetc(stream);
       /* вывести символ на экран */
       putch(ch);
    } while(ch!=EOF);
    fclose(stream);
}
```

Рис. 8.1. Программа выводит на экран содержимое текстового файла посимвольно

`fgets` считывает из потока `stream` строку символов и помещает ее в `s`. Ввод завершается после ввода `n-1` символа или при вводе символа перехода на следующую строку, смотря что произойдет раньше. В отличие от `gets`, `fgets` прекращает ввод строки при получении символа перехода на следующую строку. Нулевой байт добавляется в конец строки для определения ее конца (рис. 8.2).

```
#include <stdio.h>

 void main()
 {
    FILE *stream;
    char msg[20];
    /* создать файл для его изменения */
    stream = fopen("test.txt","r");
    /* ввести строку из файла */
    fgets(msg,strlen(string)+1,stream);
    /* напечатать строку */
    printf("%s",msg);
    fclose(stream);
}
```

Рис. 8.2. Программа выводит на экран содержимое текстового файла построчно

Запись символа или строки символов в файл

Для записи символа в файл необходимо указать сам символ, а также указатель на открытый для записи файл (рис. 8.3.).

```
void main()
{
   FILE *stream;
   stream = fopen("test.txt","w+");
   char msg[] = "Здравствуй мир";
   int i=0;
   while(msg[i])
   {
      fputc(msg[i],stream); //запись в файл stream
      i++;
   }
   fclose(stream);
}
```

Рис. 8.3. Программа записывает построчно в файл

Функция `fputs` копирует строку, ограниченную нулевым байтом в поток `stream`. Она не добавляет в конец строки символ перехода на новую строку и не выводит нулевой символ.

```
void main()
{
   FILE *stream;
   stream = fopen("test.txt","w+");
   char msg[] = "Здравствуй мир";
   fputs(msg,stream); //запись в файл stream
   fclose(stream);
}
```

Рис. 8.4. Программа записывает строку в файл

Смещение внутри файла

```
int fseek(FILE * stream, long offset, int fromwhere);
```

Функция `fseek` устанавливает адресный указатель файла, соответствующий потоку `stream`, в новую позицию, которая расположена по смещению `offset` относительно места в файле, определяемого параметром `fromwhere`. Параметр `fromwhere` может иметь одно из трех значений 0, 1 или 2, которые представлены тремя символическими константами (определенными в файле `stdio.h`), следующим образом:

Таблица 5

Значения параметра fromwhere функции fseek

Параметр	Размещение в файле `frowmwhere`
`SEEK_SET (0)`	начало файла
`SEEK_CUR (1)`	позиция текущего указателя файла
`SEEK_END (2)`	конец файла `(EOF)`

Функция `fseek` используется с операциями ввода/вывода в поток (рис. 8.5.).

```
#include <stdio.h>

 void main(void) {
    FILE *stream;
    char string[] = "Тестовый пример";
    char msg[20];
    /* создать файл для его изменения */
    stream = fopen("proba.txt","w+");
    /* записать в файл данные */
    fputs(string,stream);
    /* перейти в начало файла */
    fseek(stream,0,SEEK_SET);
    /* ввести строку из файла */
    fgets(msg,strlen(string)+1,stream);
    /* напечатать строку */
    printf("%s",msg);
    fclose(stream);
}
```

Рис. 8.5. Программа, иллюстрирующая работу функции `fseek`

Закрытие файла

После работы с файлом его необходимо закрыть. Для этого используется функция `fclose`, в качестве параметра принимающая указатель на открытый файл (см. рис. 8.1, 8.2, 8.3, 8.4 и 8.5).

9. КОМПОНОВКА ПРОГРАММ И ПРЕПРОЦЕССОР

9.1. Компоновка программ

Программа – это, прежде всего, текст на языке C++. С помощью компилятора текст преобразуется в исполняемый файл – форму, позволяющую компьютеру выполнять программу.

Если рассмотрим этот процесс чуть более подробно, то выяснится, что обработка исходных файлов происходит в три этапа. Сначала файл обрабатыва-

ется препроцессором, который выполняет операторы `#include`, `#define` и еще несколько других. После этого программа все еще представлена в виде текстового файла, хотя и измененного по сравнению с первоначальным. Затем, на втором этапе, компилятор создает так называемый объектный файл. Программа уже переведена в машинные инструкции, однако еще не полностью готова к выполнению. В объектном файле имеются ссылки на различные системные функции и на стандартные функции языка C++. Например, выполнение операции `new` заключается в вызове определенной системной функции. Даже если в программе явно не упомянута ни одна функция, необходим, по крайней мере, один вызов системной функции – завершение программы и освобождение всех принадлежащих ей ресурсов.

На третьем этапе компиляции к объектному файлу подсоединяются все функции, на которые он ссылается. Функции тоже должны быть скомпилированы, т.е. переведены на машинный язык в форму объектных файлов. Этот процесс называется **компоновкой**, и как раз его результат и есть исполняемый файл.

Системные функции и стандартные функции языка C++ заранее откомпилированы и хранятся в виде библиотек. **Библиотека** – это некий архив объектных модулей, с которым удобно компоновать программу.

Основная цель многоэтапной компиляции программ – возможность компоновать программу из многих файлов. Каждый файл представляет собой законченный фрагмент программы, который может ссылаться на функции, переменные или классы, определенные в других файлах. Компоновка объединяет фрагменты в одну «самодостаточную» программу, которая содержит все необходимое для выполнения.

Проблема использования общих функций и имен

В языке C++ существует строгое правило, в соответствии с которым прежде чем использовать в программе имя или идентификатор, его необходимо определить. Рассмотрим на примере функций. Для того чтобы имя функции стало известно программе, его нужно либо объявить, либо определить.

Объявление функции состоит лишь из ее прототипа, т.е. имени, типа результата и списка аргументов. Объявление функции задает ее формат, но не определяет, как она выполняется (рис. 9.1.)

```
double sqrt(double x);   // функция sqrt
long fact(long x);       // функция fact
 // функция PrintBookAnnotation
void PrintBookAnnotation(const Book& book);
```

Рис. 9.1. Примеры объявления функций

Определение функции – это определение того, как функция выполняется. Оно включает в себя тело функции, программу ее выполнения (рис. 9.2.).

Определение функции играет роль объявления ее имени, т.е. если в начале файла определена функция `fact`, в последующем тексте функций и классов ею можно пользоваться. Однако если в программе функция `fact` используется в нескольких файлах, такое построение программы уже не подходит. В программе должно быть только одно определение функции.

```
long fact(long x)
{
     if (x == 1)
          return 1;
     else
          return x * fact(x - 1);
}
```

Рис. 9.2. Функция вычисления факториала

Удобно было бы поместить определение функции в отдельный файл, а в других файлах в начале помещать лишь объявление, прототип функции.

```
// начало файла main.cpp
long fact(long);    // прототип функции
int main(){
     int x10 = fact(10); } // вызов функции
// конец файла main.cpp

// начало файла fact.cpp
long fact(long x) // определение функции вычисления факториала
{
     if (x == 1)
          return 1;
     else
          return x * fact(x - 1);}
// конец файла fact.cpp
```

Рис. 9.3. Программа, состоящая более чем из одного файла

Компоновщик объединит оба файла в одну программу.

Таким образом, в начале каждого файла будут сосредоточены прототипы всех используемых функций.

Программа работать будет, однако писать ее не очень удобно.

В начале каждого файла придется повторять довольно большие одинаковые куски текста. Помимо того, что это утомительно, очень легко допустить ошибку. К тому же любые изменения в объявлении функции придется вносить во все файлы.

Использование включаемых файлов

В языке C++ реализовано удобное решение. Можно поместить объявления классов и функций в отдельный файл и включать этот файл в начало других файлов с помощью оператора `#include`.

```
#include "function.h"
```

Фактически оператор `#include` подставляет содержимое файла `function.h` в текущий файл перед тем, как начать его компиляцию. Эта подстановка осуществляется во время первого прохода компилятора по программе – препроцессора. Файл `function.h` называется файлом заголовков.

В такой же файл заголовков можно поместить объявления классов и включать его в другие файлы, там, где создаются объекты этих классов.

Таким образом, текст программы на языке C++ помещается в файлы двух типов – **файлы заголовков** и **файлы программ**. В большинстве случаев имеет смысл каждый класс помещать в отдельный файл, вернее, два файла – файл заголовков для объявления класса и файл программ для определения класса. Имя файла обычно состоит из имени класса. Для файла заголовков к нему добавляется окончание `".h"` (иногда, особенно в системе Unix, ".hh" или ".H"). Имя файла программы – опять-таки имя класса с окончанием `".cpp"` (иногда ".cc" или ".C").

Включение файлов может быть вложенным, т.е. файл заголовков может сам использовать оператор `#include`. Текст файла `function.h` показан на рис. 9.4.

```
#ifndef __FUNCTION_H__
#define __FUNCTION_H__
// Включить файл стандартной библиотеки
#include <string.h>
int f1(int);
void f2(char*, char*);
#endif
```

Рис. 9.4. Заголовочный файл `function.h`

Необходимо обратить внимание на первые две и последнюю строки этого файла. Оператор `#ifndef` начинает блок так называемой условной компиляции, который заканчивается оператором `#endif`. Блок условной компиляции – это кусок текста, который будет компилироваться, только если выполнено определенное условие. В данном случае условие заключается в том, что символ __FUNCTION_H__ не определен. Если этот символ определен, текст между `#ifndef` и `#endif` не будет включен в программу. Первым оператором в блоке условной компиляции стоит оператор `#define`, который определяет символ __FUNCTION_H__ как пустую строку.

Компилятор поставляется с набором файлов заголовков, которые описывают все стандартные функции и классы. При включении стандартных файлов обычно используют немного другой синтаксис:

```
#include <string.h>
```

9.2. Препроцессор

В языке C++ имеется несколько операторов, которые начинаются со знака #: `#include`, `#define`, `#undef`, `#ifdef`, `#else`, `#if`, `#pragma`. Все они обрабатываются так называемым **препроцессором.**

Иногда препроцессор называют макропроцессором, поскольку в нем определяются макросы. Директивы препроцессора начинаются со знака #, который должен быть первым символом в строке после пробелов.

Определение макросов

Форма директивы `#define`

```
#define имя определение
```

определяет макроимя. Везде, где в исходном файле встречается это имя, оно будет заменено его определением. Например, текст:

```
#define NAME "database"
Connect(NAME);
```

после препроцессора будет заменен на

```
Connect("database");
```

По умолчанию имя определяется как пустая строка, т.е. после директивы

```
#define XYZ
```

макроимя `XYZ` считается определенным со значением – пустой строкой.

Другая форма `#define`

```
#define имя ( список_имен ) определение
```

определяет макрос – текстовую подстановку с аргументами

```
#define max(X, Y) ((X > Y) ? X : Y)
```

Текст `max(5, a)` будет заменен на

```
 ((5 > a) ? 5 : a)
```

В большинстве случаев использование макросов (как с аргументами, так и без) в языке C++ является признаком непродуманного дизайна. В языке C макросы были действительно важны, и без них было сложно обойтись. В C++ при наличии констант и шаблонов макросы не нужны. Макросы осуществляют текстовую подстановку, поэтому они в принципе не могут осуществлять никакого контроля использования типов. В отличие от них в шаблонах контроль типов полностью сохранен. Кроме того, возможности текстовой подстановки существенно меньше, чем возможности генерации шаблонов.

Директива `#undef` отменяет определение имени, после нее имя перестает быть определенным.

У препроцессора есть несколько макроимен, которые он определяет сам, их называют предопределенными именами. У разных компиляторов набор этих

имен различен, но два определены всегда: __FILE__ и __LINE__. Значением макроимени __FILE__ является имя текущего исходного файла, заключенное в кавычки. Значением __LINE__ – номер текущей строки в файле. Эти макроимена часто используют для печати отладочной информации.

Условная компиляция

Исходный файл можно компилировать не целиком, а частями, используя директивы условной компиляции (рис. 9.5.).

```
#if LEVEL > 3
     текст1
#elif LEVEL > 1
     текст2
#else
     текст3
#endif
```

Рис. 9.5. Использование условной компиляции

Предполагается, что LEVEL – это макроимя, поэтому выражение в директивах #if и #elif можно вычислить во время обработки исходного текста препроцессором. Блок условной компиляции должен завершаться директивой #endif.

В каком-то смысле директива #if похожа на условный оператор if. Однако, в отличие от него, условие – это константа, которая вычисляется на стадии препроцессора, и куски текста, не удовлетворяющие условию, просто игнорируются.

Директив #elif может быть несколько (либо вообще ни одной), директива #else также может быть опущена.

Директива #ifdef – модификация условия компиляции. Условие считается выполненным, если указанное после нее макроимя определено. Соответственно, для директивы #ifndef условие выполнено, если имя не определено.

Дополнительные директивы препроцессора

Директива #pragma используется для выдачи дополнительных указаний компилятору. Например, не выдавать предупреждений при компиляции, или вставить дополнительную информацию для отладчика. Конкретные возможности директивы #pragma у разных компиляторов различные.

Директива #error выдает сообщение и завершает компиляции. Например, директива #error выдаст сообщение и не даст откомпилировать исходный файл, если макроимя unix не определено (рис. 9.6).

```
#ifndef unix
#error "Программу можно компилировать только для Unix!"
#endif
```

Рис. 9.6. Использование директивы `#error`

Директива `#line` изменяет номер строки и имя файла, которые хранятся в предопределенных макроименах `__LINE__` и `__FILE__`.

Кроме директив, у препроцессора есть одна операция ##, которая соединяет строки, например `A ## B`.

10. СТРУКТУРЫ

10.1. Определение структур и доступ к элементам

Структуры – это составные типы данных, построенные с использованием других типов. На рис. 10.1 показано определение структуры время:

```
struct Time
{
     int hour;
     int minute;
     int second;
};
```

Рис. 10.1. Определение структуры `Time`

Ключевое слово `struct` начинает определение структуры. `Time` – тег (обозначение, имя-этикетка) структуры. Тэг структуры используется при объявлении переменных структур данного типа. В этом примере имя нового типа – `Time`. Имена, объявленные в фигурных скобках описания структуры – это элементы структуры. Элементы одной и той же структуры должны иметь уникальные имена, но две разные структуры могут содержать не конфликтующие элементы с одинаковыми именами. Каждое определение должно заканчиваться точкой с запятой.

Определение `Time` содержит три элемента типа `int` – `hour`, `minute`, `second`. Элементы структуры могут быть разного типа, и одна структура может содержать элементы многих разных типов. Структура не может содержать экземпляры самой себя. Например, элемент типа `Time` не может быть объявлен в определении структуры `Time`. Однако может быть включен указатель на другую структуру `Time`. Структура, содержащая элемент, который является указателем на такой же структурный тип, называется **структурой с самоадресацией.**

Определение структуры данных не резервирует никакого пространства в памяти; определение только создает новый тип данных, который используется

для объявления переменных. Переменные структуры объявляются так же, как переменные других типов. Строка

```
Time timeObject, timeArray[10], *timePtr;
```

объявляет `timeObject` переменной типа `Time`, `timeArray` – массивом с десятью элементами типа `Time`, а `timePtr` – указателем на объект типа `Time`.

Доступ к элементам структур

Для доступа к элементам структуры или класса используются операции селектора члена структуры или класса – операция точка(`.`) и операция стрелка(`->`). Операция точка обращается к элементу структуры (или класса) по имени переменной объекта или по ссылке на объект. Например, чтобы напечатать элемент `hour` структуры `timeObject` используется оператор

```
cout << timeObject.hour;
```

Операция стрелка, состоящая из знака минус (`-`) и знака больше (`>`), записанных без пробела, обеспечивает доступ к элементу структуры (или класса) через указатель на объект. Допустим, что указатель `timePtr` был уже объявлен как указывающий на объект типа `Time` и что адрес структуры `timeObject` был уже присвоен `timePtr`. Тогда, чтобы напечатать элемент `hour` структуры `timeObject` с указателем `timePtr`, можно использовать оператор

```
cout << timePtr->hour;
```

Выражение `timePtr->hour` эквивалентно

```
(*timePtr).hour;
```

которое разыменовывает указатель и делает доступным элемент `hour` через операцию точка. Скобки нужны здесь потому, что операция точка имеет более высокий приоритет, чем операция разыменования указателя (`*`).

Можно определить переменную-структуру без определения отдельного типа (рис. 10.2.)

