Под ред. проф. А. Д. Хомоненко

ПРОГРАММИРОВАНИЕ НА С++

Авт.: В.П.Аверкин, А.И.Бобровский, В.В.Веснич, В.Ф.Радушинский, А.Д.Хомоненко

Содержит систематическое изложение основных приемов программирования на языке С++: описание типов данных, объявление переменных, организация разветвлений и циклов, описание и использование массивов, указателей, функций. Рассматриваются понятия и приемы объектно-ориентированного программирования: определение классов и объектов, конструкторы и деструкторы, инкапсуляция, полиморфизм, наследование, шаблоны, обработка исключений, пространство имен, динамическая идентификация типов. Описывается технология разработки программ в среде Borland C++ Builder. Приводятся контрольные вопросы и задания.

Для студентов и преподавателей высших и средних учебных заведений.

Содержание

Предметный указатель

Символы

A

B

T

U

V

W

Н

О

П

Т

У

Ф

Ц

Ш

Э

Предисловие

Язык программирования C++ является одним из наиболее популярных средств объектно-ориентированного программирования, позволяющим разрабатывать программы, эффективные по объему кода и скорости выполнения. Последнее объясняется близостью языковых конструкций архитектуре ЭВМ и высокой выразительностью языка. C++ включает большое число операций и типов данных, средства управления вычислительными процессами, механизмы модификации типов данных и методы их обработки и, как следствие, является мощным языком программирования. Он позволяет описывать процессы обработки информации, начиная с уровня отдельных разрядов, видов и адресов памяти, переходя на основе механизмов объектно-ориентированного программирования к близким конкретным предметным областям понятиям.

По программированию на языке C++ имеется достаточно большое число изданий, среди которых, несомненно, важнейшее место занимают книги Бьерна Страуструпа, автора языка. В них содержится подробное изложение всех средств и тонкостей языка C++, сопровождаемое многочисленными примерами и ориентированное на подготовленного читателя. Методически удачными нам представляются работы Герберта Шилдта и В. В. Подбельского.

Эта книга задумана как учебное пособие, в котором в структурированном виде описываются важнейшие понятия языка, даются их определения, последовательно на примерах рассматриваются основные приемы программирования, отмечаются достоинства и недостатки использования различных конструкций и подходов.

В книге рассматриваются описание типов данных и переменных, организация разветвлений и циклов, описание и использование массивов, указателей, функций. Рассматриваются основные понятия и приемы объектно-ориентированного программирования: определение классов и объектов, конструкторы и деструкторы, инкапсуляция, полиморфизм, наследование. Достаточно подробно излагаются средства потокового ввода-вывода.

В пособии нашли также свое отражение относительно новые средства C++, к числу которых относятся:

- шаблоны, представляющие средство описания параметризованных функций и классов; стандартная библиотека шаблонов;
- пространство имен, используемое для управления локализацией имен переменных и объектов с целью избежания конфликтов имен;
- средства работы с исключениями, которые обеспечивают типовой порядок обработки возможных особых ситуаций и ошибок при выполнении программ;
- механизм динамической идентификации типов обрабатываемых объектов в процессе выполнения программы.

В книге описывается также технология разработки программ в среде Borland C++ Builder. При этом рассматриваются: состав и характеристика элементов интерфейса интегрированной среды разработки, а также особенности создания консольных приложений (выполняемых под управлением MS DOS) и Windows-приложений.

В языке C++ предусматривается обеспечение совместимости со всеми конструкциями языка более ранних версий. К примеру, допускается использование инструкций для подключения заголовочных файлов и применение старых функций ввода-вывода. В этой книге для сокращения объема излагаемого материала практически не приводится описание старых средств языка, а выдерживается ориентация в основном на соответствующие новые конструкции.

Пособие предназначено для студентов и преподавателей высших и средних учебных заведений. Может использоваться при подготовке к чтению лекций, проведению практических занятий и самостоятельного изучения материала. Для обеспечения контроля качества усвоения материала в конце каждого раздела приводятся контрольные вопросы и задания.

При рассмотрении конструкций языка и подготовке примеров мы ориентировались на стандарт C++ ISO/IEC 144882, принятый в августе 1998 года. Естественно, что не все из нововведений рассматриваемого нами стандарта C++ имеют воплощение в ранних версиях языка. В то же время современные версии систем программирования для языка C++ их, как правило, поддерживают.

При оформлении материалов использовались следующие соглашения о выделении текста: определения терминов даются полужирным кусивом; текст, на который обращается внимание читателя, набран курсивом.

Содержание пособия соответствует программам и может быть использовано при проведении занятий по следующим дисциплинам: «Информатика», «Технология программирования», «ЭВМ и программирование».

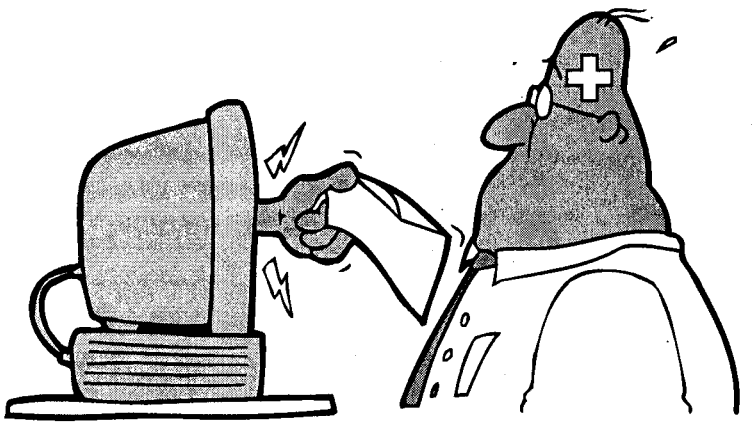
Материалы пособия получили апробацию при чтении ряда дисциплин, связанных с изучением приемов и технологии программирования на языке C++ и объектно-ориентированного программирования, преподавателями кафедры Математического обеспечения ВИКУ им. А. Ф. Можайского.

Авторы выражают признательность С.В. Овчинникову за активную помощь в подготовке материалов подразделов 4.4 и 15.2, благодарят сотрудниц лаборатории кафедры Е.А. Баглюк, Е.А. Кошмак, А.Ю. Михайлову и Н.В. Слесареву за помощь и моральную поддержку. Авторы выражают признательность руководству издательства «КОРОНА принт» С.Б. Катенину и В.К. Эртману за деловое и конструктивное сотрудничество при обсуждении проекта и издании книги.

А.Д. Хомоненко

ЧАСТЬ I

ОСНОВНЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ



1. ВВЕДЕНИЕ В C++

1.1. Общая характеристика языка

Язык программирования C++ разработан в начале 80-х годов сотрудником AT&T Bell Laboratories Бьерном Страуструпом (Bjarene Stroustrup). Основой для его создания послужил процедурно-ориентированный язык системного программирования C, позволяющий разрабатывать эффективные модульные программы решения широкого класса научно-технических задач. Эффективность написанных на языке C программ обусловлена возможностью учета в ней архитектурных и аппаратных особенностей ЭВМ, на которой программа реализуется. При создании C++ использовались определенные концепции языков Симула-67 и Алгол-68 и преследовались следующие цели:

- обеспечить поддержку абстрактных (определяемых пользователями) типов данных;
- предоставить средства для объектно-ориентированного программирования;
- улучшить существующие конструкции языка C.

В программе, написанной на языке C++, у программиста появилась возможность создания собственных типов данных на базе уже существующих. Их использование существенно облегчает обработку сложных структур данных.

Объектно-ориентированное программирование представляет собой технологию программирования, которая базируется на классификации и абстракции объектов. Язык C++ реализует три основные концепции объектно-ориентированного программирования: инкапсуляцию, наследование и полиморфизм.

Инкапсуляция — это объединение данных и обрабатывающих их функций в одном объекте. Целью инкапсуляции является автономность модулей; она позволяет локализовать последствия изменения структур данных конкретного модуля. Для остальных модулей изменения будут незаметны.

Язык C++ содержит средства, позволяющие разработать иерархию классов, или типов объектов. Соответствующий механизм называется наследованием. ***Наследование*** позволяет использовать разработанные ранее классы, что обеспечивает существенное сокращение процесса разработки программных модулей.

Полиморфизм можно определить как свойство, позволяющее использовать одно имя для обозначения действий, общих для родственных классов. При этом конкретизация выполняемых действий осуществляется в зависимости от типа обрабатываемых данных. Примером полиморфизма в C++ является перегрузка функций, позволяющая в рамках иерархии классов иметь несколько версий одной и той же функции. Решение, какая именно версия будет реализована, принимается в ходе выполнения программы.

Благодаря реализации перечисленных концепций объектно-ориентированное программирование обладает такими достоинствами, как:

- повышенная защищенность объектов от кода других частей программы как результат ипользования инкапсуляции;
- сокращение сроков построения программы на основе механизма наследования;
- исключение избыточного кода на основе концепции полиморфизма.

Справедливости ради отметим, что при объектно-ориентированном программировании необходимо решать достаточно сложную задачу модульного проектирования и определения классов и объектов.

Для улучшения языка С в С++ были сделаны следующие дополнения:

- определены новые операции доступа к глобальным объектам и управления динамической памятью;
- введен второй вид оформления комментариев;
- введены объектно-орентированные средства ввода-вывода;
- введены прототипы функций для согласования типов параметров и аргументов;
- достигнута возможность размещать в любом месте программы операторы объявления переменных и др.

В стандарт С++ вошли следующие важные дополнения:

- механизм обработки исключительных (ошибочных) ситуаций;
- динамическая (во время выполнения программы) идентификация и преобразование типов объектов;
- возможность использования параметризированных функций и классов — шаблонов;
- введение манипуляторов для работы с потоками ввода-вывода;
- введение новых базовых (встроенных) типов данных *bool* и *wchar_t*;
- введение стандартных проиводных типов данных и стандартных шаблонов;
- использование пространства имен как средства избежания конфликтов между одноименными переменными и отражения модульности программы;
- упрощение синтаксиса объявления переменных производных типов, определяемых пользователем;
- упрощение синтаксиса подключения заголовочных файлов стандартной библиотеки (не требуется указывать расширение *h*).