```
struct {
     double x;
     double y;
} coord;
```

Рис. 10.2. Создание переменной структуры без типа

Обратиться к атрибутам переменной `coord` можно `coord.x` и `coord.y`.

Использование структур

Программа на рис. 10.3 создает определенный пользователем тип структуры `Time` с тремя целыми элементами: `hour`, `minute` и `second`. Программа определяет единственную структуру типа `Time`, названную `dTime`, и использует операцию точка для присвоения элементам структуры начальных значе-

ний. Затем программа печатает введенное время в 24-часовом и 12-часовом формате. Функции печати принимают ссылки на постоянные структуры типа `Time`. Причина этого в том, что таким образом исключаются накладные расходы на копирование, связанные с передачей структур функциям по значению, а использование `const` предотвращает изменение структуры типа `Time` функциями печати

```
#include <iostream.h>

struct Time
{
     int hour;
     int minute;
     int second;
};

void print24Hours(const Time &t)
{
     cout << (t.hour <10 ? "0" : "") << t.hour << ":" << (t.minute
     < 10 ? "0" : "") << t.minute << ":" << (t.second <10 ? "0" :
     "") << t.second;
}

void print12Hours(const Time &t)
{
     cout << ((t.hour ==0 || t.hour == 12) ? 12 : t.hour %12) <<
     ":" << (t.minute < 10 ? "0" : "") << t.minute << ":" <<
     (t.second <10 ? "0" : "") << t.second << (t.hour < 12 ? " am"
     : " pm");
}

int main()
{
     Time dTime;
     cout << "Enter the time:" << endl;
     cin >> dTime.hour >> dTime.minute >> dTime.second;
     cout << "Time is ";
     print24Hours(dTime);
     cout << endl;
     cout << "that is the same ";
     print12Hours(dTime);
     cout << endl;
     return 0;
}
```

Рис. 10.3. Использование структур

Существуют препятствия созданию новых типов данных указанным способом с помощью структур. Поскольку инициализация структур специально не требуется, можно иметь данные без начальных значений и вытекающие из это-

го проблемы. Даже если данные получили начальные значения, возможно, это было сделано неверно. Неправильные значения могут быть присвоены элементам структуры, потому что программа имеет прямой доступ к данным. Не существует никакого интерфейса, гарантирующего, что программист правильно использует тип данных и что данные являются непротиворечивыми.

Существуют и другие проблемы, связанные со структурами. Структуры не могут быть напечатаны как единое целое, только по одному элементу с соответствующим форматированием каждого. В С структуры нельзя сравнивать в целом, их нужно сравнивать элемент за элементом.

10.2. Битовые поля

В структуре можно определить размеры атрибута с точностью до бита. Традиционно структуры используются в системном программировании для описания регистров аппаратуры. В них каждый бит имеет свое значение. Не менее важной является возможность экономии памяти – так как минимальный тип атрибута структуры - это байт (`char`), который занимает 8 битов. До сих пор, несмотря на мегабайты и гигабайты оперативной памяти, используемые в современных компьютерах, существует немало задач, где каждый бит на счету.

Если после описания атрибута структуры поставить двоеточие и затем целое число, то это число задает количество битов, выделенных под данный атрибут структуры. Такие атрибуты называют **битовыми полями.** Структура на рис. 10.4 хранит в компактной форме дату и время дня с точностью до секунды.

```
struct TimeAndDate
{
unsigned hours    :5; // часы от 0 до 24
unsigned mins     :6; // минуты от 0 до 60
unsigned secs     :6; // секунды от 0 до 60
unsigned weekDay  :3; // день недели
unsigned monthDay :6; // день месяца от 1 до 31
unsigned month    :5; // месяц от 1 до 12
unsigned year     :8; // год от 0 до 100
};
```

Рис. 10.4. Структура, хранящая дату и время

Одна структура `TimeAndDate` требует всего 39 битов, т.е. 8 байтов (один байт — 8 битов). Если бы для каждого атрибута этой структуры использовался тип `char`, то потребовалось бы 7 байтов.

10.3. Объединения

Особым видом структур данных является объединение (рис. 10.5). Определение объединения напоминает определение структуры, только вместо ключевого слова `struct` используется `union`:

```
union number {
     short sx;
     long lx;
     double dx;
};
```

Рис. 10.5. Объявление объединения в программе

В отличие от структуры, все атрибуты объединения располагаются по одному адресу. Под объединение выделяется столько памяти, сколько нужно для хранения наибольшего атрибута объединения. Объединения применяются в тех случаях, когда в один момент времени используется только один атрибут объединения и, прежде всего, для экономии памяти. Предположим, что нужно определить структуру, которая хранит "универсальное" число, т.е. число одного из предопределенных типов, и признак типа. Это можно сделать так, как показано на рис. 10.6:

```
struct Value {
  enum NumberType { ShortType, LongType,doubleType };
  NumberType type;
  short sx;       // если type равен ShortType
  long lx;        // если type равен LongType
  double dx;      // если type равен DoubleType
};
```

Рис. 10.6. Объявление структуры для хранения числа любого типа

Атрибут `type` содержит тип хранимого числа, а соответствующий атрибут структуры – значение числа. На рис. 10.7 приведен пример использования этой структуры

```
Value shortVal;
shortVal.type = Value::ShortType;
shortVal.sx = 15;
```

Рис. 10.7. Использование структуры для хранения числа любого типа

Хотя память выделяется под все три атрибута `sx`, `lx` и `dx`, реально используется только один из них. Сэкономить память можно, используя объединение (см. рис. 10.8.).

```
#include <iostream.h>

struct Value
{
     enum NumberType { ShortType, LongType, DoubleType };
     NumberType type;
     union number
     {
          short sx;       // если type равен ShortType
          long lx;        // если type равен LongType
          double dx;      // если type равен DoubleType
     } val;
};

main()
{
     Value shortVal;
     shortVal.type = Value::ShortType;
     shortVal.val.sx= 127000000;
     cout<<shortVal.type<<endl;
     cout<<shortVal.val.sx<<endl;
     return 0;
}
```

Рис. 10.8. Использование объединения для универсального типа данных

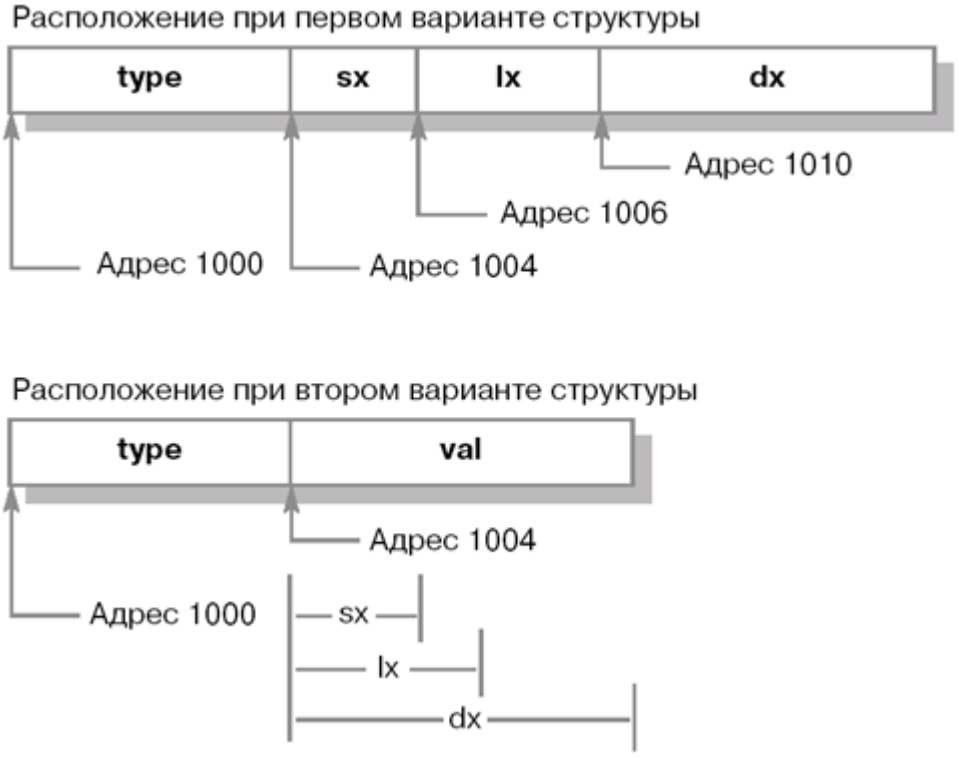


Рис. 10.9. Выделение памяти под структуру и объединение

Теперь память выделена только для максимального из этих трех атрибутов (в данном случае dx). Однако и обращаться с объединением следует осторожно. Поскольку все три атрибута делят одну и ту же область памяти, изменение одного из них означает изменение всех остальных. На рис. 10.9 поясняется выделение памяти под объединение. В обоих случаях предполагается, что структура расположена по адресу 1000. Объединение располагает все три своих атрибута по одному и тому же адресу.

10.4. Построение связных списков на основе структур с самоадресацией

Связный список - это разновидность линейных структур данных, представляющая собой последовательность элементов, обычно отсортированную в соответствии с заданным правилом. Последовательность может содержать любое количество элементов, поскольку при создании списка используется динамическое распределение памяти.

Каждый элемент связного списка представляет собой отдельный объект, содержащий поле для хранения информации и указатель на следующий элемент списка (а в случае двусвязного списка в объекте хранится также указатель на предыдущий элемент).

Связные списки существенно облегчают задачи удаления элемента (достаточно просто переставить связи) и вставки нового элемента (достаточно внести изменения в поле связи одного элемента). Так же намного проще изменить порядок следования элементов. Но при этом получить доступ к элементу становится намного сложнее (необходимо пройти все предыдущие элементы). Возникает сложность и при необходимости просто подсчитать количество элементов в списке, для этого необходимо пройти весь список.

Схема, изображающая связный и двусвязный списки из трех элементов, изображена на рис. 10.10.

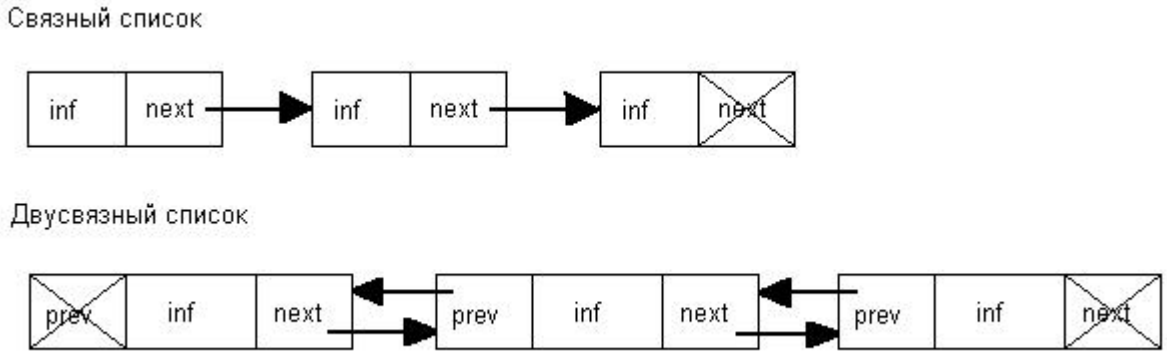


Рис. 10.10. Построение связных списков

Передвижение по списку осуществляется по указателям, которые указывают на соседние элементы списка. При добавлении нового элемента к списку необходимо динамически выделить под него память и присвоить соответст-

вующие значения указателям соседних элементов, а также указателям самого созданного элемента.

Создание простого связного списка

В качестве примера рассмотрим связный список, хранящий целые числа. Элементы списка сортируются по возрастанию в зависимости от величины числа, которое хранится в данном элементе.

Каждый элемент списка является экземпляром структуры, которая описывается так, как показано на рис. 10.11:

```
struct TNode {
   int value;
   TNode* pnext;
};
```

Рис. 10.11. Описание элемента списка

Здесь `pnext` - указатель на следующий элемент списка, а `value` - поле для хранения информации.

Для добавления нового элемента к списку используется функция `add2list()`. Ее описание приведено на рис. 10.12.

```
void add2list(TNode **pphead, int val)
{
   TNode **pp = pphead, *pnew;
   while(*pp)
   {
      if(val < (*pp)->value)
         break;
      else
         pp = &((*pp)->pnext);
   }
   pnew = new TNode;
  pnew->value = val;
  pnew->pnext = *pp;
   *pp = pnew;
}
```

Рис. 10.12. Добавление элемента в список

Здесь `pphead` - это указатель на указатель на голову списка (т.е. на его первый элемент), `val` - значение, которое необходимо добавить в список. При добавлении нового элемента списка учитывается величина числа, которое будет помещено в информационное поле создаваемого элемента. Таким образом, упорядоченность элементов списка, которые расположены по возрастанию, не нарушается, и список остается отсортированным.

Вывести список на экран можно с помощью функции `print`(). Реализация этой функции представлена на рис. 10.13.

```
void print(TNode *phead)
{
   TNode* p = phead;
   while(p)
   {
      cout << p->value << ' ';
      p = p->pnext;
   }
   cout << endl;
}
```

Рис. 10.13. Печать списка

Поскольку память для элементов списка выделяется динамически, после окончания работы со списком необходимо явно удалить все его элементы. Функция удаления элементов списка может быть реализована так, как показано на рис. 10.14.

```
void deleteList(TNode *phead)
{
   if(phead)
   {
      deleteList(phead->pnext);
      if(phead)
         delete phead;
   }
}
```

Рис. 10.14. Удаление списка

Протестировать функции для работы со связным списком можно следующим образом (рис. 10.15).

```
int main()
{
   TNode *phead = 0;
   srand(time(0));
   for(int i = 0; i < 10; ++i)
      add2list(&phead, rand() % 100);
   cout << "Elements of the list:" << endl;
   print(phead);
   deleteList(phead);
   return 0;
}
```

Рис. 10.15. Работа со связным списком

Стек

Одним из примеров односвязного списка может служить стек. Механизм функционирования стека хорошо отражает другое его название – список типа `LIFO` (`last in first out` – последним вошел – первым вышел). При работе со стеком предполагаются только две операции: занесение элемента в вершину стека и удаление элемента, находящегося в вершине стека. Таким образом, операция удаления элемента из стека применима только к элементу, помещенному в стек последним. А значит, любой элемент не может быть удален из стека раньше, чем будут удалены все элементы, помещенные в стек после него.

Примером применения стека служит задача проверки правильности расстановки скобок. Проходя выражение слева направо, мы, при появлении открывающейся скобки, помещаем ее в стек, а в случае появления закрывающейся скобки, пытаемся удалить элемент из вершины стека. Если в ходе работы возникает ситуация, когда стек пуст, а мы имеем закрывающуюся скобку, это означает, что выражение составлено неправильно. В конце работы стек должен быть пуст, иначе мы так же делаем вывод о несоответствии скобок в выражении.

Еще один пример использования стека. Когда происходит вызов функции, вызываемая функция должна знать, как вернуться в вызывающую функцию; в этом случае адрес возвращения помещается в стек. Если происходит ряд обращений к функциям, то последовательность адресов возвращения помещается в стек по принципу `LIFO` для того, чтобы каждая функция могла вернуться в свою вызывающую функцию. Стеки поддерживают как рекурсивные вызовы функций, так и обычные нерекурсивные.

Очереди

Другой стандартной структурой данных является очередь. Очередь аналогична очереди людей в магазине: первый клиент в ней обслуживается первым, а другие клиенты ожидают своей очереди. Узлы очереди удаляются только из головы очереди, а помещаются в очередь только в ее хвосте. Это структура данных типа `FIFO` (`first in first out` – первым вошел первым вышел).

У очередей имеется множество применений в вычислительных системах. У большинства компьютеров имеется только один процессор, в один и тот же момент времени может быть обслужен только один процесс. Запросы других процессов помещаются в очередь. Каждый запрос постепенно продвигается по очереди вперед по мере того, как происходит обслуживание пользователей. Запрос в начале очереди является очередным кандидатом на обслуживание.

Информационные пакеты также ожидают своей очереди в компьютерных сетях. Пакет может поступить на узел сети в любой момент времени, а затем он должен быть отправлен к следующему узлу сети по направлению своего конечного пункта назначения. Узел маршрутизации (т.е. узел, хранящий информацию о маршруте) направляет в каждый момент времени один пакет, поэтому

пакеты помещаются в очередь до тех пор, пока программа маршрутизации не обработает их.

Деревья

Связные списки, стеки и очереди являются линейными структурами данных. Дерево является нелинейной двумерной структурой данных с особыми свойствами. Узлы дерева могут содержать два и более указателей связи. Бинарные деревья содержат по два указателя. Или один из них, или оба могут быть нулевыми. Корневой узел является первым узлом дерева. Каждый указатель связи в корневом узле ссылается на дочерний узел или узел-потомок. Левый узел-потомок является первым узлом в левом поддереве, а правый узел-потомок является первым узлом в правом поддереве. Узлы-потомки, порожденные одним каким-либо узлом, называются родственными узлами (или узлами-братьями). Узел без узлов-потомков называется листом дерева или концевым узлом. Деревья обычно рисуют сверху вниз, начиная с корневого узла, то есть противоположно тому, как растут деревья в природе.

11. КЛАССЫ И АБСТРАГИРОВАНИЕ ДАННЫХ

11.1. Определения классов

Классы – это определяемые пользователем типы данных. Каждый класс содержит данные и функции, манипулирующие с этими данными. После определения они могут использоваться наравне со встроенными типами.

Классы предоставляют программисту возможность моделировать объекты, которые имеют атрибуты (данные-элементы) и варианты поведения или операции (функции-элементы). Типы, содержащие данные-элементы и функции-элементы, обычно определяются в C++ с помощью ключевого слова `class`.

Функции-элементы иногда в других объектно-ориентированных языках называют методами, они вызываются в ответ на сообщения, посылаемые объекту. Сообщение соответствует вызову функции-элемента.

Когда класс определен, имя класса может быть использовано для объявления объекта этого класса (см. рис. 11.1).

Определение класса `Time` начинается с ключевого слова `class`. Тело определения класса заключается в фигурные скобки (`{ }`). Определение класса заканчивается точкой с запятой. Определение класса `Time` содержит три целых элемента `hour`, `minute` и `second`.

Метки `public:` (открытая) и `private:` (закрытая) называются **спецификаторами доступа** к элементам. Любые данные-элементы и функции-элементы, объявленные после спецификатора доступа к элементам `public:` (и до следующего спецификатора доступа к элементам), доступны при любом об-

ращении программы к объекту класса Time. Любые данные-элементы и функции-элементы, объявленные после спецификатора доступа к элементам private: (и до следующего спецификатора доступа к элементам), доступны только функциям-элементам этого класса. Спецификаторы доступа к элементам всегда заканчиваются двоеточием (:) и могут появляться в определении класса много раз и в любом порядке.

```
class Time {
public:
    Time () ;
    void setTime (int, int, int);
    void print24Hours();
    void print12Hours();
 private:
    int hour;   // 0-23
    int minute; // 0 -59
    int second; // 0-59
   };
```

Рис. 11.1. Определение класса Time

Определение класса в программе содержит после спецификатора доступа к элементам public: прототипы следующих четырех функций-элементов: Time, setTime, Print24Hours и print12Hours. Это – **открытые функции-элементы** или **открытый интерфейс услуг класса.** Эти функции будут использоваться клиентами класса (т.е. частями программы, играющими роль пользователей) для манипуляций с данными этого класса.

Обратите внимание на функцию-элемент с тем же именем, что и класс. Она называется конструктором этого класса. **Конструктор** — это специальная функция-элемент, которая инициализирует данные-элементы объекта этого класса. Конструктор класса вызывается автоматически при создании объекта этого класса. Обычно класс имеет несколько конструкторов; это достигается посредством перегрузки функции, т.е. конструкторы должны иметь разный набор принимаемых значений.

После спецификатора доступа к элементам private: следуют три целых элемента. Это значит, что эти данные-элементы класса являются доступными только функциям-элементам класса. Таким образом, данные-элементы могут быть доступны только четырем функциям, прототипы которых включены в определение этого класса. Обычно данные-элементы перечисляются в части private, а функции-элементы — в части public. Можно иметь функции-элементы private и данные public, последнее не типично и считается в программировании дурным вкусом.

Когда класс определен, его можно использовать в качестве типа в объявлениях, например, так, как показано на рис 11.2.

```
Time sunset,          // объект типа Time
arrayOfTimes[5],      // массив объектов типа Time
*pointerToTime,       // указатель на объект типа Time
&dTime = sunset;      //ссылка на объект типа Time
```

Рис. 11.2. Использование типа Time в программе

Имя класса становится новым спецификатором типа. Может существовать множество объектов класса, как и множество переменных типа, например, такого, как `int`. Программист по мере необходимости может создавать новые типы классов. Это одна из многих причин, по которым C++ является **расширяемым языком**.

Программа на рис. 11.3 – 11.6 использует класс `Time`. Эта программа создает единственный объект класса `Time`, названный `t`. Когда объект создается, автоматически вызывается конструктор `Time`, который явно присваивает нулевые начальные значения всем данным-элементам закрытой части `private`. Затем печатается время в 24-х часовом и 12-ти часовом форматах, чтобы подтвердить, что элементы получили правильные начальные значения. После этого с помощью функции-элемента `setTime` устанавливается время, и оно снова печатается в обоих форматах. Затем функция-элемент `setTime` пытается дать данным-элементам неправильные значения, и время снова печатается в обоих форматах.

```
//time1.h**********************************************

#ifndef TIME1_H
#define TIME1_H

// Определение абстрактного типа данных (АТД) Time
class  Time{
public:
Time();   // конструктор
void setTime(int, int, int);    // установка часов, минут и секунд
void print24Hours();    //печать времени в 24-х часовом формате
void print12Hours() ;    //печать времени в 12-ти часовом формате
private:
int hour;  //0-23
int minute;  //0-59
int second;  //0-59
};

#endif
//**********************************************************
```

Рис. 11.3. Определение класса Time

```
//time1.cpp  ***********************************************
#include “time1.h”
#include <iostream.h>
// Конструктор Time присваивает нулевые начальные значения каждому
//элементу данных. Обеспечивает согласованное начальное состояние всех
//объектов
Time::Time()
{
     hour = minute = second =0;
}

// Задание нового значения Time в виде 24-х часового формата
// Проверка правильности значений данных.
// Обнуление неверных значений
void Time::setTime(int h, int m, int s) {
    hour = (h >= 0 && h < 24) ? h : 0;
    minute = (m >= 0 && m < 60) ? m : 0;
    second = (s >= 0 && s < 60) ? s : 0; }

// Печать времени в виде 24-х часового формата
void Time:: print24Hours ( ) {
    cout << (hour < 10 ? "0" :"") << hour« ":"
    << (minute < 10 ? "0": "") << minute << ":" << (second < 10 ?
    "0" : "") << second<<endl; }

// Печать времени в 12-ти часовом формате
void Time:: print12Hours() {
    cout << ( (hour ==0|| hour ==12) ? 12 : hour % 12) << ":"
    << (minute < 10 ? "0": "") << minute << ":" << (second < 10 ?
    "0" : "") <<
    second << (hour < 12 ? " AM" : " PM")<<endl; }
//*********************************************************
```

Рис. 11.4. Определение функций класса Time

Клиенты класса используют класс, не зная внутренних деталей его реализации. Если реализация класса изменяется (например, с целью улучшения производительности), интерфейс класса остается неизменным и исходный код клиента класса не требует изменений. Это значительно упрощает модификацию систем.

В программе на рис. 11.5 конструктор `Time` просто присваивает начальные значения, равные 0, данным-элементам, (т.е. задает 24-х часовой формат времени, эквивалентное 12 am). Это гарантирует, что объект при его создании находится в известном состоянии. Неправильные значения не могут храниться в данных-элементах объекта типа `Time`, поскольку конструктор автоматически вызывается при создании объекта типа `Time`, а все после-

дующие попытки изменить данные-элементы тщательно рассматриваются функцией setTime. Поэтому функции-элементы обычно короче, чем обычные функции в программах без объектной ориентации, потому что достоверность данных, хранимых в данных-элементах, идеально проверена конструктором и функциями-элементами, которые сохраняют новые данные.

```
//my.cpp  ************************************************
// Формирование  проверки простого класса Time
#include “time1.h”
#include <iostream.h>
main ()
(
    Time t;      // определение экземпляра объекта t класса Time
    cout<<"Начальное значение 24 часового формата времени равно ";
    t.print24Hours();
    cout <<"Начальное значение 12 часового формата времени равно";
    t.print12Hours() ;
    t.setTime(13, 27, 6);
    cout << endl << "24 часовой формат времени после SetTime ";
    t.print24Hours();
    cout << endl << "12-ти часовой формат времени после setTime
    равно ";
    t.print12Hours() ;
    t.setTime(99, 99, 99); //попытка установить неправильные значения
    cout <<"После попытки неправильной установки: "<< endl;
    cout<< "24 часовой формат времени: ";
    t.print24Hours ();
    cout << "12 часовой формат  времени:   ";
    t.print12Hours ();
    cout << endl;
    return 0;
    }
//*********************************************************
```

Рис. 11.5. Использование класса Time

```
Output:
//вывод на экран ****************************************
Начальное значение 24 часового формата равно 00:00:00
Начальное значение 12 часового формата  времени равно  12:00:00 AM

24 часовой формат  времени после  setTime равно  13:27:06
12 часовой формат  времени после  setTime равно 1:27:06 PM

После попытки неправильной    установки:
24 часовой формат  времени:   00:00:00
12 часовой формат  времени:   12 : 00 : 00 AM
//*********************************************************
```

Рис. 11.6. – Результат работы программы на рис. 11.3 – 11.5

Данные-элементы класса не могут получать начальные значения в теле класса, где они объявляются. Эти данные-элементы должны получать начальные значения с помощью конструктора класса или им можно присваивать значения через функции.

Функция с тем же именем, что и класс, но со стоящим перед ней символом тильда (~), называется **деструктором** этого класса. Деструктор производит «завершающие служебные действия» над каждым объектом класса перед тем, как память, отведенная под этот объект, будет повторно использована системой.

Объявление класса содержит объявления данных-элементов и функций-элементов класса. Объявления функций-элементов являются прототипами функций. Функции-элементы могут быть описаны внутри класса, но хороший стиль программирования заключается в описании функций вне определения класса.

Когда функция-элемент описывается после соответствующего определения класса (см. рис. 11.4), имя функции предваряется именем класса и **бинарной операцией разрешения области действия** (::).Поскольку разные классы могут иметь элементы с одинаковыми именами, операция разрешения области действия «привязывает» имя элемента к имени класса, чтобы однозначно идентифицировать функции-элементы данного класса.

Несмотря на то, что функция-элемент, объявленная в определении класса, может быть описана вне этого определения, эта функция-элемент все равно имеет **областью действия класс**, т.е. ее имя известно только другим элементам класса пока к ней обращаются посредством объекта класса, ссылки на объект класса или указателя на объект класса.

Интересно, что функции-элементы `print24Hours` и `print12Hours` не получают никаких аргументов. Это происходит потому, что функции-элементы неявно знают, что они печатают данные-элементы определенного объекта типа `Time`, для которого они активизированы. Это делает вызовы функций-элементов более краткими, чем соответствующие вызовы функций в процедурном программировании. Это уменьшает также вероятность передачи неверных аргументов, неверных типов аргументов или неверного количества аргументов.

11.2. Отделение интерфейса от реализации

Функции, которыми класс снабжает внешний мир, предваряются меткой `public`. Открытые функции реализуют все возможности класса, необходимые для его клиентов. Открытые функции класса называются **интерфейсом класса** или **открытым интерфейсом.**

Классы упрощают программирование, потому что клиент (или пользователь объекта класса) имеет дело только с операциями, **инкапсулированными** или **встроенными** в объект. Такие операции обычно проектируются ориентированными именно на клиента, а не на удобную реализацию. Клиентам нет не-

обходимости касаться реализации класса. Интерфейсы меняются, но не так часто, как реализации. При изменении реализации соответственно должны изменяться ориентированные на реализацию коды. А путем скрытия реализации мы исключаем возможность для других частей программы оказаться зависимыми от особенностей реализации класса.

Программа, реализующая и использующая класс `Time`, разбита на несколько файлов. При построении программы на C++ каждое определение класса обычно помещается в заголовочный файл, а определения функций-элементов этого класса помещаются в файлы исходных кодов с тем же базовыми именами.

Область действия класс и доступ к элементам класса

Данные-элементы класса (переменные, объявленные в определении класса) и функции-элементы (функции, объявленные в определении класса) имеют областью действия класс. Функции, не являющиеся элементом класса, имеют областью действия файл.

В области действия класс элементы класса непосредственно доступны всем функциям-элементам этого класса и на них можно ссылаться просто по имени. Вне области действия класс к элементам класса можно обращаться либо через имя объекта, либо ссылкой на объект, либо с помощью указателя на объект.

Переменные, определенные в функции-элементе, известны только этой функции. Если функция-элемент определяет переменную с тем же именем, что и переменная в области действия класс, последняя делается невидимой в области действия функция. Такая скрытая переменная может быть доступна посредством операции разрешения области действия с предшествующим этой операции именем класса. Невидимые глобальные переменные могут быть доступны с помощью унарной операции разрешения области действия (рис. 11.7)

```
float value = 1.2345;

void main()
{
     int value = 7;
     cout <<"local: " << value << endl;      //// 7
     cout << "global: " << ::value << endl; ///// 1,2345
     return 0;
}
```

Рис. 11.7. Использование унарной операции разрешения области действия

Операции, использованные для доступа к элементам класса, аналогичны операциям, используемым для доступа к элементам структуры. Операция выбора элемента точка (`.`) комбинируется для доступа к элементам объекта с именем объекта или со ссылкой на объект. Операция выбора элемента стрелка (`->`) комбинируется для доступа к элементам объекта с указателем на объект.

Программа на рис. 11.8 использует простой класс, названный Count, с открытым элементом данных x типа int и открытой функцией-элементом print(), чтобы проиллюстрировать доступ к элементам класса с помощью операций выбора элемента. Программа создает три экземпляра переменных типа Count — counter, counterRef (ссылка на объект типа Count) и counterPtr (указатель на объект типа Count). Переменная counterRef объявлена, чтобы ссылаться на counter, переменная counterPtr объявлена, чтобы указывать на counter. Следует отметить, что здесь элемент данных x сделан открытым просто для того, чтобы продемонстрировать способы доступа к открытым элементам. Как было уже установлено, данные обычно делаются закрытыми (private), чему и будем следовать в дальнейшем.

```
#include <iostream.h>
// Простой класс Count
class Count {
public:
int   x;
void print () { cout  << x << endl;}
};
main   (   )
{
Count counter,    // создается объект counter
*counterPtr = &counter,    // указатель на counter
&counterRef = counter;     // ссылка на counter
cout<<"Присваивание x значения 7 и печать по имени объекта:";
counter.x = 7; // присваивание значения 7 элементу данных x
counter.print ();   // вызов функции-элемента для печати
cout  << "Присваивание x значения 8 и печать по ссылке: ";
counterRef.x = 8;        // присваивание числа 8 элементу данных x
counterRef.print ();     // вызов функции-элемента для печати
cout << "Присваивание x значения 10 и печать по указателю:";
counterPtr->x = 10;       // присваивание 10 элементу данных x
counterPtr->print();      // вызов функции-элемента для печати
return 0;
}
```

Рис. 11.8 Использование операций выбора элемента класса

Управление доступом к элементам

Спецификаторы доступа к элементу public и private используются для управления доступом к данным-элементам класса и функциям-элементам класса (см. рис. 11.9). По умолчанию режим доступа для классов – private (закрытый). После каждого спецификатора режим доступа, определенный им, действует до следующего спецификатора или до завершающей фигурной скобки определения класса.

Закрытые элементы класса могут быть доступны только для функций-элементов (и дружественных функций) этого класса. Открытые элементы класса могут быть доступны для любых функций в программе.

Основная задача открытых элементов состоит в том, чтобы дать клиентам класса представление о возможностях (услугах), которые обеспечивает класс. Этот набор услуг составляет **открытый интерфейс класса**. Клиентов класса не должно интересовать, каким образом класс выполняет их задачи. Закрытые элементы класса и описания открытых функций-элементов недоступны для клиентов класса. Эти компоненты составляют реализацию класса.

Закрытые элементы класса доступны только через открытый интерфейс класса.