Материал в пособии соответствует стандарту языка С++ ISO/IEC 144882, принятому в августе 1998 г. Приведенные в книге примеры программ отлаживались с помощью компиляторов С++, поддерживающих рассматриваемые средства языка.

1.2. Технология разработки программ

Разработка программ на языке С++ ведется с помощью специальных комплексов программ, которые называются *системами программирования* и позволяют создавать программы на определенной *реализации* языка. Системы программирования даже одного производителя имеют различные *версии*, которые отражают развитие технологии программирования и эволюцию среды выполнения программ. Это стимулирует стремление максимально использовать *стандартные средства* языка для того, чтобы

снизить затраты на модификацию программ при изменении среды выполнения или при переходе на другую версию языка.

Вместе с тем многие важные аспекты языка определяются в *реализации* и не описываются стандартом. К их числу относится машинное кодирование символов, числовых и логических значений. Стандарт не определяет порядок создания программы для определенной среды выполнения. Детали процесса построения программ описаны в документации системы программирования. Если отвлечься от синтаксических, семантических и иных особенностей, присущих каждой конкретной системе программирования, процесс создания программ включает четыре этапа:

1. Написание и редактирование исходного текста программы с сохранением ее в виде *исходного файла* или *модуля*.

2. Компиляция программы и получение ее на определенном промежуточном языке с сохранением виде *объектного файла* или *модуля*.

3. Построение *исполнимого файла* или *модуля* путем объединения (компоновки) полученного объектного модуля программы с другими объектными модулями стандартных и специальных библиотек.

4. Отладка программы, которую можно проводить с помощью специального средства (*отладчика*), облегчающего обнаружение ошибок.

Соответственно, основными компонентами современных систем программирования являются:

- интегрированная среда программирования;
- редактор связей (компоновщик);
- библиотеки заголовочных файлов;
- стандартные и специальные библиотеки;
- библиотеки примеров программ;
- программы-утилиты;
- файлы документации.

Интегрированная среда программирования представляет собой программу, имеющую встроенный редактор текстов, подсистему работы с файлами, систему справочной помощи (Help-систему), встроенный отладчик, подсистему управления компиляцией и редактирования связей. Схема получения исполнимого модуля программы в интегрированной среде показана на рис. 1.1.

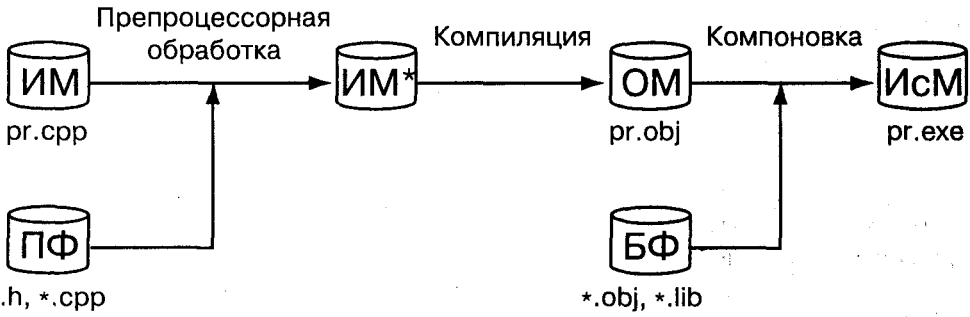


Рис. 1.1. Схема получения исполнимого модуля

Исходный модуль (ИМ) программы подготавливается с помощью встроенного или внешнего текстового редактора и размещается в файле с расширением *cpp*. После этого ИМ обрабатывается препроцессором и, в случае необходимости, к исходному тексту программы присоединяются ***подключаемые файлы*** (ПФ). В дальнейшем ***модернизированный исходный модуль*** (ИМ*) обрабатывается компилятором. Выявленные синтаксические ошибки устраняются, и безошибочно откомпилированный ***объектный модуль*** (ОМ) помещается в файл с расширением *obj*. Затем ОМ обрабатывается компоновщиком, который дополняет программу нужными библиотечными функциями из ***библиотечных файлов*** (БФ). Полученный модуль называется ***исполнимым модулем*** (ИсМ) и помещается в файл с расширением *exe*, который в дальнейшем исполняется.

Программы-утилиты — это вспомогательные программы, которые могут потребоваться при создании программ. В качестве примера программы-утилиты следует назвать программу библиотекарь, которая позволяет объединять объектные модули в один файл, называемый статической библиотекой. Такой файл в ПЭВМ имеет расширение *.lib*. Другим примером программы-утилиты является автономный отладчик. С помощью ***отладчика*** можно, в частности, выполнять программу в пошаговом режиме и контролировать содержимое всех переменных программы.

1.3. Пример программы

Приведем простейшую программу на языке C++, написанную в *процедурно-ориентированном стиле*. Программа вычисляет объем цилиндра по радиусу и высоте, которые вводятся с клавиатуры. Полученный результат выводится на экран монитора.

```
/* первая программа */                                                    //1
#include <iostream.h>// подключение средств ввода-вывода: cin, cout      //2
#include <math.h> // подключение математических средств: M_PI            //3
                                                                          //4
double circle_area(int radius);   //заголовок функции, вычисляющей       //5
                          //площадь окружности по заданному радиусу       //6
                                                                          //7
int main()                //заголовок главной функции программы          //8
  {                                                                       //9
                                                                          //10
    int  r,  h, v;        //объявление переменных для хранения значений  //11
                          //радиуса, высоты и объема цилиндра            //12
                                                                          //13
    /* вывод сообщения-подсказки  на экран */                             //14
    cout << "\nВведите радиус и высоту цилиндра, разделенные пробелами:";//15
    /* ввод значений с клавиатуры */                                      //16
    cin >> r >> h;                                                        //17
                                                                          //18
    //вычисление выражения и присвоение полученного значения переменной v //19
    v = h * circle_area( r );                                             //20
```

```
                                                                    //21
    /* вывод результата */                                          //22
   cout << "\nОбъем цилиндра радиуса " << r                         //23
        << " и высоты "  << h                                       //24
        << " равен "     << v;                                      //25
  return 0;              //возвращаемое значение в среду выполнения//26
  }                      //main                                     //27
                                                                    //28
// определение функции вычисления площади окружности               //29
double circle_area(double radius)                                   //30
  {                                                                 //31
   return M_PI*radius*radius; // вычисление выражения и возврат    //32
                          // полученного значения в место вызова    //33
  }                       // функции circle_area                    //34
```

В первой строке записан комментарий. В языке C++ комментарий можно оформить двумя способами:

- с помощью двух символов //, после которых следует комментарий до конца строки;
- с помощью парных символов /* и */, между которыми указывается комментарий.

Во второй строке записана директива препроцессора, подключающая заголовочный файл *iostream.h* (*input* — ввод; *output* — вывод; *stream* — поток; *header* — заголовок). Файл *iostream.h* обеспечивает включение в программу средства связи (потоки) со стандартными устройствами ввода-вывода. Стандартный поток ввода *cin* обеспечивает считывание символов с клавиатуры и преобразование их в соответствующие числовые значения переменных. Стандартный поток вывода *cout* обеспечивает вывод данных на экран дисплея.

В третьей строке подключается заголовочный файл, который обеспечивает возможность использования в программе библиотечных математических (*math*) функций и констант. Для нашей программы потребуется только число *PI*, которое имеет библиотечное имя *M_PI*.

При работе со стандартными библиотеками в соответствии со стандартом языка C++ можно опускать расширение заголовочных файлов. Например, вместо *#include <iostream.h>* можно записать *#include <iostream>*.

После пустой строки следует *объявление заголовка* (*прототип*) вспомогательной функции программы, что позволяет компилятору проводить контроль правильности использования (вызовов) функции, не ожидая определения тела функции. Вынесение вперед (в голову программы) только заголовка, а не всего определения функции позволяет программисту легче уяснить состав используемых вспомогательных функций и правила их использования (интерфейс). *Прототип функции* содержит информацию о количестве и типах параметров, о типе возвращаемого значения и заканчивается точкой с запятой. Оформление типовых последовательностей действий и вычислений в виде функций облегчает программирование сложных алгоритмов. *circle_area* — это имя функции, а *radius* — имя аргумента или формального параметра. Ключевое слово *double* (вещественные числа с удвоенной точностью) говорит компилятору о типе значений, которые могут принимать аргументы функции и ее результат, возвращаемый в место вызова функции.

Пустые строки используются для разделения программы на основные части.

Любая программа на языке C++ должна иметь в своем составе функцию с именем *main*, с которой начинается выполнение программы. Поэтому эта функция называется главной и имеет соответствующее фиксированное имя. Заголовок функции *main()* начинается с ключевого слова *int* (*integer* — целое), которое говорит компилятору, что данная функция возвращает в место вызова целочисленное значение. Местом вызова главной функции является среда выполнения или среда операционной системы. Нулевое возвращаемое (*return*) значение (строка 26) говорит об отсутствии ошибок во время выполнения функции *main()*. В круглых скобках должен находиться список формальных параметров. В нашей функции *main()* параметры отсутствуют, т. е. список пуст.

Тело функции представляет собой последовательность *операторов*, и оно заключается в фигурные скобки, т. е. является *блоком*. В *определении функции* тело функции следует сразу за заголовком. Обычно в начале тела функции объявляются важнейшие переменные с помощью соответствующих операторов. *Оператор* представляет собой законченное предложение языка и заканчивается точкой с запятой. *Операторы объявления* переменных начинаются с *имени типа*, после которого указывается список *имен переменных* данного типа. В частности, *прототип функции* — это оператор объявления заголовка функции.