```
Time t;
t.hour = 7;                  //ошибка 'Time :: hour' недоступно
cout << "Minute is: " << t.minute; //ошибка 'Time :: minute' недоступно
```

Рис. 11.9. Доступ к закрытым элементам класса запрещен

Из того, что данные класса закрытые, не следует, что клиенты не могут изменять эти данные. Данные могут быть изменены функциями-элементами или друзьями этого класса.

Функции доступа

Функции-элементы можно разбить на ряд категорий. Функции, которые читают и возвращают значения закрытых данных-элементов. Функции, которые устанавливают значения закрытых данных-элементов. Функции, которые реализуют возможности класса. Функции, которые выполняют для класса различные вспомогательные операции, такие, как задание начальных значений объектам класса, присваивания объектам класса, преобразования между классами и встроенными типами или между классами и другими классами, выделение памяти для объектов класса.

Доступ к закрытым данным класса должен тщательно контролироваться использованием функций-элементов, называемых **функциями доступа**. Например, чтобы разрешить клиентам прочитать закрытое значение данных, класс может иметь функцию «получить» (`get`). Чтобы дать клиентам возможность изменять закрытые данные, класс может иметь функцию «установить» (`set`). (В классе `Time` это функция `setTime`)

Еще одним применением функций доступа является проверка истинности или ложности условий – такие функции называют предикатными функциями. Например, может быть определена функция `IsEmpty` для любого класса-контейнера – класса, способного содержать внутри себя много объектов, например, связного списка, очереди или стека. Программа проверяла бы функцию

`IsEmpty` прежде, чем пытаться прочесть очередной элемент из объекта контейнера.

Не все функции-элементы необходимо делать открытыми как часть интерфейса класса. Некоторые функции-элементы оставляются закрытыми и служат обслуживающими функциями-утилитами для других функций класса. **Функция-утилита** не является частью интерфейса класса. Она является закрытой функцией-элементом, которая поддерживает работу открытых функций-элементов класса. Функции-утилиты не предназначены для использования клиентами класса.

12. СПЕЦИАЛЬНЫЕ ЧЛЕНЫ КЛАССА

Каждый класс содержит три специальных члена – конструктор, деструктор и указатель на сам объект `this`.

Конструкторы

После создания объекта его элементы могут быть инициализированы с помощью функции конструктора. **Конструктор** – это функция-элемент класса с тем же именем, что и класс. Программист предусматривает конструктор, который затем автоматически вызывается при создании объекта (при создании экземпляра класса). **Данные-элементы класса не могут получать начальные значения в определении класса**. Они либо должны получить эти значения в конструкторе класса, либо их значения можно установить позже, после создания объекта. **Конструкторы не могут указывать типы возвращаемых значений или возвращать какие-либо значения.** Конструкторы можно перегружать, чтобы обеспечить множество начальных значений объектов класса.

Когда объявляется объект класса, между именем объекта и точкой с запятой можно в скобках указать **список инициализации элементов**. Эти начальные значения передаются в конструктор класса.

Конструктор может содержать значения аргументов по умолчанию. Тогда при определении класса `Time` прототип конструктора будет выглядеть следующим образом:

```
Time(int = 0, int = 0, int = 0);  //конструктор
                                  // с умолчанием
```

А в описании класса будет следующая функция-конструктор:

```
Time::Time(int hr, int min, int sec)
{
     setTime(hr, min, sec);
}
```

Рис. 12.1. Конструктор класса Time

Для каждого класса может существовать только один конструктор с умолчанием. В этой программе конструктор вызывает функцию-элемент `setTime` со значениями, передаваемыми конструктору или значениями по умолчанию, чтобы гарантировать, что значение, предназначенное для `hour` находится в диапазоне от 0 до 23, а значения для `minute` и `second` – в диапазоне от 0 до 59.

Создание экземпляров объектов Time может быть выполнено следующим образом:

```
Time t1, t2(2), t3(21,34), t4(12,25,42),
     t5(27,74,99);
```

Если для класса не определено никакого конструктора, компилятор создает конструктор с умолчанием. Такой конструктор не задает никаких начальных значений, так что после создания объекта нет никакой гарантии, что он находится в непротиворечивом состоянии.

Деструкторы

Деструктор — это специальная функция-элемент класса. Имя деструктора совпадает с именем класса, но перед ним ставится символ тильда (~). Это соглашение о наименовании появилось интуитивно, потому что операция тильда является поразрядной операцией дополнения, а по смыслу деструктор является дополнением конструктора.

Деструктор класса вызывается при уничтожении объекта — например, когда выполняемая программа покидает область действия, в которой был создан объект этого класса. На самом деле деструктор сам не уничтожает объект — он выполняет подготовку завершения перед тем, как система освобождает область памяти, в которой хранился объект, чтобы использовать ее для размещения новых объектов.

Деструктор не принимает никаких параметров и не возвращает никаких значений. Класс может иметь только один деструктор — перегрузка деструктора не разрешается.

Представленные до сих пор классы не были обеспечены деструкторами. На самом деле, деструкторы редко используются с простыми классами. Деструкторы имеют смысл в классах, использующих динамическое распределение памяти под объекты (например, для массивов и строк).

Когда вызываются конструкторы и деструкторы

Конструкторы и деструкторы вызываются автоматически. Последовательность, в которой выполняется вызов этих функций, зависит от последовательности, в которой процесс выполнения входит и выходит из областей действия, в которых создаются объекты. В общем случае вызовы деструктора выполняются в порядке, обратном вызовам конструктора. Однако классы памяти могут изменять последовательность вызовов деструкторов.

Конструкторы объектов, объявленных в глобальной области действия, вызываются раньше, чем любая функция данного файла (включая `main`) начинает выполняться. Соответствующие деструкторы вызываются, когда завершается `main`.

Конструкторы автоматических локальных объектов вызываются, когда процесс выполнения достигает места, где объекты объявляются. Соответствующие деструкторы вызываются, когда покидается область действия объектов (т.е. покидается блок, в котором эти объекты объявлены). Конструкторы и деструкторы для автоматических объектов вызываются каждый раз при входе и выходе из области действия.

Конструкторы статических локальных объектов вызываются сразу же, как только процесс выполнения достигает места, где объекты были впервые объявлены. Соответствующие деструкторы вызываются, когда завершается `main`.

Программа на рис. 12.2 – 12.4 показывает последовательность, в которой вызываются конструкторы и деструкторы объектов типа `CreateAndDestroy` в нескольких областях действия. Программа объявляет объект `first` в глобальной области действия. Его конструктор вызывается, как только программа начинает выполнение, а его деструктор вызывается по завершении программы, после того, как все другие объекты уничтожены.

```
// CREATE.H
// Определение класса CreateAndDestroy
//Функции-элементы определены в CREATE.CPP.

#ifndef CREATE_H
#define CREATE H

class CreateAndDestroy
{
public:
   CreateAndDestroy (int);         //конструктор
   ~CreateAndDestroy ();           //деструктор
private:
   int data;
};

#endif
```

Рис. 12.2. Объявление класса CreateAndDestroy

Функция `main` объявляет три объекта. Объекты `second` и `fourth` являются локальными автоматическими объектами, а объект `third` — статическим локальным объектом. Конструкторы каждого из этих объектов вызываются, когда процесс выполнения достигает места, где объекты были объявлены. Деструкторы объектов `fourth` и `second` вызываются в соответствующем по-

рядке, когда заканчивается `main`. Поскольку объект `third` — статический, он существует до завершения программы. Деструктор объекта `third` вызывается раньше деструктора для `first`, но после уничтожения всех других объектов.

```
// CREATE.CPP
// Определения функций-элементов для класса  CreateAndDestroy
#include  <iostream.h>
#include  "create.h"

CreateAndDestroy::CreateAndDestroy(int value){
   data = value;
   cout << "Объект " << data << "конструктор";}

CreateAndDestroy::~CreateAndDestroy(){
   cout << "Объект " << data « " деструктор " << endl;}
```

Рис. 12.3. Конструктор и деструктор класса CreateAndDestroy

```
// EXAMPLE.CPP
// Демонстрация последовательности, в которой вызываются
// конструкторы и деструкторы.
#include <iostream.h>
#include  "create.h"
void create(void);                                  //прототип
CreateAndDestroy first(1);                  //глобальный объект
main ( ){
   cout << (глобальный созданный до main)" << endl;
   CreateAndDestroy second(2);              // локальный объект
   cout << " (локальный автоматический в main) " << endl;
   static CreateAndDestroy third(3);  // локальный объект
   cout << " (локальный статический в main) " << endl;
   create ();                  // вызов функции создания объектов
   CreateAndDestroy fourth(4);             // локальный объект
   cout << " (локальный автоматический в main)" << endl;
    return 0;}

// Функция создания объектов
void create (void) {
   CreateAndDestroy fifth (5);
   cout << " (локальный автоматический в create) " << endl;
   static CreateAndDestroy sixth (6);
   cout << " (локальный статический в create) " << endl;
   CreateAndDestroy seventh (7);
   cout << " (локальный автоматический в create) " << endl;}
```

Рис. 12.4. – Использование класса CreateAndDestroy

Функция `create` объявляет три объекта — локальные автоматические объекты `fifth` и `seventh` и статический локальный объект `sixth`. Деструкторы для объектов `seventh` и `fifth` вызываются в соответствующем порядке по окончании `create`. Поскольку `sixth` — статический объект, он существует до завершения программы. Деструктор для `sixth` вызывается раньше деструктора для `third` и `first` , но после уничтожения всех других объектов.

```
Output:
Объект 1 конструктор (глобальный созданный до main)
Объект 2 конструктор (локальный автоматический в main)
Объект 3 конструктор (локальный статический в main)
Объект 5 конструктор (локальный автоматический в create)
Объект 6 конструктор (локальный статический в create)
Объект 1 конструктор (локальный автоматический в create)
Объект 7 деструктор
Объект 5 деструктор
Объект 4 конструктор (локальный автоматический в main)
Объект 4 деструктор
Объект 2 деструктор
Объект 6 деструктор
Объект 3 деструктор
Объект 1 деструктор
```

Рис. 12.5. Результат работы программы на рис. 12.2-12.4

Указатель `this`

```
#include <stdio.h>

class Test
{
public:
    Test(int = 0);
    void print () const;
private:
    int x;
};

Test::Test(int a){ x =a;}

void Test::print() const{
   printf("x = %d\nthis-> x= %d\n(*this).x = %d\n", x, this-
   >x, (*this).x);}

int main(){
    Test a(12);
    a.print();
    return 0;}
```

Рис. 12.6. Использование указателя `this`

Каждый объект содержит указатель на самого себя – называемый указателем `this`. Этот указатель неявно присутствует как аргумент во всех ссылках на элементы внутри объекта. Указатель `this` также можно использовать явно (см. рис. 12.6.). Каждый объект может определить свой собственный адрес с помощью ключевого слова `this`.

13. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

13.1. Композиция

Класс может включать в себя объекты других классов в качестве элементов. Такая возможность называется **композицией**, а объекты – **вложенными** (см. рис. 13.1). Когда объект входит в область действия, автоматически вызывается его конструктор и необходимо указать, как аргументы передаются конструкторам объектов-элементов. Объекты-элементы создаются в том порядке, в котором они **объявлены**, а не в том порядке, в котором они перечислены в списке инициализаторов элементов конструктора, и **до того**, как будут созданы объекты включающего их класса.

```
class Date{
public:
    Date(int = 1, int = 1, int = 1900);
……….
private:
    int day;
    int month;
    int year;};

Date :: Date(int d, int m, int y)
{   //здесь код основного конструктора}

class Employee{
public:
    Employee(char *, char *, int, int, int, int, int, int);
private:
    char lastName[25];
    char firstName[25];
    Date birthDate;
    Date hireDate;};

Employee ::Employee(char * fname, char *lname, int bday, int
                    bmonth, int byear, int hday, int hmonth,
                    int hyear): birthDate(bday, bmonth, byear),
                    hireDate(hday, hmonth, hyear)
{//здесь будет основной код конструктора}
```

Рис. 13.1. Отношения композиции

Класс Employee содержит закрытые данные-элементы lastName, firstName, birthDate, hireDate. Элементы birthDate и hireDate являются объектами класса Date, который содержит закрытые данные-элементы day, month и year.

Конструктор класса Employee принимает восемь аргументов. Двоеточие в заголовке отделяет **инициализаторы элементов** от списка параметров. Инициализаторы элементов указывают, что аргументы Employee передаются конструкторам объектов-элементов. Инициализаторы элементов в списке разделяются запятыми.

Объекты-элементы не нуждаются в задании начальных значений посредством инициализации элементов. Если инициализаторы не заданы, конструктор с умолчанием объекта-элемента будет вызван автоматически.

Если вложенный объект не имеет конструктора по умолчанию, то при попытке создать объект-элемент возникнет ошибка.

13.2. Друзья класса. Дружественные функции и дружественные классы

Дружественные функции класса определяются **вне** области действия этого класса, но имеют **право доступа** к **закрытым** элементам private и protected данного класса. Класс в целом может быть объявлен **другом** (friend) другого класса.

Чтобы объявить функцию как друга (friend) класса, перед ее прототипом в **описании** класса ставится ключевое слово friend (см. рис. 13.3). Чтобы объявить класс Two как друга класса One, необходимо записать объявление в форме

```
friend Two
```

в определении класс One (рис. 13.2).

```
class One {
     friend Two;
     .......
};

class Two {
......
};
```

Рис. 13.2. Отношения дружественности между двумя классами

На отношения дружественности накладываются следующие ограничения:

1. Дружественность требует разрешения, т.е. чтобы класс B стал другом класса A, класс A должен объявить, что класс B – его друг.
2. Дружественность не обладает свойством симметричности, т.е. если класс A друг класса B, то отсюда не следует, что B является другом класса A.

3. Дружественность не является транзитивной, т.е. если класс A друг класса B, а класс B – друг класса C, то отсюда не следует что класс A – друг класса C.
4. Отношения дружественности не наследуются, т.е. друзья базовых классов не являются друзьями производных классов.

```
#include <stdio.h>
class Count
{
    friend void setX(Count &, int);
public:
    Count() {x= 0;}
    void print() const {printf("%d\n", x);}
private:
    int x;};
void setX(Count &c, int val)
{
    c.x = val;
}

void cannotSetX(Count &c, int val)
{
 c.x = val;  // cannot access private member declared in class 'Count'
}

int main()
{
    Count object;
    printf ("object.x after creation: ");
    object.print();
    printf ("object.x after SetX: ");
    setX(object, 10);
    object.print();
    return 0;
}
```

Рис. 13.3. Объявление внешней функции дружественной

14. КОНСТАНТНЫЕ ЭЛЕМЕНТЫ И ЭКЗЕМПЛЯРЫ КЛАССА. СТАТИЧЕСКИЕ ЭЛЕМЕНТЫ КЛАССА

14.1. Константные элементы и экземпляры класса

Константные объекты и константные функции

Объект класса может быть создан в программе с использованием спецификатора `const`. Такой объект будет константным, то есть во время выполнения программы объект изменять нельзя. Работать с таким объектом обычные функции-элементы класса не могут. Для доступа к такому объекту можно ис-

пользовать только функции, объявленные с суффиксом `const` – **константные функции** (рис. 14.1). Такие функции не могут изменять значение объекта – за этим следит компилятор. Вызов из константной функции неконстантной функции запрещен.

В классе можно использовать два варианта одной и той же функции: один вариант для обычных объектов, второй - для константных. Две функции с одинаковым именем и одинаковым списком параметров, отличающиеся только суффиксом const считаются разными функциями.

```
#include <iostream.h>

class My
{
public:
     My(int a=0){my = a;}
     int Get_My() const{return my;}
     int Get_My(){return my;}
     void Set_My(int a) {my =a;}
private:
     int my;
};

int main()
{
     My a(5);
     cout << a.Get_My()<<endl;
     a.Set_My(12);
     cout << a.Get_My()<<endl;//будет вызвана функция без
                              //спецификатора const
     const My b(11);
     cout << b.Get_My()<<endl;//будет вызвана функция с
                              //суффиксом const
     //b.Set_My(12); нельзя использовать неконстантную
                     //функцию с константным объектом
     return 0;
}
```

Рис. 14.1. Работа с константными объектами

Константные элементы класса

Константные элементы класса не могут получать свое значение как обычные элементы-данные с помощью операции присваивания в конструкторе. Также они не могут получать значение в описании класса. Для задания значений константным элементам класса необходимо использовать **список инициализаторов**. В конструкторе после списка параметров нужно поставить двоеточие, далее указать имя константы и в скобках – передаваемое ей значение.

Таким образом, значение константного элемента класса становится известным только в процессе выполнения программы. Это значит, что значение константного элемента класса не может использоваться в качестве указания размера массива, объявленного внутри класса.

Чтобы задать размер массива можно использовать макрос, глобальную переменную-константу или перечисление, объявленное внутри класса. На рис. 14.2 приведена программа, которая использует перечисления для создания массива внутри класса.