В теле функции *main()* объявлены три переменные *r*, *h* и *v* (в 11 строке). В результате объявления под переменные будет *выделена оперативная память* в соответствии с их типом (по 8 байтов).

Дальше в теле функции находятся операторы (15, 17, 20, 23–26), которые задают действия, осуществляемые вычислительной машиной. Каждый такой оператор содержит *выражение*, состоящее из *операндов* и *знаков операций*.

В 15 строке записан *оператор вывода* в стандартный поток *cout*. Операция << (*вставки* в поток) обеспечивает вывод на экран через стандартный поток (левый операнд) значение правого операнда. Здесь правый операнд — это строка символов, заключенная в двойные кавычки, или *строковая константа*. Первые два символа *\n* строки означают *управляющий символ «новая строка»*, который обеспечивает перевод курсора в начало следующей строки экрана и на экране не отображается. Таким образом, выполнение программы начинается с вывода подсказки на экран дисплея:

```
Введите радиус и высоту цилиндра, разделенные пробелами:
```

В 16 строке программы записан *оператор ввода* из стандартного потока *cin*. Операция >> (*извлечения* из потока) обеспечивает *считывание* данных с клавиатуры, их *преобразование* в соответствии с типом правого операнда и присвоение полученного значения правому операнду.

В 20 строке записан *оператор присваивания*, в котором в соответствии с *приоритетами* выполнения действий сначала осуществляется *вызов функции circle_area(r)* для фактического параметра *r*, в результате чего будет получено значение площади основания цилиндра. Полученное значение умножается (*) на значение переменной *h*, и результат произведения будет *присвоен* (=) переменной *v*.

С 23 по 25 строки записан оператор вывода в стандартный поток *cout* значений исходных данных и полученного результата с сопутствующими комментариями, оформленными в виде строковых констант.

После определения главной функции следует определение вспомогательной функции *circle_area()*. В теле функции имеется один *оператор возврата* (*return*). В результате выполнения этого оператора будет вычислено значение площади и полученный результат будет возвращен в место вызова функции.

Для удобства фигурная скобка, закрывающая блок определения (*реализации*) функции, комментируется именем функции.

При анализе приведенной программы можно выделить следующие крупные части типового исходного модуля:

1) *подключение* заголовочных файлов библиотек (с функциями, константами и др.);

2) *объявления* пользовательских символических констант, вспомогательных функций, классов и типов данных, глобальных переменных;

3) *заголовок главной функции*;

4) *определение главной функции* — блок, содержащий:

- *объявление* локальных переменных и констант и их инициализацию (присвоение начальных значений);
- *ввод* исходных данных (диалог с пользователем);
- *обработка* — обращение к функциям, вычисление выражений, выполнение операторов;
- *вывод* результата;
- *возвращение кода завершения главной функции*;

5) *определения пользовательских вспомогательных функций* и методов классов; для каждой функции после заголовка размещается блок, содержащий:

- *объявление* локальных переменных и их инициализацию;
- *обработку* формальных параметров с использованием локальных переменных;
- *возвращение* результата.

Подобное структурирование текста исходного модуля, который размещается в *одном* файле с расширением *cpp*, не единственно возможное. Но следование данному образцу облегчает понимание и отладку программ.

Общие принципы, позволяющие написать синтаксически правильную программу, таковы:

- прежде чем использовать функции, переменные, типы данных, следует объявить их или подключить файлы с их объявлениями (сделать известными компилятору);
- чтобы вызов функции мог быть выполнен, функция должна быть определена, т. е. описаны действия, которые она осуществляет;
- в любой последовательности действий нужно стремиться к триаде: инициализация (ввод), обработка, возвращение значения (вывод).

При определении функции полезно придерживаться следущих рекомендаций:

- объявить и проинициализировать входные и выходные переменные;
- промежуточные переменные объявлять и инициализировать в начале блоков или перед операторами, в которых они используются.

Правила оформления текста программы, направленные на облегчение понимания смысла и повышение наглядности, таковы:

1) разделять логические части программы пустыми строками;

2) разделять операнды и операции пробелами;

3) для каждой фигурной скобки отводить отдельную строку;

4) в каждой строке должно быть, как правило, не более одного оператора;

5) ограничивать длину строки 60–70 символами;

6) отступами слева отражать вложенность операторов и блоков;

7) длинные операторы располагать в нескольких строках;

8) проводить алгоритмизацию так, чтобы определение одной функции занимало, как правило, не более одного экрана текста;

9) стремиться использовать типовые заготовки фрагментов программ, включая и типовую структуру блока и программы в целом.

Конечно, эти правила нужно использовать совместно с *правилами именования* элементов программы и *комментирования* текста программы, которые рассмотрены в п.2.1.

Контрольные вопросы

1. Какие цели преследовали разработчики языка программирования C++ ?
2. Нарисуйте схему, отражающую порядок разработки программы в интегрированной среде.
3. Назовите основные компоненты программы на языке C++.
4. Объясните, каким образом происходит ввод исходных данных.
5. Назовите основные концепции объектно-ориентированного программирования.
6. Укажите допустимые варианты записи директивы *#include* для подключения заголовочных файлов.
7. Какое имя имеет главная функция программы?
8. Что входит в состав тела функции?
9. Какая информация содержится в прототипе функции?
10. Что представляет собой оператор?
11. Назовите основные части типового исходного модуля.
12. Каким образом записывается комментарий?
13. Назовите правила оформления исходного текста программы, направленные на повышение наглядности.

2. ТИПЫ ДАННЫХ И ВЫРАЖЕНИЯ

2.1. Алфавит и идентификаторы

При написании программ на языке C++ используются символы, составляющие его ***алфавит***. Набор символов зависит от среды выполнения. На ПЭВМ широко используется символьный набор ISO 646-1983, называемый кодом ASCII (American Standart Code for Information Interchange – американский стандартный код обмена информацией). Он содержит *латинские буквы*, *арабские цифры*, *специальные* и *управляющие* символы, которые в своем большинстве входят в состав *алфавита языка* C++. Каждый символ кодируется семибитным значением. Для представления *кириллических символов* используется восьмибитный *расширенный* ASCII-код, в котором единичное значение старшего бита говорит об использовании дополнительного символьного набора.

Алфавит C++ включает *латинские* прописные и строчные *буквы*: A,..., Z, a,..., z, *арабские цифры*: 0, 1,..., 9, *специальные символы*:

```
+ - * / < > = | & ! \ ~ ' @ # $ % ^ ? _ : ; , . ( ) [ ] { } " .
```

В качестве *символа-разделителя* элементов (слов) предложений языка используется *пробел*, который на экране не отображается, а для наглядности при записи на бумаге часто обозначается символом ␣. Предложения (операторы) языка обычно заканчиваются точкой с запятой. Исключение составляют директивы препроцессора, начинающиеся с символа #, составные операторы и блоки определения функций, которые обрамлены фигурными скобками – { }.

Кроме того имеются *управляющие символы*, которые непосредственно на экране не отображаются. Для их записи используются специальные приемы, которые будут рассмотрены позже. В качестве примера записи управляющего символа «*горизонтальная табуляция*» приведем `'\t'`. Еще один пример записи управляющего символа «*новая строка*» был рассмотрен в первой программе на языке C++.

Использование кириллических символов в некоторых случаях возможно и целесообразно (в комментариях, символьных строках, названиях файлов, если это допускает среда выполнения). Но пока единого стандарта на кодировку кириллических символов нет. Поэтому могут быть сложности при переносе таких программ с одного компьютера на другой, при переходе из одной среды выполнения в другую.

Специальные символы используются для обозначения (именования) *операций* и записи *выражений*. Например, запись *((a + b) * z)* является *выражением*, задающим вычисление суммы значений переменных *a* и *b* с последующим умножением на значение переменной *z*.

Совокупность двух и трех специальных символов может задавать *имя* (*знак*) *операции*.

Например, ++ и – – являются знаками *унарных* операций *инкремента* (увеличения значения *операнда* на 1) и *декремента* (уменьшения значения *операнда* на 1) соответственно. Так, оператор инкремента переменной *time* имеет вид

```
time++;
```

Имена имеют многие элементы программы: константы, переменные, типы данных, функции и ряд других. Такие имена являются *идентификаторами*. Имена вводятся для того, чтобы отличать (идентифицировать) различные элементы одного вида (типа) от других и оперировать (производить действия) с ними. ***Идентификатором*** называется последовательность символов из латинских букв, символа подчеркивания и арабских цифр, которая начинается с буквы и служит для именования различных элементов программы. Примеры идентификаторов: *var1, Table7, bad_call, _limit.*

Идентификаторы могут включать любое число символов, из которых значимыми являются первые 32, т. е. длинные идентификаторы считаются различными, если у них отличаются последовательности из первых 32 символов.

Строчные и заглавные буквы суть разные символы. Поэтому идентификатор *Radius* отличается от идентификатора *radius.*

Некоторые идентификаторы языка зарезервированы в служебных целях и их нельзя использовать для именования переменных, констант и функций. Такие идентификаторы называют ***служебными*** или ***ключевыми*** *(keyword)* словами и входят в алфавит языка. Используемые в стандарте C++ ключевые слова приведены в приложении.

При подключении *стандартных библиотек* добавляется ряд специальных идентификаторов, таких как *cerr, cin, clog, complex, cout, list, map, set, size_t, string, valarray, vector.* Их также не рекомендуется использовать в качестве идентификаторов.

Рекомендации по ***именованию:***

- использовать имена из постановки задачи;
- давать короткие осмысленные имена, отражающие назначение переменной, функции, объекта или типа;
- не начинать с символа подчеркивания, поскольку такой прием широко используется в библиотеках системы программирования;
- следовать единой системе именования; здесь существуют различные варианты, например:
  - начинать с прописной буквы, если требуется подчеркнуть уникальность идентификатора;
  - использовать символ подчеркивания или прописные буквы внутри идентификатора для построения хорошо читаемых сложных идентификаторов.

Обычно редко удается в именах элементов программы прокомментировать ее содержание. Поэтому для пояснения отдельных частей или всей программы используют *комментарии.* Для введения однострочного комментария используют пару символов //, после которых следует поясняющий текст до конца строки. Многострочные комментарии начинаются с пары символов /* и заканчиваются парой символов */.

Рекомендации по ***комментированию:***

- начинать программу с кратких комментариев, описывающих основные этапы алгоритма, переменные для хранения исходных данных, промежуточных и выводимых результатов;
- писать комментарии в терминах постановки задачи и выбранного метода решения;

- не вставлять комментарии в середину строки программы;
- не писать очевидных комментариев;
- начинать комментарий с той же позиции в строке, что и комментируемый текст.

2.2. Операции, выражения и операторы

Операции и ***выражения*** задают определенную последовательность действий, но не являются законченными предложениями языка. *Простые выражения* содержат знак операции и операнды. Пример простого выражения с *бинарной* операцией вычитания: *3.4 – x.*

Операции могут проводиться с одним, двумя и тремя *операндами*. Соответственно, различают ***унарные***, ***бинарные*** и ***тернарные*** операции.

Операнд представляет собой элемент-участник операции. Операндами могут быть *константы*, *переменные*, *вызовы функций* и *выражения*.

Рассмотрим основные арифметические операции.

+, –, *, / — знаки *бинарных* операций сложения, вычитания, умножения и деления соответственно. Если в операции деления оба операнда целые числа, то результат операции тоже целое число. Следовательно, имеется две разновидности операции деления: *целочисленное деление* (деление с остатком) и *деление без остатка* (результат вещественное число). Например, 5/2 = 2, а 5.0/2 = 2.5.

Для целых чисел определена операция % — *деление по модулю* (нахождение остатка от деления). Например, 5%2 = 1.

Примером *унарной* арифметической операции является операция *изменения знака числа*, обозначаемая символом «минус», который стоит перед одним операндом:

```
- x  + a  - 35.2
```

Как видим, смысл символа операции зависит от контекста, т. е. тех условий, в которых используется операция.

Компилятор по *имени операции* и *типам операндов* определяет возможность ее выполнения и необходимые для этого машинные действия и свои собственные действия. После выполнения ***операции*** получается результат определенного *типа*, подставляемый в текст программы вместо знака операции и операндов, участвующих в данной операции. Например, можно считать, что в результате выполнения операции умножения строка *(t * 5.2)* заменится строкой *15.6*, если *t* = 3.

Вызов функции представляет собой указание имени вызываемой функции, за которым в круглых скобках указывается список аргументов (возможно пустой). Примеры выражений с вызовом функций:

```
a*sin(x+k)+b*cos(x-k)
pow(2, n+1)-1
```

Последнее выражение может быть математически записано в виде -1.

Можно считать, что во время выполнения программы результат, возвращаемый вызванной функцией, заменяет вызов функции.

Выражение — это последовательность знаков операций, операндов и круглых скобок, которая задает вычислительный процесс получения результата определенного

типа. *Простейшим выражением* является *константа*, *переменная* или *вызов функции*. Можно считать, что при выполнении программы результат вычисления выражения заменяет само выражение. Например, выражение

```
-2-(cos(x)+5*y)
```

заменится значением -8.0, если $x = 0$ и $y = 1$.

Символ = обозначает бинарную операцию *простого присваивания*, в результате выполнения которой значение правого операнда присваивается левому операнду. Пример выражения с операцией присваивания имеет вид

```
v=0.5*a*t*t+7
```

Результатом выполнения операции присваивания является присвоенное значение. Поэтому возможно следующее выражение *множественного присваивания*

```
z=y=x=t=state0
```

В результате вычисления данного выражения переменные z, y, x, t примут значение *state0*, если все они одного *типа*.

Порядок вычисления выражения определяется расположением знаков операций, круглых скобок и *приоритетами* выполнения операций. Например, результат вычисления выражения $(11 + 25 / 5 + 7)$ не равен результату вычисления выражения $(11+25)/(5+7)$.

Выражения с наивысшим приоритетом вычисляются первыми. Приоритеты операций приведены в табл. 2.1. Операции, находящиеся в начале таблицы, имеют более высокий приоритет.

Таблица 2.1

Приоритеты и порядок выполнения операций

Приоритет	Знаки операций	Названия операций	Порядок выполнения
1	::	разрешение области видимости	слева
2	. –>[ ] ( ) ++ – – *typeid* *dynamic_cast* *static_cast* *reinterpret_cast* *const_cast*	выбор элемента по имени выбор элемента по указателю выбор элемента по индексу вызов функции или конструирование значения постфиксный инкремент постфиксный декремент идентификация типа преобразование с проверкой при выполнении преобразование с проверкой при компиляции преобразование без проверки константное преобразование	слева
3	*sizeof* ++ – – ~ !	размер операнда в байтах префиксный инкремент префиксный декремент инверсия (поразрядное НЕ) логическое НЕ	справа

Продолжение табл. 2.1

Приор-итет	Знаки операций	Названия операций	Порядок выполнения
	+ – & * *new* *delete* (имя_типа)	унарный плюс унарный минус адрес разыменование выделение памяти или создание освобождение памяти или уничтожение преобразование типа	
4	.* –>*	выбор элемента по имени через указатель выбор элемента по указателю через указатель	слева
5	* / %	умножение деление остаток от деления целых (деление по модулю)	слева
6	+ –	сложение вычитание	слева
7	<< >>	сдвиг влево сдвиг вправо	слева
8	< > <= >=	меньше больше меньше или равно больше или равно	слева
9	= = !=	равно не равно	слева
10	&	поразрядное И	слева
11	^	поразрядное исключающее ИЛИ	слева
12	|	поразрядное ИЛИ	слева
13	&&	логическое И	слева
14	||	логическое ИЛИ	слева
15	? :	условная	справа
16	= *=, /=, %=, +=, –= <=, >>= , &=, ^=, |=	присваивания (простое и составные)	справа
17	*throw*	генерация исключения	справа
18	,	последовательность выражений	слева

Если в выражении содержится несколько операций одного приоритета на одном и том же уровне, то их обработка производится в соответствии с порядком выполнения: справа налево или слева направо. Например:

```
y=x/2+a*5%15;          // y=(x/2)+((a*5)%15)
x=y--+d/10;            // x=(y--)+(d/10)
```

В языке C++ предусмотрены две нетрадиционные операции ++ и -- соответственно для увеличения и уменьшения на единицу значения операнда (унарные операции *инкремента* и *декремента*). Их можно указывать перед операндом и после операнда. В первом случае (*++x* или *--x*) значение операнда *x* изменяется перед его использованием в выражении , а во втором (*x++* или *x--*) — после его использования. Например:

```
b=b1=3; c=c1=5; a=b+c++;       /* b=3; c=5+1=6; a=3+5=8; */
a1=b1+(++c1);                  /* b1=3; c1=5+1=6 ; a1=3+6=9 */
```

<, >, ==, >=, <=, != суть знаки *операций отношения* (меньше, больше, равно, больше или равно, меньше или равно, не равно), используемые в *логических выражениях*;

&&, ||, ! суть знаки *логических операций* «И», «ИЛИ» и «НЕ»;

&, |, ^, <<, >>, ~ суть знаки *поразрядных логических операций*, выполняемых над одноименными битами машинного слова: «И», «ИЛИ», исключающее «ИЛИ», сдвиг влево, сдвиг вправо, инверсия;

? и : суть знаки, используемые в *условной тернарной операции*. Например, результатом выражения

```
(a>0) ? a:0
```

будет значение *a*, если *a* > *0* и нулевое значение — в противном случае.

*, & суть знаки *адресных операций*;

, суть знак специальной операции задающей последовательность выражений (*«операция запятая»*) . Например, выражение

```
(i=1, j=4, k=7)
```

состоит из трех простых выражений.

[и] суть квадратные скобки, используются при работе с массивами для задания размеров массива и доступа к элементу массива;

(и) суть скобки, используемые для изменения очередности выполнения операций в выражениях и при указании аргументов функций;

sizeof (имя_тип) или *sizeof выражение* — операция вычисления размера типа или значения в байтах.

Удобны в использовании следующие *составные операции присваивания*:

```
*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
```

Они применяются для компактной записи *операторов присваивания*. Например, при выполнении одного из двух операторов *a = a + b;* или *a += b;* результат будет одинаков.

Операторы так же, как операции и выражения, задают определенную последовательность действий компилятора или вычислительной машины, но, в отличие от выражений, являются законченными предложениями языка. Они могут содержать несколько выражений, разделенных запятой.

Например, *оператор вывода-ввода*, содержащий два выражения, имеет вид:

```
cout << "Введите значение x:",  cin  >>  x;
```

Здесь *cout* и *cin* правые операнды операций вставки (<< – вывода) и извлечения (>> – ввода) соответственно, а строковая константа и переменная x – правые операнды этих операций. Компилятор по *типу* правого операнда определяет, что имеет дело с операциями ввода-вывода, а не с операциями логического сдвига.

Обычно операторы заканчиваются *точкой с запятой. Пустой оператор* содержит только точку с запятой. Простые операторы можно объединить в *составной оператор* или *блок* с помощью фигурных скобок, за которым точку с запятой можно не ставить. Составные операторы и операторы, полученные в результате вложения (суперпозиции) операторов называются *сложными*.

Операторы имеют называния, отражающие их назначение или осуществляемые действия. Так, оператор, в котором используется операция присваивания, обычно называют ***оператором присваивания***. Например,

```
y=(a*x+b)*x+c;
```

Примером *оператора-выражения* является

```
count–;
```

При его выполнении значение переменной *count* уменьшится на единицу.