```
#include <iostream.h>


class A
{

public:
     A(int d =0, int s = 10):SIZE2(s)
     {
          b = d;
          for (int i = 0; i < SIZE; i++)
          arr1[i] = i+1;
     }
     void print()const
     {
          for (int i = 0; i < SIZE; i++)
          cout<<arr1[i]<<endl;
          cout << endl;
          cout<< b<<endl;
     }

private:
     const int SIZE2;
     int b;
     int arr[SIZE];
     enum t {SIZE=10};
};


int main()
{
     A a(5);
     A *b = &a;
     const A d(3,6);
     b->print();
     d.print();
     return 0;
}
```

Рис. 14.2. Использование перечислений для указания размера массива

Изменение константных объектов

Компилятор не позволяет изменять значения константных объектов. Но иногда бывает необходимо в программе изменить значение константного объекта (например, во время отладки). Такое случается редко, но на этот случай существует обходной путь. Простейший способ основан на обращении к локальному объекту через указатель, который является синонимом для `this` (рис. 14.3).

```
#include <iostream.h>

class My
{
public:
    My(int a=0){my=a;}
    int Get_My() const
    {
         return my;
    }
    My* Set_My(int a) const
    {
         My *This = (My *)this;
         This->my = a;
         return This;
    }
private:
    int my;
};


int main()
{
    const My b(11);
    cout << b.Get_My()<<endl;
    b.Set_My(12);
    cout << b.Set_My(45)->Get_My()<<endl;
    return 0;
}
```

Рис. 14.3. Изменение константных объектов

В программе на рис. 14.3 используется сцепленный вызов:

```
cout << b.Set_My(45)->Get_My()<<endl;
```

Сначала вызывается функция `Set_My(45)` для объекта `b`, которая изменяет его значение и возвращает указатель на этот же объект. Этот указатель передается в операцию выбора члена `->`, и через этот указатель выызвается функция `Get_My()` для того же объекта `b`.

14.2. Статические элементы класса

Обычно каждый объект класса имеет свою собственную копию всех данных-элементов класса. Но в определенных случаях во всех объектах класса должна фигурировать только одна копия некоторых данных элементов для всех объектов класса. Для этих и других целей используются статические элементы-данные, которые содержат информации «для всего класса». Объявление статических элементов начинается с ключевого слова `static`.

```
#include <stdio.h>

class Test{
public:
    Test(int a)
    {
         t=a;
         printf("Constructor Test %d\n", this->t);
         ++count;
    }
    ~Test()
    { printf("Destructor Test %d\n", t);
      --count;}
    //static int get_Count() const {return count;}
    //статическая функция не может быть константной
    //int get_Count(){return count;}
    //функция должна быть статической
    //static int get_Count(){return this->count;}
    //нельзя использовать указатель this
    static int get_Count() {return count;}
private:
    int t; static int count;
};

int Test::count = 0; //инициализация статического элемента

int main()
{
    printf("Before: %d\n", Test::get_Count());
    Test *a = new Test(1);
    Test *b = new Test(2);
    printf("Have objects: %d\n", a->get_Count());
    printf("Have objects: %d\n", b->get_Count());
    delete a; delete b;
    printf("After: %d\n", Test::get_Count());
    return 0;
}
```

Рис. 14.4. Использование статических элементов в классе

```
Output:

Before: 0
Constructor Test 1
Constructor Test 2
Have objects: 2
Have objects: 2
Destructor Test 1
Destructor Test 2
After: 0
```

Рис. 14.5. Результат работы программы на рис. 14.4

Статические переменные похожи на глобальные, но они имеют **областью действия класс**. Статические элементы могут быть открытыми, закрытыми или защищенными. Статическим данным-элементам можно задать значение **один и только один раз** в **области действия файл**. Доступ к статическим элементам класса возможен посредством любого объекта класса или посредством имени класса, с помощью бинарной операции разрешения области действия. Закрытые и защищенные статические элементы класса должны быть доступны открытым функциям-элементам этого класса или друзьям класса. Статические элементы класса существуют даже тогда, когда не существует никаких объектов этого класса. Для обеспечения доступа к закрытому или защищенному элементу класса должна быть предусмотрена открытая статическая функция-элемент (см. рис. 14.4).

Функция-элемент может быть объявлена как `static`, даже если она не должна иметь доступ к нестатическим элементам класса. В отличие от нестатических функций-элементов статическая функция-элемент не имеет указателя `this`, потому что статические данные-элементы и статические функции-элементы существуют независимо от каких-либо объектов класса.

Статические функции нельзя объявлять со спецификатором `const`.

15. ОДИНОЧНОЕ НАСЛЕДОВАНИЕ

Понятие наследования

Часто объекты одного класса на самом деле являются также и объектами другого класса. Прямоугольник является также четырехугольником. Можно сказать, что класс прямоугольник – это наследник класса четырехугольник. Класс четырехугольник является базовым классом, а класс прямоугольник – производным классом. Прямоугольник - это специальный тип четырехугольника, но нельзя утверждать, что четырехугольник является прямоугольником.

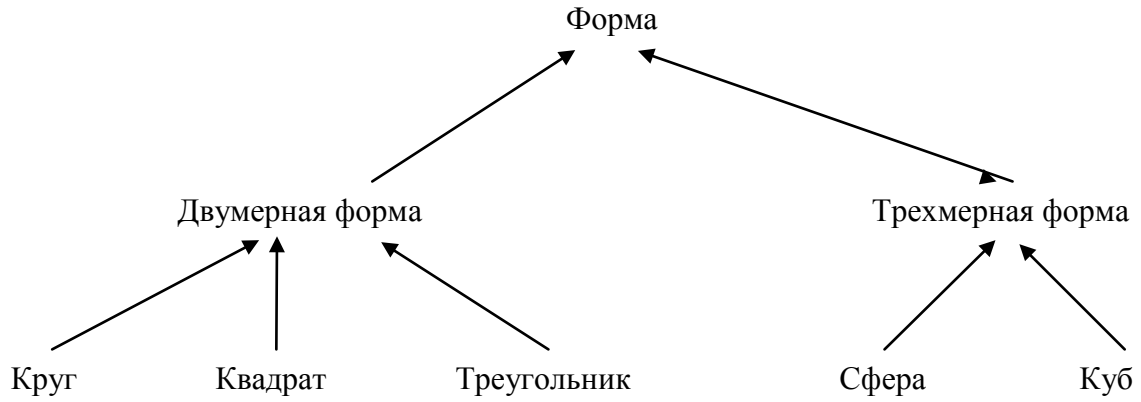


Рис. 15.1. Иерархическая структура, образуемая наследованием

Наследование – это способ повторного использования программного обеспечения, при котором новые классы создаются из уже существующих классво путем заимствования их атрибутов и функций и обогащения этими возможностями новых классов.

Наследование формирует древовидные иерархические структуры. Например так, как показано на рис. 15.1.

Класс, от которого наследуют, называется **суперклассом, базовым** классом, классом-**родителем**. Класс, который наследует, называется **подклассом, производным** классом, классом-**потомком**. Далее будем использовать только термины базовый / производный класс и родитель (предок) / потомок (наследник). В общем случае базовый класс содержит меньше элементов, чем производный класс, но представляет собой более широкий круг объектов. Производный класс содержит большее количество элементов, но является более специфичным, представляет менее широкую группу объектов. Например, базовый класс – автомобиль. Производный класс – автомобиль с откидным верхом.

При построении иерархии наследования используется спецификатор доступа `protected`. В этом разделе помещают элементы-данные и элементы-функции, которые должны быть доступны друзьям и потомкам класса.

Типы наследования

В C++ существует открытое, защищенное и закрытое наследование. Для указания типа наследования используются спецификаторы доступа `public`, `protected` и `private` соответственно. Защищенное и закрытое наследование встречается редко и каждое из них нужно использовать с большой осторожностью. По умолчанию наследование закрытое, поэтому в объявлении производного класса перед именем базового необходимо использовать спецификатор `public` в явном виде.

Тип наследования влияет на доступ к элементам базового класса через объект производного класса (см. табл. 6).

Таблица 6

Влияние типа наследования на доступ к элементам базового класса через объект производного класса

Спецификатор доступа к элементам в базовом классе	Тип наследования		
	public открытое наследование	**protected** защищенное наследование	**private** закрытое наследование
public	public в производном классе	protected в производном классе	private в производном классе
protected	protected в производном классе	protected в производном классе	private в производном классе
private	невидим в производном классе	невидим в производном классе	невидим в производном классе

Базовый класс может быть **прямым** или **косвенным** базовым классом производного класса. Прямой базовый класс явно перечисляется в заголовке при объявлении производного класса. Косвенный базовый класс явно не перечисляется в заголовке производного класса, он наследуется через два или более уровней иерархии классов.

Конструкторы и деструкторы при наследовании

Поскольку производный класс наследует элементы базового класса, то при создании объекта производного класса должен быть вызван конструктор базового класса для задания начальных значений элементам базового класса, содержащимся в объекте производного класса. В конструкторе производного класса при явном вызове конструктора базового класса может быть предусмотрен список инициализаторов элементов, в противном случае конструктор производного класса будет неявно вызывать конструктор базового класса с умолчанием.

Конструкторы, деструкторы, отношения дружественности и операции присваивания не наследуются производным классом. Однако конструкторы и операции присваивания производного класса могут вызывать конструкторы и операции присваивания базового класса..

Если конструктор производного класса отсутствует, то конструктор по умолчанию производного класса вызывает конструктор базового класса.

Деструкторы вызываются в последовательности, обратной вызовам конструкторов, то есть деструктор производного класса вызывается раньше деструктора базового класса.

Неявное преобразование объектов производных классов в объекты базовых классов

Объекты производных классов «являются» также и объектами базового класса, типы объектов производного и базового классов различны. Объекты производных классов можно рассматривать как объекты базового класса. Это

имеет смысл, потому что производный класс имеет элементы, соответствующие каждому элементу базового класса.

Но производный класс обычно имеет большее количество элементов, чем его базовый класс. Поэтому присваивание объекта базового класса объекту производного класса оставило бы неопределенными добавочные элементы производного класса. Такое присваивание не разрешено.

Указатель на объект производного класса может быть неявно преобразован в указатель на объект базового класса.

Существуют четыре возможных способа комбинирования и попарного сопоставления указателей и объектов базового и производного классов:

1. Ссылка на объект базового класса с помощью указателя базового класса.
2. Ссылка на объект производного класса с помощью указателя производного класса.
3. Ссылка на объект производного класса с помощью указателя базового класса. Если происходит ссылка с помощью указателя базового класса на элементы, имеющиеся только в объектах производного класса, произойдет синтаксическая ошибка.
4. Ссылка на объект базового класса с помощью указателя производного класса является синтаксической ошибкой.

Учебный пример: точка, круг, цилиндр

Рассмотрим иерархию точка, круг, цилиндр (рис. 15.2 –15.6).

```
class Point {
public:
    Point(float a =0, float b =0) {SetPoint(a,b);}
    void SetPoint (float a, float b) {x = a; y = b;}
    float getX() {return x;}
    float getY() {return y;}
    void print() {printf("[%.2f, %.2f]\n",x, y);}
protected:   float x, y;};
```

Рис. 15.2. Описание базового класса `Point`

```
class Circle : public Point {
public:
    Circle(float r = 0, float x = 0, float y = 0) :
          Point(x,y) {radius = r;}
    void setRadius(float r) {radius = r;}
    float getRadius() {return radius;}
    float area() {return 3.14159*radius*radius;}
    void print() {printf("Center = [%.2f, %.2f]; Radius =
          %.2f\n",x, y, radius);}
protected:
    float radius;};
```

Рис. 15.3. Описание производного класса `Circle`

```
class Cylinder : public Circle
{
public:
    Cylinder(float h=0, float r= 0, float x= 0, float y= 0):
          Circle(r,x,y) {height = h;}
    void setHeight(float h){height = h;}
    float getHeight() {return height;}
    float area()
    {return 2* Circle::area() + 2*3.14159*radius*height;}
    float volume() {return Circle::area()*height;}
    void print() {printf("Center = [%.2f, %.2f]; Radius =
          %.2f; Height = %.2f\n", x, y, radius, height);}
protected:
    float height;
};
```

Рис. 15.4. Описание производного класса `Cylinder`

```
int main()
{
    Cylinder cyl(5.7, 2.5, 1.2, 2.3);
    cyl.setHeight(10);
    cyl.setRadius(5);
    cyl.SetPoint(2,2);
    printf("New:\n");
    cyl.print();
    Point &pRef = cyl;
    printf("As Point:\n");
    pRef.print();
    Point p(1,1);
    Point *a = &cyl; //  такое вполне законно
    //Cylinder &aRef = p; cannot convert from 'class Point' to 'class
    //Cylinder &'
    return 0;
}
```

Рис. 15.5. Использование иерархии классов: `Point – Circle - Cylinder`

```
Output:
Constructor Point
Constructor Circle
Constructor Cylinder
New:
Center = [2.00, 2.00]; Radius = 5.00; Height = 10.00
As Point:
[2.00, 2.00]
Destructor Cylinder
Destructor Circle
Destructor Point
```

Рис. 15.6. Результат работы программы

16. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

16.1. Понятие множественного наследования

Класс может порождаться более чем от одного базового класса. Такое порождение называется **множественным наследованием.** Множественное наследование означает, что производный класс наследует элементы нескольких базовых классов (рис. 16.1).

```
class Base1{
public:
    Base1(int x) {value = x;}
    int getData() {return value;}
protected:
    int value;};

class Base2{
public:
    Base2(char c) {letter = c;}
    char getData() {return letter;}
protected:
    char letter;};

class Child : public Base1, public Base2{
public:
    Child (int i, char c, float f) : Base1(i), Base2(c)
        {real = f;}
    float getReal() {return real;}
    void print() {printf("Integer: %d\nSymbol: %c\nReal:
        %.2f\n", value, letter, real);}
private:
    float real;};

int main(){
    Base1 b1(10), *base1Ptr;
    Base2 b2('S'), *base2Ptr;
    Child ch(7, 'D', 3.5);
    printf("Object b1 contains integer %d\n", b1.getData());
    printf("Object b2 contains integer %c\n", b2.getData());
    printf("Object ch contains:\n");    ch.print();
    printf("Elements ch:\n");
    printf("Integer:  %d\nSymbol:   %c\nReal:     %.2f\n",
    ch.Base1::getData(), ch.Base2::getData(), ch.getReal());
    base1Ptr = &ch;
    printf("Result base1Ptr->getData() %d\n",
        base1Ptr->getData());
    base2Ptr = &ch;
    printf("Result base2Ptr->getData() %c\n",
        base2Ptr->getData());
    return 0;}
```

Рис. 16.1. Пример множественного наследования

```
Output
Object b1 contains integer 10
Object b2 contains integer S
Object ch contains:
Integer:  7
Symbol:   D
Real:     3.50
Elements ch:
Integer:  7
Symbol:   D
Real:     3.50
Result base1Ptr->getData() 7
Result base2Ptr->getData() D
```

Рис. 16.2. Результат работы программы

Множественное наследование указывается двоеточием после имени класса с последующим перечислением через запятую базовых классов. Конструктор производного класса вызывает конструкторы каждого из своих базовых классов с использованием списка инициализаторов переменных. Конструкторы базовых классов вызываются в той последовательности, в которой определено наследованием.

16.2. Виртуальные базовые классы

Одна из проблем множественного наследования связана с многократным вхождением одного класса в число предков другого класса. Она решается объявлением некоторых базовых классов **виртуальными.** Это означает, что хотя класс A несколько раз входит в ориентированный ациклический граф наследования, его данные войдут в D только однажды (см. рис. 16.3).

По умолчанию производный класс содержит столько экземпляров данных базового класса, сколько раз этот базовый класс выступает в роли предка по разным линиям наследования. Объявление класса производным виртуально означает, что он готов ограничиться только одним экземпляром данных базового класса, если этот базовый класс многократно является его виртуальным предком. Если все пути от базового класса к производному в графе наследования проходят через виртуальное наследование, то производный класс будет содержать только одну копию данных. Если будет использоваться комбинация виртуального и невиртуального вариантов наследования, то для каждого невиртуального наследования будет создана отдельная копия данных, плюс одна общая копия для всех виртуальных вариантов наследования.

Учтите, что классы, имеющие общий виртуальный базовый класс и входящие в один ориентированный ациклический граф множественного наследования, могут нарушить инкапсуляцию друг друга. И это вполне естественно, поскольку эти классы выразили свою готовность совместно использовать данные базового класса посредством виртуального наследования. Свертка данных базовых классов полностью находится под контролем программиста — в отли-

чие от других языков программирования, она не задействуется по умолчанию и не является обязательной.