Операторы, в которых объявляются константы, переменные, заголовки функций и типы данных называются *операторами объявления*. Имеются операторы с другими названиями.

Операторы, задающие действия вычислительной машины, могут быть помечены идентификатором, который называется *меткой* и заканчивается двоеточием. Метки используются для перехода к ним с помощью оператора *goto* (*безусловного перехода*).

2.3. Классификация типов данных

В программе задаются действия с переменными и константами, которые предварительно должны быть объявлены для того, чтобы сообщить компилятору:

1) имя переменной или константы;
2) размер памяти, необходимый для хранения значений;
3) какие действия можно выполнять с переменной или константой;
4) вид и способ выделения памяти под переменную и константу;
5) начальное значение переменной или значение константы.

Перед рассмотрением способов объявления констант и переменных в программе охарактеризуем типы данных языка C++, которые играют важную роль в обработке данных. Под ***типом данных*** понимают множество допустимых значений этих данных и множество разрешенных операций над ними. Одновременно тип данных определяет и размер памяти, который занимают переменные и константы данного типа. Каждый тип данных в языке имеет *имя* (идентификатор), простое или составное.

Заметим, что память не выделяется для типа данных, а выделяется для размещения переменной или константы.

В языке C++ выделяют следующие *категории* ***типов***:

1) *базовые типы* данных;

2) *производные* (определяемые) типы.

Базовые типы имеют имена, которые являются ключевыми словами языка.

К ***базовым*** типам относятся: *скалярные типы* и *пустой* тип — *void*.

Тип ***void*** не имеет значения и введен в основном для описания функций, не возвращающих значений, и для некоторых других целей.

Скалярные типы делятся на *целочисленные* и *вещественные* типы.

Логический тип, *символьные* и *целые* типы данных являются *целочисленным типом*, для которого определены все операции с целыми числами.

Производные типы определяются (образуются) на основе базовых типов. Производные типы делятся на *скалярные* и *структурные* (агрегатные).

К скалярным производным типам относятся:

- *перечисления* (***enum*** — enumeration) — множество поименованных целых значений;
- *указатели* (*имя_типа* *******);
- *ссылки*(*имя_типа* **&**).

Структурными типами являются:

- *массивы* (*тип_элемента имя_массива*[*число_элементов*]);
- *структуры* (***struct***);
- *объединения* (***union***);
- *классы* (***class***).

В структурах, объединениях и классах могут использоваться *битовые поля* (*bit field*).

Дерево классификации типов языка C++ приведено на рис. 2.1. Рассмотрим подробнее базовые типы данных.

2.4. Объявление переменных

Обрабатываемые в программе данные можно разделить на *переменные* и *константы*. Перед использованием переменные и константы должны быть объявлены с помощью оператора объявления вида

```
[<спецификатор класса памяти>] [const] <спецификатор типа>
<идентификатор> [= <начальное значение>]
[,<идентификатор> [= <начальное значение>]]... ;
```

Например:

```
int a=5, y;
const float  g = 9.81, C = 0.577216;
```

Здесь и в дальнейшем квадратные скобки означают необязательность операнда.

Ключевое слово *const* указывает, что записанные справа идентификаторы являются ***символическими константами*** (константными переменными). При этом значение константы задается обязательно и в программе изменяться не может. Кроме константных переменных, константы могут задаваться в виде ***литеральных*** (самоопределенных) констант (см. подраздел 2.5).

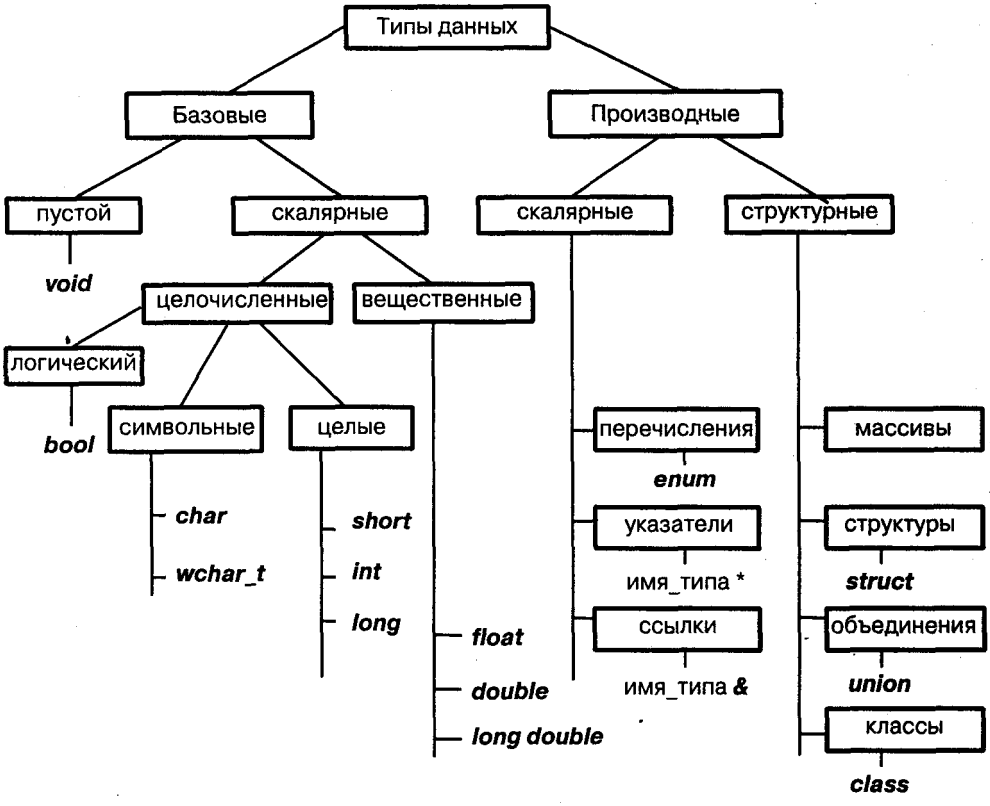


Рис. 2.1. Дерево классификации типов языка C++

В табл. 2.2 привидены имена основных скалярных базовых типов, диапазоны допустимых значений и размер памяти, отводимой под переменные и константы данных типов применительно к ПЭВМ.

Тип *bool* введен в стандарте C++, раньше он определялся на основе целых типов. *Логические переменные* типа *bool* могут принимать одно из двух значений: *false* (ложь) или *true* (истина). По определению значение *false* равно 0, а *true* равно 1. Логические переменные широко используются в операциях сравнения, логических операциях и логических выражениях. Размер переменной зависит от реализации, но обычно составляет 2 байта. Примеры объявлений:

```
bool reload = false, in_range = true;
```

В переменной типа *char* может храниться один из ASCII-символов. Например:

```
char ach = 'f';
```

Одиночные кавычки используются для задания символьного значения переменной. Запись вида `'s'` называют *символьной константой* или *символьным литералом.*

Для явного задания диапазона можно использовать ***модификаторы*** *signed, unsigned.* В объявлениях символьных типов обычно эти модификаторы используют-

Таблица 2.2

Характеристики основных типов данных языка C++

Имя типа	Размер памяти, байтов (16/32-разрядная)	Диапазон значений для 16-разрядной архитектуры	
[signed] char	1 / 1	–128	127
unsigned char	1 / 1	0	255
[signed] short [int]	2 / 2	–32768	32767
unsigned short [int]	2 / 4	0	65535
[signed] int	2 / 4	–32768	32767
unsigned int	2 / 4	0	65535
[signed] long [int]	4 / 4	–2147483648	2147483647
[unsigned] long [int]	4 / 4	0	4294967295
float	4 / 4	3.4e–38	3.4e38
double	8 / 8	1.7e–308	1.7e308
long double	10 / 10	3.4e–4932	3.4e4932

ся, когда значение символа задается с помощью числового значения или требуется «обратить внимание» на зависимость от реализации.

```
unsigned char scode = 87;   // эквивалентно 'W'
```

Для работы с символами из расширенных наборов, таких как UNICODE, применяется тип *wchar_t*.

Переменные и константы целых типов также могут объявлятся с помощью модификаторов *signed* и *unsigned*. При указании модификаторов *short* и *long* допускается опускать имя *int*, которое подразумевается по умолчанию. Модификатор *signed* также подразумевается по умолчанию.

Размеры целых типов зависят от реализации, но для всех версий языка C++ принято, что выполняется следующая система нестрогих неравенств.

```
sizeof( short ) <= sizeof( int ) <= sizeof( long ).
```

Кроме того, во всех реализациях гарантируется

```
1 = sizeof(char) <= sizeof(short);
1<= sizeof(bool)<=sizeof(long);
```

Типы с плавающей запятой (точкой) или вещественные типы представлены тремя размерами, характеризующими точность представления вещественных чисел:

float — одинарной точности;

double — двойной точности;

long double — расширенной точности.

В файлах *limits.h*, *values.h* и *float.h* определены константы, характеризующие диапазоны значений базовых типов среды выполнения. Используя их, можно создавать программы, пригодные для перенесения в другую среду выполнения. Другой путь построения мобильных (переносимых в другую среду) программ — использование унарной операции *sizeof(имя_типа)*, результат выполнения которой равен размеру памяти, необходимой для переменных данного типа. Константы числовых диапазонов среды выполнения можно использовать для создания более корректных вычислительных программ.

Пример 1. Объявление данных.

```
int x,y; short a, b;
long e; char st;
unsigned i,j; /* или unsigned int i, j; */
float v, z;  double u;
```

Используя спецификатор типа *typedef*, можно в своей программе вводить удобные имена для сложных описаний типов. Например:

```
typedef unsigned char cod;
cod p;
```

Здесь определена переменная *p* типа *cod*, который описан спецификатором *typedef*. Целью такого объявления обычно является введение короткого синонима, определяющего назначение типа данных в программе.

2.5. Задание констант

По способу задания константы можно разделить на *символические* и *литеральные* (самоопределенные).

Обычно константы задаются в выражениях в качестве конкретных значений операндов и значений параметров функций. Сама константа является простейшим выражением.

Литеральная константа представляет собой константу, тип и значение которой определяются ее внешним видом. В необходимых случаях память под константу выделяется автоматически, но может быть не выделена совсем, например для литералов в составе константного выражения. Содержимое выделенной памяти не изменяется и определяется типом и значением константы.

Различают следующие *виды* ***литеральных констант:***

1) числовые;
2) символьные;
3) строковые.

Рассмотрим на примерах эти виды констант.

Числовые константы:

- *вещественные* типа *double*:

  3.14 эквивалентна 314e–2, также эквивалентна 0.314E1;

  –2.5 эквивалентна –0.25e1;

- *целые*:
  - *десятичные* типа *int*: 0, 278, –579 — используются десятичные цифры;
  - *восьмеричные*: 00, 01, ..., 077777, ...; — используются восьмеричные цифры, запись числа начинается с нуля;
  - *шестнадцатеричные*: 0x0000, 0x0001, ..., 0x7FFF, ..., 0xFFFF, ...

По форме записи числовой константы компилятор определяет ее тип: по умолчанию целые десятичные константы имеют тип *int*, вещественные — *double*, если они принадлежат соответствующим множествам значений указанных типов. Если константа выходит за пределы множества значений типа *double* или *int*, то она относится к типу, следующему за *double* или *int* по мощности множества допустимых значений.

Тип числовой константы можно задать явно с помощью *суффиксов*:

l, L — long int; например — *–35L*;

uh, Uh, UH, hu, Hu, hU — unsigned short; например — *227UH*;

f, F — float; например — *1.5F*;

l, L — long double; например — *1E–10L*;

Символьные (литеральные) константы:

- *клавиатурные*: `'1'`, `'t'`, `'Y'` — клавиатурный символ задается в апострофах;
- *кодовые* — для задания некоторых управляющих и разделительных символов:

```
'\n', '\a', '\\', '\t','\'','\?'
```

- *кодовые числовые* — для задания любых ASCII-кодов символов, имеют вид:

`'\xHH'`, `'\XHH'`, `'\0OOO'`, где H, O — шестнадцатеричная (hexadecimal) и восьмеричная цифра (octal) соответственно;

`'\0'` эквивалентна `'\x0'`;

`'\x5C'` эквивалентна `'\0134'` эквивалентна `'\\'`.

Строковые константы — последовательности символов, ограниченные двойными кавычками:

```
"This string constant",
 "ERROR:".
```

Для хранения строковой константы требуется $n+1$ байтов, где n — число символов между ограничителями строковой константы. Дополнительный байт последний и требуется для хранения кода 000, который записывается автоматически. Поскольку машинное представление строки имеет последний нулевой байт, то говорят, что строки C оканчиваются нулем.

Для задания управляющих и разделительных символов внутри строковых констант используются символьные кодовые и числовые константы без апострофов.

"Y\a" эквивалентно "Y\07", эквивалентно "Y\x7", эквивалентно Y со "звуком";

"\nNew string" эквивалентно "\012\015New string"

"\"PATRIOT\"" эквивалентно "PATRIOT".

Символические константы базовых типов объявляются так же, как и переменные соответствующих типов, но с указанием модификатора *const* (см. подраздел 2.4). Например:

```
const  cnt      x = 5; y = 10;
const  double   z = 1E-5;
const  float    PI = 3.141593;
```

```
const char   Smb = 't';
const char   SC[] = "String constant";
```

Символические константы можно задавать также средствами препроцессора.

```
#define HI "Hello, dear"     //определение символической строковой константы
#define PI 3.141593          //определение символической числовой константы PI
```

Далее в тексте программы можно использовать только имя PI; препроцессор перед компиляцией будет везде его заменять десятичной числовой константой 3.14. Именно так определены математические и другие числовые константы в файлах *limits.h, values.h, math.h.* Рекомендуется имя константы задавать с помощью заглавных букв, указывая тем самым программисту на константу.

Другой способ задания *символических целочисленных констант* — использование типа ***enum***.

Например, в результате следующего объявления

```
enum error {OK, NO_OPEN, NO_CREATE, FAIL = 5};
```

мы получим константы с указанными в списке именами и со значениями 0, 1, 2 и 5 соответственно.

2.6. Время существования и область видимости переменных

Каждая переменная, объявленная в программе, имеет две важнейших характеристики:

- время существования;
- область видимости.

Эти характеристики взаимосвязаны и существенно влияют на возможности использования переменной в программе. Взаимосвязь характеристик определяется способом выделения памяти для переменной.

Время существования, или время жизни переменной, измеряется в следующих двух относительных единицах.

1. *Локальное* время жизни — это время существования переменной при выполнении блока, в котором она объявлена.

2. *Глобальное* время жизни — это время существования переменной при выполнении всей программы.

Область видимости, или область действия (доступности) переменной, также измеряется в двух относительных единицах.

1. *До конца блока*, в котором объявлена переменная.

2. *До конца файла*, в котором объявлена переменная.

Управлять этими характеристиками переменных программист может двумя путями:

1) изменением места объявления переменной в программе.

2) использованием модификаторов ***auto, register, static, extern***.

Автоматическая (*auto*) переменная или константа имеет *локальную* область действия и известна только внутри блока, в котором она определена. Для автоматичес-

кой переменной выделяется временная память. Память выделяется при входе в блок, а при выходе из блока память, выделенная для переменной, считается свободной, т. е. переменная уничтожается. Если спецификатор класса памяти не указан, то переменная в блоке по умолчанию считается автоматической.

Регистровая *(register)* переменная отличается от автоматической только памятью, которая выделяется для ее хранения. Регистровая переменная хранится в регистре процессора, и, соответственно, доступ к этой переменной гораздо быстрее, чем к той, которая хранится в оперативной памяти (*auto*). В случае отсутствия свободных регистров регистровая переменная становится автоматической.

Внешняя *(extern)* переменная является глобальной переменной. Спецификатор *extern* информирует компилятор, что переменная будет объявлена (без *extern*) в другом файле, где ей и будет выделена память.

Статической *(static)* переменной (константе) выделяется память после ее объявления и сохраняется до конца выполнения программы. Статические переменные при объявлении по умолчанию инициализируются нулевыми (логические, целые и вещественные) или пустыми значениями.

В табл. 2.3 приведены характеристики переменных, объявленных с использованием модификаторов выделения памяти.

Пример 1. Использование локальных и глобальных переменных.

```
#include <iostream.h>
int globvar1 = 0;          // глобальная переменная файла
static int globvar2 = 1;   // глобальная переменная файла
int main(void)
  {
    extern int globvar1;   //инициализировать при объявлении нельзя
    globvar1 = globvar2;
    static int var2 = 2;
    int *pstatvar = NULL;
    register int regvar3 = 3;
    int var4 = 4;
      {
        int var5 = 5;
        static int var6 = 6;
        pstatvar = &var6;
        register int regvar7 = 7;
        cout << globvar1 << var2 << regvar3 << var4
             << var5     << var6 << regvar7 <<"\n";
      }
    cout << globvar1  << var2 << regvar3 << var4
         << *pstatvar <<"\n";
    // переменные var5 и var7 здесь не существуют,
    // а переменная var6 непосредственно не доступна.

    return 0;
  }
```

Таблица 2.3

Характеристики переменных

Модификатор выделения памяти	Место объявления переменной	Область видимости	Время существования	Обобщенная характеристика
не указан (по умолчанию STATIC)	вне функции	до конца файла	глобальное	глобальная переменная файла
не указан (по умолчанию AUTO)	в функции	до конца блока	локальное	локальная переменная
AUTO	запрещено вне функции			
AUTO	в функции	до конца блока	локальное	локальная переменная
REGISTER	запрещено вне функции			
REGISTER	в функции	до конца блока	локальное	быстрая локальная переменная
STATIC	вне функции	до конца файла	глобальное	глобальная переменная файла
STATIC	в функции	до конца блока	глобальное	сохраняемая локальная переменная
EXTERN	вне функции	до конца файла	глобальное	глобальная переменная программы
EXTERN	в функции	до конца файла	глобальное	специальная глобальная переменная

В приведенной программе есть переменные, которые имеют глобальное время существования и глобальную область видимости (*globvar1, globvar2*). В ней также имеются переменные с глобальным временем существования и локальной областью видимости (*var2, var6*). Остальные переменные имеют локальное время существования и локальные области видимости, причем различные. Во внутреннем блоке существуют переменные *var5, regvar7*, а в блоке функции *main()*, включая вложенный блок, существуют переменные *pstatvar, regvar3* и *var4*.

Контрольные вопросы и задания

1. Какие символы входят в состав алфавити языка C++?
2. Дайте определение идентификатора.
3. Каково назначение ключевых слов и ограничения по их использованию?
4. Назовите рекомендации по выбору идентификаторов.
5. Поясните смысл понятия оператор.
6. Что понимается под типом данных?
7. Какая информация сообщается компилятору при объявлении переменных и констант?
8. Перечислите производные типы данных.
9. Каким образом различаются переменные по времени жизни?
10. Укажите назначение модификаторов выделения памяти.
11. Каково назначение идентификаторов?
12. Дайте определение выражению.
13. Укажите правила вычисления выражений.
14. Приведите примеры операций с одинаковым приоритетом.
15. Укажите операции с наивысшим и наименьшим приоритетом.
16. Перечислите ключевые слова, используемые при объявлении простых типов данных.
17. Назовите тип данных, обеспечивающий наибольший диапазон представления чисел.

Укажите, как влияет размер операндов арифметической операции на тип результата этой операции.