```
class A{
    int a;
public:
    A(int aa=8)
    {a=aa;cout<<"Constructor A:"<<'\t'<<a<<endl;}};

class B:virtual public A{
    int b;
public:
    B(int bb=6, int aa=18):A(aa)
    {b=bb;cout<<"Constructor B:"<<'\t'<<b<<endl;}};

class C:virtual public A{
    int c;
public:
    C(int cc=3, int aa=14):A(aa)
    {c=cc;cout<<"Constructor C:"<<'\t'<<c<<endl;}};

class D:public C, public B{
    int d;
public:
    D(int f=4, int b=16, int c=12,int a=67):B(b, a), C(c,a)
    {d=f;cout<<"Constructor D:"<<'\t'<<d<<endl;}};

int main()
{
    D asd;
    return 0;
}
```

Рис. 16.3. «Ромбовидное» наследование

16.3. Порядок инициализации различных частей создаваемого класса в C++

Порядок инициализации определяеся рекурсивным применением следующеого набора правил:

1. Конструктор последнего производного класса вызывает конструкторы подобъекто виртуальных базовых классов. Инициализация виртуальных базовых классов выполняется в глубину, в порядке слева направо;
2. Конструируются подобъекты непосредственных базовых классов в порядке их объявления в определении класса;
3. Конструируются нестатические подобъекты-члены в порядке их объявления в определении класса;
4. Выполняется тело конструктора.

17. ПЕРЕГРУЗКА ОПЕРАЦИЙ

Общие правила перегрузки операций

Программист может использовать встроенные типы, а может определять и новые типы. Встроенные типы можно использовать с определенным набором операций C++. Программист также может использовать операции с типами, определенными пользователем. C++ не позволяет создавать новые операции, но он позволяет перегружать уже существующие операции так, что при использовании этих операций с объектами классов они приобретают смысл, соответствующий новым типам.

Операции перегружаются путем составления описания функции, как мы обычно это делали, только имя функции должно состоять из ключевого слова `operator`, после которого записывается перегружаемая операция (см. рис. 17.1.).

Чтобы использовать операцию над объектами класса, эта операция должна быть перегружена, кроме двух исключений. Операции **присваивания** (=) и **адресации** (&) могут быть использованы с каждым классом без явной перегрузки, но их можно также перегружать.

Перегружать можно любые операции, кроме:

```
.   .*   ::   ?:   sizeof
```

На перегрузку операций накладываются следующие ограничения:

1. Старшинство операций не может быть изменено перегрузкой;
2. Ассоциативность операций не может быть изменена перегрузкой;
3. Изменить количество операндов, которое берет операция, невозможно (бинарная операция останется бинарной, унарная - унарной);
4. Создавать новые операции невозможно;
5. Невозможно с помощью перегрузки операций изменить смысл работы операции с объектом встроенного типа. Программист не может изменить смысл того, как складываются два целых числа.

Функции операции могут быть, а могут не быть функциями-элементами. Если функции не являются элементами, они являются друзьями. **При перегрузке операций `()`, `[]`, `->` или `=` функция перегрузки операции должна быть объявлена как элемент класса**.

Когда функция-операция реализована как функция элемент, крайний левый (или единственный) операнд должен быть объектом того класса (или ссылкой на объект того класса), элементом которого является функция. Если левый операнд должен быть объектом другого класса или встроенного типа, такая функция-операция не может быть реализована как функция-элемент, она должна быть реализована как друг, если она должна иметь прямой доступ к закрытым или защищенным элементам класса.

Далее приведен код программы:

```
#include <iostream.h>

class Arr
{
    friend ostream & operator << (ostream &, const Arr &);
    friend istream & operator >> (istream &, Arr &);
public:
    Arr()
     {for (int i=0; i<SIZE; i++)
         arr[i]=i+1;
      cout<<"Constructor Arr"<<endl;}
    Arr operator++()
    {for (int i=0; i<SIZE; i++)arr[i]++;
     return *this;}
    Arr operator++(int)
    {Arr temp = *this;
    for (int i=0; i<SIZE; i++)arr[i]++;
    return temp;}
    Arr& operator+=(const Arr & a)
    {for (int i=0; i<SIZE; i++)
         arr[i]+=a.arr[i];
      return *this;}
    Arr operator+(const Arr & a)
    {Arr temp = *this;
     for (int i=0; i<SIZE; i++)
         temp.arr[i]+=a.arr[i];
     return temp;}
    Arr operator+(const int a)
    {Arr temp = *this;
    for (int i=0; i<SIZE; i++)
         temp.arr[i]+=a;
    return temp;}
    Arr operator~()
    {for (int i=0; i<SIZE; i++)
         arr[i]*=(-1);
     return *this;}
protected:
    enum {SIZE=10};
    int arr[SIZE];};

ostream & operator << (ostream & s, const Arr & a){
    for(int i=0; i<10; i++)
         s<<a.arr[i]<<endl;
    return s;}

istream & operator >> (istream & is, Arr & a){
    for(int i=0; i<10; i++)
         is>>a.arr[i];
    return is;}
```

Рис. 17.1. Определение класса массива с перегруженными операциями

```
int main(){
    Arr b,c;
    int d =100;
    cout<< c++;
    cout <<++c;
    cout<<c+b<<endl;
    cout <<c<<b<<endl;
    c+=b;
    cout <<c<<b<<endl;
    cout<< c+d<<endl;
    cout <<c<<endl;
    cin>> c;
    cout <<c<<endl;
    cout <<(~c)<<endl;
    return 0;}
```

Рис. 17.2. – Использование класса Arr из программы на рис 17.1

Перегрузка операций инкремента и декремента

Операции инкремента и декремента в префиксной и постфиксной форме могут быть перегружены.

```
class A
{
    int a;
public:
    A(int b = 0){a=b;}
    int getA(){return a;}
    //префиксная форма
    A operator++()
    {++a;
    return *this;}
    //постфиксная форма
    A operator++(int)
    {A temp = *this;
    ++a;
    return temp;}
};

int main(){
    A a;
    cout << a.getA()<<endl;
    cout << (++a).getA()<<endl;
    cout << (a++).getA()<<endl;
    cout << a.getA()<<endl;
    return 0;}
```

Рис. 17.3. Перегрузка операции инкремента

Для того чтобы перегрузить операцию инкремента для получения возможности использования и префиксной и постфиксной форм, каждая из этих двух перегруженных функций-операций должна иметь разную сигнатуру. Тогда компилятор сможет определить, какая версия операции ++ имеется в виду в каждом конкретном случае.

Префиксный оператор перегружается, как и любой другой префиксный унарный оператор.

При перегрузке инкремента в постфиксной форме операция реализуется как функция, имеющая чисто формальный параметр `int`. Его вводят для того, чтобы сделать список аргументов функции `operator++`, используемой в постфиксной форме, отличным от списка аргументов функции `operator++`, используемых для префиксной формы инкремента (см. рис. 17.3.).

Все это применимо и к перегрузке префиксной и постфиксной форм операции декремента.

18. ВИРТУАЛЬНЫЕ ФУНКЦИИ И ПОЛИМОРФИЗМ

18.1. Виртуальные функции

Пусть ряд классов форм, таких как круг (`Circle`), треугольник (`Triangle`), прямоугольник (`Rectangle`), квадрат (`Square`) и т.д. являются производными от базового класса форма (`Shape`). В объектно-ориентированном программировании каждый из этих классов может быть наделен способностью рисовать свою форму. Хотя каждый класс имеет свою функцию рисования `draw`, для разных форм эти функции совершенно различны. При рисовании любой формы, какая бы она ни была, было бы прекрасно иметь возможность работать со всеми этими функциями в целом как с объектами базового класса `Shape`. Тогда для рисования любой формы можно было бы просто вызвать функцию `draw` базового класса `Shape` и предоставить программе динамически (т.е. во время выполнения программы) определять, какую из функций `draw` производного класса следует использовать.

Для того, чтобы предоставить такого рода возможность, объявим функцию `draw` **виртуальной** функцией и затем **переопределим** функцию `draw` в каждом производном классе, чтобы она рисовала соответствующую форму. Функция объявляется виртуальной с помощью ключевого слова `virtual`, предшествующего прототипу функции в базовом классе. Например:

```
virtual void draw() const;
```

Этот прототип объявляет, что функция `draw` является константной функцией, которая не принимает никаких аргументов, ничего не возвращает и является виртуальной функцией.

Если функция `draw` в базовом классе объявлена как `virtual` и если мы затем вызываем функцию `draw` через указатель базового класса, указывающий на объект производного класса (`shapePtr->draw()`), то программа будет

динамически (т.е. во время выполнения программы) выбирать соответствующую функцию `draw` производного класса. Это называется **динамическим связыванием или поздним связыванием.**

Когда виртуальная функция вызывается путем обращения к заданному объекту по имени и при этом используется операция доступа к элементу точка (`squareObject.draw()`), тогда эта ссылка разрешается (обрабатывается) **во время компиляции** (это называется **статическим связыванием**) и в качестве вызываемой определяется функция класса данного объекта (или наследуемая этим классом).

Абстрактные базовые классы и конкретные классы

В программировании бывают задачи, в которых полезно определять классы, объекты которых программист создавать не намерен. Такие классы называются **абстрактными**. Они обычно бывают базовыми классами в иерархии наследования, поэтому их могут называть **абстрактными базовыми классами. Объекты абстрактного класса не могут быть реализованы**. Но можно создать указатель или ссылку на абстрактный базовый класс.

Классы, объекты которых могут быть реализованы, называются **конкретными классами.**

Абстрактные базовые классы являются слишком общими для определения реальных объектов, требуется больше определенности, чтобы можно было думать о реализации объектов. Для этого предназначены конкретные классы; они обладают необходимой спецификой, делающей реальным создание объектов.

Класс делается абстрактным путем объявления одной или более его виртуальных функций чисто виртуальными. **Чистой виртуальной функцией** является такая функция, у которой в ее объявлении тело определено как 0 (инициализатор равен 0), например:

```
virtual float earning( )const = 0;
```

Иерархия не требует обязательного включения каких-либо абстрактных классов. Но многие программы имеют иерархию классов, порожденную абстрактным базовым классом.

18.2. Полиморфизм

Полиморфизм – это возможность для объектов разных классов, связанных с помощью наследования, реагировать различным образом при обращении к одной и той же функции-элементу.

Полиморфизм реализуется посредством механизма наследования и виртуальных функций. Если при использовании виртуальной функции запрос осуществляется с помощью указателя базового класса (или ссылки), то C++ выбирает правильную переопределенную функцию в соответствующем производном классе, связанном с данным объектом.

Иногда функция-элемент определена в базовом классе не как виртуальная, но переопределена в производном классе. Если такая функция-элемент вызывается через указатель базового класса, то используется версия базового класса. Если же эта функция-элемент вызывается через указатель производного класса, то используется версия производного класса. Это не полиморфное поведение. Для объекта производного класса вариант функции базового класса должен быть вызван явным образом (рис. 18.1.)

```
Cylinder *pCyl = &cyl;
pCyl->Circle::print();
```

Рис. 18.1. Вызов виртуальной функции базового класса через указатель производного класса

Новые классы и динамическое связывание

Полиморфизм и виртуальные функции могут прекрасно работать, если все возможные классы известны заранее. Но они также прекрасно работают, когда в систему добавляются новые типы классов.

Новые классы встраиваются при помощи динамического связывания. Во время компиляции нет необходимости знать тип объекта, чтобы скомпилировать вызов виртуальной функции.

Во время выполнения программы вызов виртуальной функции-элемента будет направлен варианту виртуальной функции соответствующего класса. Для этого существует **таблица виртуальных методов**, которая реализуется в виде массива, содержащего указатели на функции. У каждого класса, который содержит виртуальные функции, имеется таблица виртуальных методов. Для каждой виртуальной функции в классе таблица имеет элемент, содержащий указатель на вариант виртуальной функции, используемый в объектах данного класса.

Виртуальная функция, используемая в некотором классе, может быть определена в этом классе или прямо или косвенно наследоваться из базового класса, стоящего выше в иерархии.

Если базовый класс имеет виртуальную функцию-элемент, то производные классы могут переопределить эту функцию, но могут этого и не делать. Тогда производный класс будет использовать вариант виртуальной функции-элемента базового класса и это будет отражено в таблице виртуальных методов.

Каждый объект класса, содержащего виртуальные функции, имеет указатель на таблицу виртуальных методов этого класса, недоступный для программиста. Во время выполнения программы полиморфные вызовы виртуальных функций осуществляются через разыменование указателя объекта на таблицу виртуальных методов, что дает доступ к таблице виртуальных методов класса. Затем в таблице виртуальных методов находится соответствующий указатель на функцию, он разыменовывается, что и завершает вызов виртуальной функции во время выполнения программы.

Виртуальные деструкторы

Если у класса имеются виртуальные функции, то классу необходимо создать **виртуальный деструктор**. Это автоматически приведет к тому, что все деструкторы производных классов станут виртуальными. В этом случае, если объект в иерархии уничтожен явным использованием операции `delete`, примененной к указателю базового класса на объект производного класса, то будет вызван деструктор соответствующего класса.

Если деструктор базового класса, содержащего виртуальные функции, не объявить виртуальным, то если объект в иерархии уничтожен явным использованием операции `delete`, примененной к указателю базового класса на объект производного класса, то будет вызван деструктор производного класса, что может привести к краху всей программы.

Например, если бы в программе на рис. 18.2 не был бы определен виртуальный деструктор, то при уничтожении объекта типа B через указатель типа A могли бы возникнуть проблемы с динамической памятью.

```
#include <iostream.h>

class A
{
public:
     A(){cout<<"Constructor A"<<endl;}
     virtual ~A(){cout<<"Destructor A"<<endl;}
};


class B:public A
{
public:
     B(int a=10){b=a;cout<<"Constructor B"<<endl;}
     virtual ~B(){cout<<"Destructor B"<<endl;}
private:
     int b;
};


int main()
{
     A *a[2];
     a[0] = new A(100);
     a[1] = new A();
     delete a[1];
     delete a[0];
     return 0;
}
```

Рис. 18.2. Использование вирутального деструктора

18.3. Учебный пример: точка, круг, цилиндр

Вспомним учебный пример из п. 15 и попробуем изменить его. В имеющуюся иерархию классов добавим абстрактный базовый класс Shape и определим все функции классов как виртуальные (см. рис. 18.3 – 18.7).