18. Для заданных наборов символов укажите, какие из них являются идентификаторами:
    а) MS, #K025, PC_67, ПУСК, ik=5, karta, 8E15;
    б) Слон, alfa, X–2, DR1a, N&88, _r13, DSO.4;
    в) BEPA, GOL, Upor, 0123, MAXMIN, I&I, PQ–17;
    г) IF(1), POTA_3, DR1A, DOS_4.1, hr8;
    д) СИНУС, X*X*X, veter3, 2NOO, Le;
    е) Тип, Sirius_1, KOCA, C++, OO5, PC 67;
19. В программе имеются следующие объявления переменных:
int x, y,i; long a,b; float d,c; unsigned char f,k; double n,m;.
Укажите тип результата для следующих выражений:
    а) f/(i+a*k) ;
    б) x–c/k*n ;
    в) y*f–m/i ;
    г) (i+c)/(a–k).
20. Заданы следующие арифметические выражения:
    а) x+a*b–2/c+4 ;
    б) z–y/(8+c++) ;
    в) m*((n–4)/(c+2))/k ;
    г) y–+5*z/x+n.

С помощью круглых скобок укажите порядок выполнения операций.

3. РАЗВЕТВЛЕНИЯ И ЦИКЛЫ

3.1. Программирование разветвлений

Для программирования разветвлений в языке С предназначены условный оператор (*if ... else*), операция условия (*?:*), оператор переключатель (*switch*).

Условный оператор

Условный оператор обеспечивает выполнение или невыполнение некоторого оператора или группы операторов в зависимости от заданного условия. Оператор *if* является одним из самых популярных средств, изменяющих естественный порядок выполнения операторов программы. Он может использоваться в одной из следующих форм:

```
if (условное выражение) оператор1
if (условное выражение) оператор1
else оператор2
```

Если значение условного выражения истинно (отлично от нуля), то выполняется *оператор1;* если ложно (равно нулю), то для первой формы *оператор1* пропускается, а для второй формы после пропуска *оператора1* выполняется *оператор2*, стоящий после слова *else*. Иногда после проверки условия необходимо выполнить более чем один оператор, тогда требуемая для выполнения после *if* часть программы заключается в блок с помощью фигурных скобок {}.

Пример 1. Найти минимум из двух чисел *x* и *y*.

```
. . .
if(x<y) min=x;
else min= y;
cout<<"min="<<min;
. . .
```

Пример 2. Проверка правильности ввода переменной, которая может содержать числа от 1 до 31.

```
. . .
cin>>den;
if(den<1||den>31) cout<<"Ошибка ввода";
. . .
```

Операторы *if* могут быть вложены друг в друга.

Пример 3. Отыскание максимума из трех чисел *a*, *b*, *c*.

```
. . .
if(a>b&&a>c) max=a;
else if(b>c) max=b;
else max=c;
cout<<"max="<<max;
. . .
```

Операция условия ? :

В языке C++ имеется короткий способ записи оператора *if ... else*. Для этого используют ***операцию условия***. Она имеет следующую форму записи:

```
(условное выражение) ? выражение1 : выражение2
```

Если условное выражение истинно, то выполняется *выражение1*, если ложно — *выражение2*.

Пример 4. Найти максимум из двух чисел *x* и *y*.

```
. . .
max=(x>y)?x:y;
cout<<"max="<<max;
. . .
```

Операцию условия удобно использовать в случаях выбора значения из двух возможных. Применение этой операции не является обязательным, так как тех же результатов можно достичь при помощи оператора *if ... else*. Однако получаемые при использовании операции условия выражения более компактны и их применение приводит к получению более компактного машинного кода.

Множественный выбор: операторы *switch* и *break*

Рассмотренные выше операция условия и условный оператор *if ... else* позволяют легко осуществить выбор между двумя вариантами. Однако иногда возникает необходимость осуществить выбор одного варианта из нескольких. Это можно сделать, используя вложенные условные операторы *if*. При этом сама программа становится более сложной, а главное — менее доступной для понимания и анализа.

Удобным средством для осуществления выбора из множества вариантов является оператор *switch*, который имеет следующую форму записи:

```
switch ( выражение )
  {
    case константа1: оператор1 break;
    . . .
    case константа N: операторN break;
    default: оператор break;
  }
```

Оператор выбора работает следующим образом. Сначала вычисляется выражение, стоящее в скобках после слова *switch*. Затем осуществляется переход на одну из меток, обозначенную словом *case*, значение константы, после которой совпало со значением выражения в скобках после *switch*. Константа, стоящая после *case*, должна быть целого типа. Если проверяемое выражение не совпало ни с одной из проверяемых констант, то осуществляется переход на метку *default* (ее использование не является обязательным).

Опытные программисты для поиска ошибок часто включают *default*, даже когда учтены все возможные случаи.

Обычно действие каждой ветви заканчивается оператором *break*. Выполнение этого оператора приводит к выходу из оператора *switch*. Если *break* отсутствует, то управление передается следующему оператору, помеченному *case* или *default*. Подобным образом выполняются все последующие операторы внутри *switch*, пока не встретится оператор *break*.

Ключевые слова *case* и *default* не могут находиться за пределами блока *switch*.

Пример 5. Применение оператора *switch*.

Требуется проанализировать значение переменной *rez*, которая является выставленной оценкой.

```
. . .
switch (rez)
  {
    case 5: cout<<"Оценка – отлично."; break;
    case 4: cout<<"Оценка – хорошо."; break;
    case 3: cout<<"Оценка – удовлетворительно."; break;
    case 2: cout<<"Оценка – неудовлетворительно."; break;
    default: cout<<"Неверное значение rez.";
  }
. . .
```

Как видно из примера, анализируемая переменная *rez* помещается после *switch*. Ее значение сравнивается поочередно со значениями, стоящими после *case*. Если совпадение произошло, то выполняется соответствующая ветвь. Если не выявлено ни одного совпадения, выдается сообщение об ошибке.

3.2. Типы операторов циклов

При выполнении программы нередко возникает необходимость неоднократного повторения однотипных вычислений над различными данными. Для этих целей используются так называемые циклы.

Цикл представляет собой участок программы, в котором одни и те же вычисления реализуются неоднократно над различными значениями одних и тех же переменных (объектов).

Для организации циклов в C++ используются следующие три оператора: *while*, *for* и *do – while*.

Цикл типа *while*

Цикл типа *while* является циклом с предусловием. Он используется в случае, когда, во-первых, не известно точное число повторов и, во-вторых, при этом нет необходимости, чтобы цикл непременно был выполнен хотя бы один раз. Цикл типа *while* имеет следующую форму записи:

```
while ( выражение ) оператор
```

В качестве выражения обычно используются условные выражения. В общем случае допускается использование выражений произвольного типа. На месте оператора, который является телом цикла, может стоять простой оператор или совокупность операторов, объединенных в блок скобками {}.

Если выражение истинно (не равно нулю), то тело цикла выполняется один раз, затем выражение проверяется заново. Итерации (проверка условия и тело цикла) выполняются до тех пор, пока выражение не станет ложным (равным нулю).

При организации цикла типа *while* в его тело должны быть включены конструкции, изменяющие логику проверяемого выражения так, чтобы в конце концов оно стало ложным. В противном случае выполнение цикла никогда не закончится.

Пример 1. Цикл типа *while*.

Пользователю дается 10 попыток для угадывания заданного в программе числа.

```
. . .
i=1; rez=1;
while ( i++<=10&&rez!=25)
  {
    cout<<"\nВведите число:";
    cin>>rez;
  }
if (i==12)cout<<"\nВы не угадали.";
else cout<<"\nПоздравляю! Вы угадали число.";
. . .
```

В данном примере цикл выполняется до тех пор, пока не угадано число или не исчерпано количество попыток.

Цикл типа *for*

Цикл типа *for* является циклом с параметрами и обычно используется в случае, когда известно точное количество повторов вычислений. При этом осуществляются три операции: инициализация счетчика циклов, сравнение его величины с некоторым граничным значением и изменение значения счетчика при каждом прохождении тела цикла.

В принципе эти действия можно реализовать с помощью цикла типа *while*. С помощью условного выражения можно произвести сравнение, приращение осуществить в теле цикла, а инициализацию счетчика выполнить за пределами цикла перед его выполнением. При таком подходе есть опасность не проинициализировать счетчик или не выполнить его приращение. Это ведет к повышению вероятности возникновения ошибок и затрудняет отладку.

В цикле *for* данные три действия собраны вместе, что позволяет избежать указанных неприятностей. Цикл *for* имеет следующую форму записи:

```
for ( выражение1; выражение2; выражение3 ) оператор
```

Выражение1 вычисляется первым. Обычно здесь производится инициализация счетчика циклов и переменных. Это выражение вычисляется один раз, когда цикл *for* начинает выполняться. Затем вычисляется *выражение2*. Оно предназначено для проверки условия. Если значение *выражения2* отлично от нуля, то выполняется *оператор* (тело цикла). Если же значение *выражения2* равно нулю, то цикл завершается. *Выражение3* вычисляется в конце каждого выполнения тела цикла.

В качестве оператора может использоваться простой оператор или совокупность операторов, заключенных в блок с использованием скобок {}.

В круглых скобках после слова *for* могут отсутствовать все выражения или любое из них, но должны обязательно быть две точки с запятой. Специальное правило для цикла *for* гласит, что если отсутствует *выражение2*, то результат проверки всегда истина.

Пример 2. Цикл типа *for*.

Пусть требуется вычислить y^{10}. Возможный вариант решения имеет вид:

```
. . .
for ( i=1, rez=1; i<=10; i++ ) rez=rez*y;
cout<<"rez="<<rez;
. . .
```

Цикл типа *do — while*

Цикл типа *do — while* является циклом с постусловием и используется в тех случаях, когда неизвестно точное количество повторов, но в то же время цикл необходимо выполнить по меньшей мере один раз. Цикл типа *do — while* очень похож на цикл типа *while*; разница состоит только в том, что проверка истинности выражения в цикле *do — while* происходит после выполнения тела цикла. Этот цикл имеет следующую форму записи:

```
do оператор while ( выражение )
```

Пример 3. Цикл типа *do — while*.