```
class Shape
{
public:
    Shape()
    {
         printf("Constructor Shape\n");
    }
    virtual ~Shape() {printf("Destructor Shape\n\n");}
    virtual float area() const {return 0.0;}
    virtual float volume() const {return 0.0;}
    virtual void printShapeName() const = 0;
    virtual void print() const =0;
};
```

Рис. 18.3. Определение абстрактного базового класса Shape

```
class Point: public Shape
{
    friend ostream &operator<< (ostream &, const Point &);
public:
    Point(float a =0, float b =0)
         {
         SetPoint(a,b);
         cout<<"Constructor Point\n";
         }
    virtual ~Point() {cout<<"Destructor Point\n";}
    void SetPoint (float a, float b) {x = a; y = b;}
    float getX() const {return x;}
    float getY() const {return y;}
    virtual void printShapeName() const
         {cout<<"Point: \n";}
    virtual void print() const
         {cout<<"["<<x<<", "<< y<<"]";}
protected:
    float x, y;
};

ostream &operator<< (ostream &output, const Point &p)
{
    p.print();
    return output;
}
```

Рис. 18.4. Определение производного класса Point

```
class Circle : public Point
{
    friend ostream &operator<< (ostream &, const Circle &);
public:
    Circle(float r = 0, float x = 0, float y = 0) :
         Point(x,y)
   {radius = r;cout<<"Constructor Circle\n";}
    virtual ~Circle(){cout<<"Destructor Circle\n";}
    void setRadius(float r) {radius = r;}
    float getRadius() {return radius;}
    virtual float area() const
    {return 3.14159*radius*radius;}
    virtual void printShapeName() const
    {cout<<"Circle: \n";}
    virtual void print() const
     {cout<<"Center = ["<<getX()<<", "<<getY()<<
         "]; Radius = "<< radius <<endl;}
protected:
    float radius;};
ostream &operator<< (ostream &output, const Circle &c){
    c.print();
    return output;}
```

Рис. 18.5 Определение производного класса `Circle`

```
class Cylinder : public Circle
{
    friend ostream &operator<< (ostream &, const Cylinder &);
public:
Cylinder(float h=0, float r= 0, float x= 0, float y= 0) :
          Circle(r,x,y)
{height = h;cout<<"Constructor Cylinder\n";}
    ~Cylinder(){cout<<"Destructor Cylinder\n";}
    void setHeight(float h){height = h;}
    float getHeight() {return height;}
virtual float area() const
    {return 2* Circle::area() + 2*3.14159*radius*height;}
    virtual float volume() const
    {return Circle::area()*height;}
    virtual void printShapeName() const
    {cout<<"Cylinder: \n";}
    virtual void print() const
{Circle::print(); cout<<"Height = "<< height<<endl;);}
protected:
    float height;};

ostream &operator<< (ostream &output, const Cylinder &c){
    c.print();
    return output;}
```

Рис. 18.6 Определение производного класса `Cylinder`

```
int main()
{
//**********************************
    Point point(7, 11);
    point.printShapeName();
    cout << point << endl;
//**********************************
    Circle circle(3.5, 22, 8);
    circle.printShapeName();
    cout << circle << endl;
//**********************************
    Cylinder cylinder(10, 3.3, 10, 10);
    cylinder.printShapeName();
    cout << cylinder << endl;
//**********************************
    Shape *arrayOfShape[3];
    arrayOfShape[0] = &point;
    arrayOfShape[1] = &circle;
    arrayOfShape[2] = &cylinder;
    for (int i = 0; i< 3; i++)
    {
       arrayOfShape[i]->printShapeName();
       arrayOfShape[i]->print();
       cout<<"Area is "<< arrayOfShape[i]->area()<<endl;
       cout<<"Volume is "<<arrayOfShape[i]->volume()<<endl;
    }
//**********************************
    return 0;
}
```

Рис. 18.7 Использование иерархии классов:
`Shape - Point – Circle – Cylinder`

19. ШАБЛОНЫ

19.1. Шаблоны функций

Перегруженные функции обычно используются для выполнения похожих операций над различными типами данных. Если же для каждого типа данных должны выполняться идентичные операции, то удобнее использовать **шаблоны функций**. При этом программист должен написать всего одно описание шаблона функции (см. рис. 19.1.).

Все описания шаблонов функций начинаются с ключевого слова `template`, за которым следует список формальных параметров шаблона, заключенный в угловые скобки, каждому формальному параметру должно предшествовать ключевое слово `class`:

`template<class T, class B>`

Формальные параметры в описании шаблона используются (наряду с другими параметрами) для определения типов параметров функции, типа возвращаемого функцией значения и типов переменных, объявляемых внутри функции. Ключевое слово `class` фактически означает «Любой встроенный или определенный пользователем тип данных».

```
#include <iostream.h>

template<class T>
void printArray(T *array, const int count)
{
   for (int i = 0; i< count; i++)
        cout << array[i] << '\t';
   cout << endl;
}

int main()
{
   const int aCount = 5, bCount = 7, cCount = 6;
   int a[aCount] = {1,2,3,4,5};
   float b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
   char c[cCount] = "HELLO";
   cout << "Array of integer: \n";
   printArray(a, aCount);
   cout << "Array of float: \n";
   printArray(b, bCount);
   cout << "Array of char: \n";
   printArray(c, cCount);
   return 0;
}
```

Рис. 19.1. Использование шаблонов функций

```
Output:
Array of integer:
1   2    3    4    5
Array of float:
1.1 2.2  3.3  4.4  5.5  6.6  7.7
Array of char:
H   E    L    L    O
```

Рис. 19.2. Результат работы программы

В данной программе `T` называется **параметром типа**. Когда компилятор обнаруживает в тексте программы вызов функции `printArray`, он заменяет `T` во всей области определения шаблона на тип первого параметра функции `printArray` и C++ создает **шаблонную функцию** вывода массива указанного типа данных. После этого вновь созданная функция компилируется.

Например, так, как показано на рис. 19.3, если массив целого типа.

```
void printArray(int *array, const int count)
{
     for (int i = 0; i< count; i++)
          cout << array[i] << '\t';
     cout << endl;
}
```

Рис. 19.3. Создание шаблонной функции для типа `int`

Каждый формальный параметр из описания шаблона функции должен появиться в списке параметров функции по крайней мере один раз. Имя формального параметра может использоваться в списке параметров заголовка шаблона только один раз. Одно и то же имя формального параметра шаблона функции может использоваться несколькими шаблонами.

Шаблонные функции и перегрузка функций тесно связаны друг с другом. Все родственные функции, полученные из шаблона, имеют одно и то же имя; поэтому компилятор использует механизм перегрузки для того, чтобы обеспечить вызов соответствующей функции.

Если шаблон вызывается с определяемым пользователем типом в качестве параметра и если этот шаблон использует операции (например, ==, + ,<= и др.)с объектами этого типа, то эти операции должны быть перегружены.

19.2. Шаблоны классов

Зачем программисту может понадобиться определить такой тип, как вектор целых чисел? Как правило, ему нужен вектор из элементов, тип которых неизвестен создателю класса `Vector`. Следовательно, надо суметь определить тип вектора так, чтобы тип элементов в этом определении участвовал как параметр, обозначающий «реальные» типы элементов (рис. 19.4.).

```
template < class T >
class Vector { // вектор элементов типа T
    T * v;
    nt sz;
public:
    Vector ( int s )
    {
    if ( s <= 0 )
         printf ( "недопустимый для Vector размер" );
    v = new T [sz = s]; // выделить память для массива s типа T
    }
    T & operator [] ( int i );
    int size () { return sz; }
    // …
};
```

Рис. 19.4. Описание шаблона `Vector`

Это определение шаблона типа. Он задает способ получения семейства сходных классов. В нашем примере шаблон типа `Vector` показывает, как можно получить класс вектор для заданного типа его элементов. Это описание отличается от обычного описания класса наличием начальной конструкции `template<class T>`, которая и показывает, что описывается не класс, а шаблон типа с заданным параметром-типом (здесь он используется как тип элементов). Теперь можно определять и использовать вектора разных типов, как показано на рис. 19.5.

```
main ()
{
   Vector <int> v1 (100); // вектор из 100 целых
   Vector <double> v2 (200); //вектор из 200 чисел двойной точности
// ...
}
```

Рис. 19.5. Использование шаблонного класса `Vector`

Шаблоны классов часто называют **параметризованными типами**, так как они имеют один или большее количество параметров типа, определяющих настройку родового шаблона класса на специфический тип данных при создании объекта класса.

Шаблоны классов не могут вкладываться друг в друга.

Шаблоны фактически являются макросами, поэтому они должны полностью определяться в заголовочных файлах.

20. ОБРАБОТКА ОШИБОК

Понятие особой ситуации или исключения

Создатель библиотеки способен обнаружить динамические ошибки, но не представляет, какой в общем случае должна быть реакция на них. Пользователь библиотеки способен написать реакцию на такие ошибки, но не в силах их обнаружить. Если бы он мог, то сам разобрался бы с ошибками в своей программе, и их не пришлось бы выявлять в библиотечных функциях. Для решения этой проблемы в язык введено понятие **особой ситуации** или **исключения**

Суть этого понятия в том, что функция, которая обнаружила ошибку и не может справиться с нею, запускает особую ситуацию, рассчитывая, что устранить проблему можно в той функции, которая прямо или опосредованно вызывала первую. Если функция рассчитана на обработку ошибок некоторого вида, она может указать это явно, как готовность перехватить данную особую ситуацию.

Рассмотрим в качестве примера как для класса `Vector` можно представлять и обрабатывать особые ситуации, вызванные выходом за границу массива. Предполагается, что объекты класса `Range` будут использоваться как особые

ситуации. Если в функции предусмотрена реакция на ошибку недопустимого значения индекса, то ту часть функции, в которой эти ошибки будут перехватываться, надо поместить в оператор `try`. В нем должен быть и обработчик особой ситуации.

Обработчиком особой ситуации называется конструкция, показанная на рис. 20.1.

```
catch ( /* ... */ )
{
    // ...
}
```

Рис. 20.1. Блок обработки исключения

Ее можно использовать только сразу после блока, начинающегося служебным словом `try`, или сразу после другого обработчика особой ситуации. Служебным является и слово `catch`. После него идет в скобках описание, которое используется аналогично описанию формальных параметров функции, а именно, в нем задается тип объектов, на которые рассчитан обработчик, и, возможно, имена параметров.

Процесс запуска и перехвата особой ситуации предполагает просмотр цепочки вызовов от точки запуска особой ситуации до функции, в которой она перехватывается. При этом восстанавливается состояние стека, соответствующее функции, перехватившей ошибку, и при проходе по всей цепочке вызовов для локальных объектов функций из этой цепочки вызываются деструкторы.

Если при просмотре всей цепочки вызовов, начиная с запустившей особую ситуацию функции, не обнаружится подходящий обработчик, то программа завершается.

Если обработчик перехватил особую ситуацию, то она будет обрабатываться и другие, рассчитанные на эту ситуацию, обработчики не будут рассматриваться. Иными словами, активирован будет только тот обработчик, который находится в самой последней вызывавшейся функции, содержащей соответствующие обработчики.

Особые ситуации и традиционная обработка ошибок

Наш способ обработки ошибок по многим параметрам выгодно отличается от более традиционных способов. Перечислим, что может сделать операция индексации `Vector::operator[]()` (см. рис. 20.2. – 20.5.) при обнаружении недопустимого значения индекса:

1) завершить программу;
2) возвратить значение, трактуемое как "ошибка";
3) возвратить нормальное значение и оставить программу в неопределенном состоянии;
4) вызвать функцию, заданную для реакции на такую ошибку.

Вариант первый («завершить программу») реализуется по умолчанию в том случае, когда особая ситуация не была перехвачена. Для большинства ошибок можно и нужно обеспечить лучшую реакцию.

Вариант второй («возвратить значение «ошибка») можно реализовать не всегда, поскольку не всегда удается определить значение «ошибка». Так, в нашем примере любое целое является допустимым значением для результата операции индексации. Если можно выделить такое особое значение, то часто этот вариант все равно оказывается неудобным, поскольку проверять на это значение приходится при каждом вызове. Так можно легко удвоить размер программы. Поэтому для обнаружения всех ошибок этот вариант редко используется последовательно.

Вариант третий («оставить программу в неопределенном состоянии») имеет тот недостаток, что вызывавшая функция может не заметить ненормального состояния программы.

Обработка особых ситуаций не предназначалась для тех случаев, на которые рассчитан *вариант четвертый* («вызвать функцию реакции на ошибку»). Отметим, однако, что если особые ситуации не предусмотрены, то вместо функции реакции на ошибку можно как раз использовать только один из трех перечисленных вариантов.

Механизм особых ситуаций успешно заменяет традиционные способы обработки ошибок в тех случаях, когда последние являются неполным, некрасивым или чреватым ошибками решением. Этот механизм позволяет явно отделить часть программы, в которой обрабатываются ошибки, от остальной ее части, тем самым программа становится более понятной и с ней проще работать различным сервисным программам. Свойственный этому механизму регулярный способ обработки ошибок упрощает взаимодействие между раздельно написанными частями программы.

В этом способе обработки ошибок есть для программирующих на С новый момент: стандартная реакция на ошибку (особенно на ошибку в библиотечной функции) состоит в завершении программы. Традиционной была реакция продолжать программу в надежде, что она как-то завершится сама. Поэтому способ, базирующийся на особых ситуациях, делает программу более «хрупкой» в том смысле, что требуется больше усилий и внимания для ее нормального выполнения. Но это все-таки лучше, чем получать неверные результаты на более поздней стадии развития программы (или получать их еще позже, когда программу сочтут завершенной и передадут ничего не подозревающему пользователю). Если завершение программы является неприемлемой реакцией, можно смоделировать традиционную реакцию с помощью перехвата всех особых ситуаций или всех особых ситуаций, принадлежащих специальному классу

Механизм особых ситуаций можно рассматривать как динамический аналог механизма контроля типов и проверки неоднозначности на стадии трансляции. При таком подходе более важной становится стадия проектирования программы, и требуется большая поддержка процесса выполнения программы, чем для программ на С. Однако в результате получится более предсказуемая про-

грамма, ее будет проще встроить в программную систему, она будет понятнее другим программистам и с ней проще будет работать различным сервисным программам.

Таким образом, можно сделать вывод, что механизм особых ситуаций поддерживает, подобно другим средствам C++, «хороший» стиль программирования, который в таких языках, как C, применяется только не в полном объеме и на неформальном уровне.

Другие точки зрения на особые ситуации

Особая ситуация - одно из тех понятий, которые имеют разный смысл для разных людей. В C++ механизм особых ситуаций предназначен для обработки ошибок. В частности, он предназначен для обработки ошибок в программах, состоящих из независимо создаваемых компонентов (см. рис. 20.2 – 20.5).

Этот механизм рассчитан на особые ситуации, возникающие только при последовательном выполнении программы (например, контроль границ массива). Асинхронные особые ситуации такие, например, как прерывания от клавиатуры, нельзя непосредственно обрабатывать с помощью этого механизма. В различных системах существуют другие механизмы, например, сигналы, но они здесь не рассматриваются, поскольку зависят от конкретной системы.

```
#include <iostream>
using namespace std;
class Vector {
  int* p;
  int sz;
public:
  enum { max = 32000 };
  class Range //особая ситуация индекса
  { public:
     int index;
     Range(int i) : index(i) { }  };
  class Size // особая ситуация "неверный размер"
   {   public:
     int index;
     Size(int i) : index(i) { }};
  Vector(int sz=1000);
  int& operator[](int i);};
Vector::Vector(int sz){
  if (sz<0 || max<sz) throw Size(sz); //генерация исключения
  p = new int [Vector::sz= sz];
  for(int i=0;i<sz;i++)
       p[i] = i;}
int& Vector::operator [](int i){
     if(i<0||sz<=i) throw Range(i); //генерация исключения
     return p[i];}
```

Рис. 20.2. Класс `Vector` с вложенными классами для генерации исключений

```
Vector* CreateV(int s){
     Vector *vnew;
     try{ vnew  = new Vector(s);
     cout<<(*vnew)[s];   }
     catch(Vector::Size s) //перехват исключения
     {
          cerr << "Wrong vector size "<< s.index << endl;
          vnew = CreateV(100);
     }
     catch(...)//перехват исключения
     {
          cerr<<"OPS"<<endl;
     }
     return vnew;}
```

Рис. 20.3. Функция для создания объекта типа `Vector`

```
void print(Vector& v, int i)
{
     try
     {
          cout<<v[i]<<endl;
          Vector *vnew = new Vector(64000);
     }
     catch(Vector::Range r) //перехват исключения
     {
          cerr << "Wrong index: "<< r.index << endl;
          print(v, 0);
     }
     catch(...)//перехват исключения
     {
          cerr<<"OOPS"<<endl;
     }
}
```

Рис. 20.4. Функция для печати объекта типа `Vector`

```
int main()
{
     Vector * vec[3];
     vec[0] = CreateV(6);
     vec[1] = CreateV(64000);
     vec[2] = CreateV(1000);
     print(*vec[0],6);
     print(*vec[1], 99);
     print(*vec[2],1000);
     return 0;
}
```

Рис. 20.5. Использование класса `Vector`

Механизм особых ситуаций является конструкцией с нелокальной передачей управления и его можно рассматривать как вариант оператора `return`. Поэтому особые ситуации можно использовать для целей, никак не связанных с обработкой ошибок.

21. ПРИВЕДЕНИЕ ТИПОВ

21.1. Динамическая идентификация типов

Недавнее добавление в проект стандарта языка Си++ механизма динамической идентификации типов (`RTTI - Run-Time Type Identification`) расширяет язык набором средств, позволяющих идентифицировать конкретные типы объектов во время выполнения программы, даже если известны только указатель или только ссылка на интересующий программиста объект.

```
#include <iostream>
using namespace std;

class A
{
public:
     virtual ~A(){}
};

class B:public A
{
     char* str;
public:
     B(const char* s)
     {
          str = new char[strlen(s)+1];
          strcpy(str, s);
     }
     ~B(){delete [] str;}
     const char* Get(){return str;}
};

int main()
{
     B b("Hello!");
     A& pa = b;
     if(typeid(pa)==typeid(B))
          cout<<((B&)pa).Get()<<endl;
     return 0;
}
```

Рис. 21.1. Использование операции `typeid`

Механизм динамического определения (идентификации) типов позволяет проверить, является ли некоторый объект объектом заданного типа, а также сравнивать типы двух данных объектов. Для этого используется операция `typeid`, которая определяет тип аргумента и возвращает ссылку на объект типа `const typeinfo`, описывающий этот тип. В качестве аргумента `typeid` можно также использовать и имя некоторого типа. В этом случае `typeid` вернет ссылку на объект `const typeinfo` этого типа. Класс `typeinfo` содержит операции-функции `operator==` и `operator!=`, которые используются для сравнения типов объектов. Класс `typeinfo` также содержит компонентную функцию `name ()`, возвращающую указатель на строку, содержащую имя типа аргумента (см. рис. 21.1.).

Операция `typeid` имеет две формы:

```
typeid (выражение),
typeid (имя_типа)
```

Если операнд операции `typeid` суть разыменованный нулевой указатель, порождается исключение `bad_typeid`.

21.2. Специальные операции приведения типов

Стандарт `ANSI` определяет специальный синтаксис операций приведения типа, позволяющий программисту воспользоваться преимуществами `RTTI` и, кроме того, указать точно, что он хочет получить в результате таких операций. Новых операций приведения четыре: `dynamic_cast`, `static_cast`, `reinterpret_cast` и `const_cast`.

Здесь необходимо вспомнить, для чего вообще может служить приведение типа. Можно назвать следующие случаи:

1. Чтобы изменить действительное представление данных либо поведение объекта, на который ссылается некоторый указатель. Простейшее приведение такого рода — преобразование целого типа в вещественный;
2. Чтобы изменить лишь интерпретацию компилятором некоторых данных, не меняя их действительного (физического) представления. Таково, например, приведение типа `int` к типу `unsigned` и наоборот;
3. Чтобы снять ограничения на возможные манипуляции с объектом, накладываемые модификатором `const`.

Эти три случая соответствуют, говоря несколько упрощенно, трем последним из перечисленных в начале раздела операций. Операция же `dynamic_cast` позволяет безопасно приводить типы в различных полиморфных иерархиях классов, в том числе с виртуальными базовыми классами.

Операция reinterpret_cast

Синтаксис данной формы операции приведения имеет вид:

`reinterpret_cast <целевой_тип> (аргумент)`

Такую операцию можно применить для того, чтобы изменить интерпретацию объекта без действительного преобразования данных.

Целевой_тип может быть типом ссылки, указателя, целым, перечислимым или вещественным типом.

Если целевой_тип — тип указателя или ссылки, то аргумент может быть указателем или ссылкой, а также числовой (вещественной, целой, перечислимой) переменной; когда целевым типом является числовой тип, то операнд может быть указателем или ссылкой.

Операция возвращает значение целевого типа.

Возможно, например, явное преобразование указателя в целый тип, равно как и обратная операция. Можно приводить указатель на функцию одного типа к указателю на функцию другого типа или на некоторый объект, при условии, что он (указатель на объект) имеет достаточную разрядность.

```
#include <iostream>
using namespace std;

class A {
public:
     int i;
     A(){i=10;}virtual ~A(){}} ;
class B: public A {
public:
     int i;
     B(int ii): i(ii) {}
     B(A&): i(11)
     {
          cout << "Derived conversion constructor... ";
     }
};

int main(){
     int i = 7;
     int *ip = &i;
     int temp = reinterpret_cast<int>(ip);
     cout << "Pointer value is "<< ip << endl;
     cout << "Representation of a pointer as int is " << hex
          << temp << endl;
     cout << "Convert it back and dereference:"
          <<reinterpret_cast<int*>(temp) << endl;
     return 0;
}
```

Рис. 21.2. Использование операции `reinterpret_cast`

Операция `const_cast`

Операция `const_cast` имеет ту же форму, что и предыдущая:

```
const_const <целевой_тип> (аргумент)
```

Целевой тип, возвращаемый такой операцией, может быть любым и должен отличаться от типа аргумента только модификаторами `const` и `volatile`.

```
#include <iostream>
#include <string.h>
using namespace std;

int main()
{
     const char *ip;
     ip = new char[20];
     strcpy(const_cast<char*>(ip), "New const string,");
     cout << ip << endl;
     strcpy(const_cast<char*>ip, "Test");
     cout << ip << endl;
     delete [] ip;
     return 0;
}
```

Рис. 21.3. Использование операции `const_cast`

Операция static_cast

Операция статического приведения типа

```
static_cast <целевой тип> (аргумент)
```

может выполнять преобразования между числовыми типами, а также между указателями либо ссылками на объекты классов, находящихся в иерархическом отношении (если оно однозначно и базовый класс — не виртуальный). Операция реализуется во время компиляции (см. рис. 21.4.).

Преобразования числовых типов происходят точно так же, как в случае обычной нотации приведений. Приведение указателей и ссылок возможно как от производного класса к базовому (тут все достаточно просто), так и от базового к производному (нисходящее приведение типа). Конечно, следует помнить, что во многих случаях нисходящее приведение указателя базового типа не будет безопасным, если только он не ссылается в действительности на представитель производного класса.

Если некоторый указатель может быть приведен к типу `T*`, то объект этого типа может быть приведен к типу `T&`.

Объект или значение могут быть приведены к объекту некоторого класса, если в данном классе объявлен соответствующий конструктор или имеется подходящая операция преобразования.

```
#include <iostream>
using namespace std;

class A {public: int i; A(){i=10;}virtual ~A(){}} ;
class B: public A {
public:
     int i;
     B(int ii): i(ii) {}
     B(A&): i(11) {
          cout << "Derived conversion constructor... ";}};

int main(){
     B b(22), *pb = &b;
     A &ra = static_cast<A&>(b); // Ссылка на b как базовый объект.
     A *pa = static_cast<A*>(&b); // Указатель на b как базовый
                                  //объект.
     cout << "Derived object: " << b.i << endl;
     cout << "Downcasting pointer to pointer: " <<
     static_cast<B*>(pa)->i << endl; // Приведение указателей.
     cout << "Downcasting referense to referense: "<<
     static_cast<B&>(ra).i<< endl; // Приведение к ссылке.
     cout << "Downcasting reference to object: ";
     cout << static_cast<B>(ra).i<< endl; // Приведение к объекту.
     return 0;}
```

Рис. 21.4. Использование операции `static_cast`

Операция dynamic_cast

Операция динамического приведения типа

`dynamic_cast <целевой_тип> (аргумент)`

не имеет аналогов среди операций, выполняемых с применением “классической” нотации приведения. Операция и проверка ее корректности при известных условиях происходит во время выполнения программы (см. рис. 21.5.).

Целевой тип операции должен быть типом указателя, ссылки или `void*`. Если целевой тип — тип указателя, то аргументом должен быть указатель на объект класса; если целевой тип — ссылка, то аргумент должен также быть соответствующей ссылкой. Если целевым типом является `void*`, то аргумент также должен быть указателем, а результатом операции будет указатель, с помощью которого можно обратиться к любому элементу «самого производного» класса иерархии, который сам не может быть базовым ни для какого другого класса.

Приведение от производного класса к базовому разрешается на этапе компиляции. Преобразования от базового класса к производному, либо перекрестные преобразования на некоторой иерархии, происходят во время выполнения программы. Операция нисходящего приведения типа допустима только в случае, если базовый класс (класс аргумента) является полиморфным.

```
#include <iostream>
using namespace std;

class A {
     public: int i;
     A(){i=10;}virtual ~A(){}} ;

class B: public A
{
public:
     int i;
     B(int ii): i(ii) {}
     B(A&): i(11)
     {
          cout << "Derived conversion constructor... ";
     }
};


int main()
{
     B b(22), *pb = &b;
     A a, *ppa=&a;
     A &ra = dynamic_cast<A&>(b); //Ссылка на b как базовый объект
     A *pa = dynamic_cast<A*>(&b); //Указатель на b как базовый объект
     cout << "Derived object: " << b.i << endl;
     // Приведение указателей
     cout << "Downcasting pointer to pointer: " <<
          dynamic_cast<B*>(pa)->i << endl;
     // Приведение к ссылке
     cout << "Downcasting referense to referense: "<<
           dynamic_cast<B&>(ra).i<< endl;
     // Приведение к объекту
     cout << static_cast<B*>(ppa)->i<< endl;
     return 0;
}
```

Рис. 21.5. Использование операции `dynamic_cast`

При попытке произвести некорректное преобразование операция возвращает нуль, если целевой_тип — указатель. Если ссылка, операция выбрасывает исключение типа `bad_cast`.

С помощью операции `dynamic_cast` можно выполнять нисходящее приведение виртуального базового класса, что невозможно сделать с применением обычной нотации приведений, при условии, что базовый класс является полиморфным и преобразование разрешается однозначно.

ГЛОССАРИЙ

Абстрактный класс	– это класс, объекты которого в программе создать нельзя
Адрес переменной	– это адрес области памяти, с которой связана данная переменная
Время жизни	– это время, в течение которого переменная связана с определенной областью памяти
Деструктор	– функция, вызываемая в момент разрушения объекта. Содержит освобождение динамической памяти, используемой объектом. Имя совпадает с именем класса, но в начале ставится знак ~
Дружественность	– отношения между классами или между классом и внешней функцией, позволяющие получить доступ к закрытым и защищенным элементам класса
Заголовочный файл	– файл, содержащий прототипы функций и описание пользовательских типов данных. Имеет расширение .h
Значение переменной	– это содержимое ячейки или совокупности ячеек памяти, связанных с данной переменной
Имя переменной	– это идентификатор, используемый в программах для ссылки на значение переменной
Индекс	– номер позиции элемента массива, заключенный в квадратные скобки

Инкапсуляция	– один из основных принципов объектно-ориентированного программирования. Позволяет скрывать реализацию класса от конечного пользователя класса
Исключение	– событие, которого не ждали и которое нужно каким-то образом обрабатывать
Класс	– это абстрактный тип данных, состоящий из элементов-данных и элементов-функций
Компилятор	– программа, обрабатывающая пользовательскую программу, проверяющая синтаксические ошибки, подключающая библиотечные файлы и формирующая исполняемый файл
Композиция	– включение в класс объектов других типов в качестве элементов
Конкретный класс	– класс, объекты которого в программе создать можно
Конструктор	– функция класса, используемая для инициализации элементов-данных класса. Имя конструктора совпадает с именем класса
Массив	– последовательная группа ячеек памяти, имеющих одинаковое имя и одинаковый тип
Наследование	– один из способов повторного использования программного обеспечения, при котором новые классы создаются из уже существующих классов путем заимствования их функций и атрибутов и обогащения этими возможностями новых классов

Область видимости	– это блок программы, из которого можно обратиться к этой переменной
Перегрузка функций	– определение в программе нескольких функций с одним именем, но разным списком параметров
Перегрузка операций	– переопределение встроенных операций для пользовательских типов данных
Переменная	– это объект данных, который явным образом определен и именован в программе
Полиморфизм	– возможность для объектов разных классов, связанных с помощью наследования, реагировать различным образом при обращении к одной и той же функции-элементу
Программа	– алгоритм + структура данных
Прототип функции	– это заголовок функции, после которого ставится точка с запятой. Необходим для использования функции до ее определения
Рекурсивная функция	– это функция, которая вызывает сама себя либо непосредственно, либо через другие функции
Связный список	– разновидность линейных структур данных, представляющая собой последовательность элементов, обычно отсортированную в соответствии с заданным правилом
Спецификатор доступа	– один из private, protected, public. Определяет доступ к элементу пользовательского класса из внешних функций или объектов других типов

Ссылка, ссылочный параметр	– это скрытый указатель, псевдоним параметра
Строка	– это массив символов, заканчивающийся специальным символом конца строки
Структура	– это абстрактный тип данных, состоящий из элементов-данных
Тип переменной	- связывает переменную с множеством значений, которые она может принимать
Указатель	– переменная, которая содержат в качестве своих значений адреса памяти.
Файл программы (кода)	– файл, содержащий определение пользовательских функций. Имеет расширение .cpp
Функция	– именованный блок программы, созданный для решения одной небольшой задачи
Шаблон класса	– параметризованный тип. Задает способ получения семейства сходных классов
Шаблон функции	– позволяет определять функции для любого типа данных. В отличие от перегруженных функций работа по созданию функций для конкретного типа ложится на компилятор

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

Основная литература

1. **Дейтел, Пол Дж**. Как программировать на C++ / Пол Дж. Дейтел, Харви Дейтел. – М.: ЗАО Издательство Бином, 1999. – 1024 с.

2. **Каррано, Ф.М.** Абстракция данных и решение задач на C++. Стены и зеркала / Ф.М. Каррано, Дж.Дж. Причард. – 3-е изд. – М.: Издательский дом «Вильямс», 2003. – 848 с.

3. **Страуструп, Б**. Язык программирования C++. Специальное издание / Б. Страуструп. – М.: Бином-Пресс, 2006. – 1104 с.

4. **Коплиен, Дж.** Программирование на C++. Классика CS / Дж. Коплиен. – СПб.: Питер, 2005. – 479 с.

5. **Элджер, Дж.** C++: библиотека программиста / Дж. Элджер — СПб.: Питер, 1999. – 259 с.

6. **Культин, Н.** C/C++ в задачах и примерах / Н. Культин. – СПб.: БХВ-Петербург, 2005. – 288 с.

7. **Керниган, Б.** Язык программирования C / Б. Керниган, Д. Ритчи. – 2-е изд. – М.: Издательский дом «Вильямс», 2006. – 304 с.

8. **Давыдов, В.Г.** Программирование и основы алгоритмизации: учебное пособие / В.Г. Давыдов. – М.: Высшая школа, 2003. – 447 с.

9. **Александреску, А.** Современное проектирование на C++ / А. Александреску. – М.: Издательский дом «Вильямс», 2002. – 336 с.

10. **Сатер, Г.** Новые сложные задачи на C++ / Г. Сатер. – М.: Издательский дом «Вильямс», 2005. – 272 с.

Дополнительная литература

11. **Мэйерс, С.** Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / Скотт Мэйерс. – М.: ДМК Пресс, 2006. – 300 с.

12. **Мэйерс, С.** Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / Скотт Мэйерс. – М.: ДМК Пресс, 2000. – 304 с.

13. **Хэзфилд, Р.** Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста / Ричард Хэзфилд [и др.]. – Киев: Изд-во «ДиаСофт», 2001. – 736 с.

14. **Дэвис, С.Р.** C++ для «чайников» / Стефан Р. Дэвис. – М.: Издательский дом «Вильямс», 2003. – 336 с.

15. **Лафоре, Р.** Объектно-ориентированное программирование в C++ / Р. Лафоре. – 4-е изд. – СПб.: Питер, 2004. – 923 с.

16. **Якушев,** Д.М. «Философия» программирования на C++ / Д.М. Якушев. – 2-е изд. – М.: Бук-пресс, 2006. – 320 с.

17. **Джосьютис, Н.** C++ стандартная библиотека. Для профессионалов / Н.М. Джосьютис. – СПб.: Питер, 2004. – 730 с.

18. **Щупак, Ю.А.** Win32 API. Эффективная разработка приложений / Ю.А. Щупак. – СПб.: Питер, 2007. – 572 с.

19. **Фленов, М.Е.** Программирование на C++ глазами хакера / М.Е. Фленов. – СПб.: БХВ–Петербург, 2004. – 336 с.