Требуется составить программу, позволяющую пользователю угадывать заданное в программе число. Возможный вариант циклической части программы имеет следующий вид:

```
. . .
do cin>>r;
while ( r!=13);
cout<<"Вы успешно угадали число."
. . .
```

В данном примере пользователь вводит числа до тех пор, пока не будет введено число 13. После этого выдается сообщение об успешном угадывании числа.

3.3. Вложенные циклы

Вложенным циклом называют конструкцию, в которой один цикл выполняется внутри другого. Внутренний цикл выполняется полностью во время каждой итерации внешнего цикла.

Пример 1. Вложенные циклы.

Требуется заполнить весь экран символами '#'. Возможный вариант решения имеет вид:

```
. . .
for ( i=1; i<=25; i++ )
  for (k=1; k<=80; k++ ) cout<<'#';
. . .
```

В данной программе 25 раз осуществляется вывод по 80 символов.

В программе можно использовать любые комбинации вложенных циклов всех типов: *while, for, do — while*, если этого требует логика построения программы.

Пример 2. Осуществить ввод десяти значений дней месяца с проверкой правильности ввода.

```
. . .
for ( i=1; i<=10; i++ )
  {
    do cin>>den;
    while (den<1||den>31);
    cout<<den;
  }
. . .
```

В данном примере внешний цикл выполняется 10 раз, а внутренний будет выполняться до тех пор, пока не будет введено правильное значение.

3.4. Рекомендации по выбору циклов

После того как выяснилось, что для решения задачи необходимо использовать оператор цикла, возникает вопрос: циклом какого типа лучше всего воспользоваться?

Сначала надо решить, нужен ли цикл с предусловием или с постусловием. Чаще всего возникает необходимость в цикле с предусловием. По оценкам специалистов, в среднем циклы с постусловием (*do — while*) составляют 5% от общего числа используемых циклов.

Существует ряд причин, по которым профессиональные программисты предпочитают пользоваться циклами с предусловием. Первая — это то, что лучше сначала принять решение о том, надо ли что-то делать, а не после того, когда это уже сделано. Вторая — то, что программа более понятна, когда проверяемое условие находится в начале, а не в конце циклического участка. Третья причина состоит в том, что в большинстве случаев важно, чтобы тело цикла игнорировалось полностью, если условие не выполняется.

В случае выбора цикла с предусловием возникает второй вопрос: что лучше, *for* или *while*? В принципе все, что можно сделать с помощью одного цикла, можно сде-

лать и с помощью другого. Исходя из соображений правильного стиля программирования применение цикла *for* представляется более предпочтительным, когда в цикле используется инициализация и коррекция переменной. А цикл *while* удобнее применять, когда этого делать не требуется.

В языке C++ оператор цикла *for* является более гибким средством, чем аналогичные операторы циклов в других языках программирования, например, Паскале, Бэйсике, Фортране и PL/1. Эта гибкость является следствием использования трех выражений в скобках после *for*. Кроме описанных выше, существует еще много других возможностей применения этого типа циклов. Рассмотрим некоторые из них.

1. Можно применять операцию уменьшения для счета в порядке убывания.

Пример 1. Счет в порядке убывания.

Требуется вычислить y^5. Возможное решение имеет вид:

```
. . .
for ( i=5, r=1; i>=1; i- ) r=r*y;
  cout<<"r="<<r;
. . .
```

2. При желании можно организовать счет двойками, тройками, десятками и т. д.

Пример 2. Приращение при счете, отличное от 1.

```
. . .
for ( n=5; n<61; n+=15) cout<<n;
. . .
```

3. Можно в качестве счетчика использовать не только цифры, но и символы.

Пример 3. Использование символов в качестве счетчика.

Требуется напечатать алфавит. Возможное решение имеет вид:

```
. . .
for ( chr='A'; chr<='Z'; chr++) cout<<chr;
. . .
```

4. Можно проверять любое другое условие, отличное от числа итераций.

5. Можно задать возрастание значений счетчика не в арифметической, а в геометрической прогрессии.

Пример 4. Изменение счетчика в геометрической прогрессии.

Требуется подсчитать долг. Возможное решение имеет вид:

```
. . .
for ( k=100; k<185; k*=1.1) cout<<"Долг="<<k;
. . .
```

6. В качестве третьего выражения можно использовать любое правильно составленное выражение. Оно будет вычисляться в конце каждой итерации.

Пример 5. Использование в качестве счетчика выражения.

```
. . .
for ( k=1; z<=196; z=5*k+23 ) cout<<z;
. . .
```

7. Можно пропускать одно или несколько выражений. (При этом нельзя пропускать символы «точка с запятой».)

Пример 6. Неполный список выражений в заголовке тела цикла.

```
. . .
for (p=2; p<=202;) p=p+n/k;
. . .
```

8. Первое выражение не обязательно должно инициализировать переменные, оно может быть любого типа.

Пример 7. Произвольное первое выражение в заголовке цикла.

```
. . .
for ( cout<<"Вводите числа."; p<=30;) cin>>p;
. . .
```

9. Переменные, входящие в выражения спецификации цикла, можно изменять в теле цикла.

Пример 8. Изменение управляющих переменных в теле цикла.

```
. . .
delta=0.1;
for (k=1; k<500; k+=delta)      if (a>b)    delta=0.5;
. . .
```

10. Использование операции «запятая» в спецификации цикла позволяет включать несколько инициализирующих и корректирующих выражений.

Пример 9. Использование операции «запятая» в спецификации цикла.

```
. . .
for ( i=1, r=1; i<=10; i++, r*=y )
cout<<"y в степени "<<i<<" равен:"<<r;
. . .
```

Большая свобода выбора вида выражений, управляющих работой цикла *for*, позволяет с помощью этого оператора делать гораздо больше дополнительных действий, чем выполнять просто фиксированное число итераций.

3.5. Управляющие операторы в циклах

Условные операторы и операторы циклов являются важнейшими средствами управления порядком выполнения программы на языке C++. Существует еще три оператора, предназначенных для этих же целей. Они применяются реже, поскольку частое их использование ухудшает наглядность программы и увеличивает вероятность ошибок.

Оператор ***break*** является наиболее важным из этих трех операторов и уже встречался при рассмотрении оператора выбора *switch*. Оператор *break* может использоваться в циклах всех трех типов. Выполнение оператора *break* приводит к выходу из цикла, в котором он содержится, и переходу к следующему за циклом оператору. Если

оператор *break* находится внутри вложенных циклов, то его действие распространяется только на тот цикл, непосредственно в котором он находится.

Пример 1. Использование оператора *break*.

Требуется определить задуманное число с 10 попыток.

```
. . .
i=1;
while ( i++<=10 )
  {
    cin>>rez;
    if ( rez==15 ) break;
    cout<<"\nПопытка неудачная.";
  }
if ( i!=12 ) cout<<"\nВы угадали число.";
. . .
```

В этом примере при угадывании числа происходит прекращение выполнения цикла с помощью оператора *break*.

Оператор ***continue*** может использоваться только среди операторов тела цикла. Этот оператор вызывает пропуск оставшейся части итерации внутри цикла и переход к следующей итерации.

Пример 2. Использование оператора *continue*.

Вводятся числа месяца для обработки. Необходимо осуществить проверку правильности ввода. Число 31 обозначает конец обработки.

```
. . .
while ( den!=31)
  {
    cin>>den;
    if ( den<1||den>31 ) continue;
. . .                        // Обработка числа den
  }
. . .
```

В данном примере неправильный ввод значения приводит к пропуску части итерации, предназначенной для обработки этого значения.

Заметим, что такое изменение условия *if(den<1||den>31)* на обратное *if(den>0&&den<32)* позволяет исключить использование *continue*. С другой стороны, с помощью *continue* иногда можно сократить некоторые программы, особенно если они включают в себя вложенные операторы *if ... else*.

Оператор ***goto*** (переход на заданную метку) в языке C++ является плохим средством. Его использование приводит к значительному усложнению логики программы и идет вразрез с правильным стилем программирования. В C++ без использования этого оператора структура программы всегда оказывается лучше, чем в других языках программирования.

Существует лишь один случай, когда программисты профессионалы допускают использование *goto*, — это выход из вложенного набора циклов при обнаружении ошибок (*break* дает возможность выхода лишь из одного цикла).

Контрольные вопросы и задания

1. Что такое разветвляющийся вычислительный процесс?
2. Какие формы записи имеет условный оператор *if*?
3. Назовите отличительные особенности операции условия в сравнении с условным оператором.
4. Для решения каких задач удобно применять оператор *switch*?
5. Что такое циклический вычислительный процесс?
6. Какие операторы используются для организации циклов? Для решения каких задач каждый из них удобно использовать?
7. Какие три операции должны производиться в программе при организации любого цикла?
8. Охарактеризуйте преимущества, которые дает цикл *for*.
9. Укажите назначение управляющих операторов, позволяющие изменять порядок работы цикла.
10. Составьте программу, вычисляющую

$$z=\begin{cases} y, \text{ если } y>0; \\ y^2, \text{если } y\le 0. \end{cases}$$

11. Составьте программу, печатающую по номеру дня недели (число от 1 до 7) название этого дня (понедельник, вторник и т. д.).
12. Составьте программу, вычисляющую $y=x!$.
13. Составьте программу вычисления суммы

$$S=\sum_{i=1}^{\infty} a_i, \text{ суммируя } |a_i|\ge 10^4, \text{ где } a_i=\frac{x^i}{i!}.$$

4. МАССИВЫ И УКАЗАТЕЛИ

4.1. Массивы

Массивы позволяют удобным образом организовать размещение и обработку больших объемов информации. ***Массив*** представляет собой набор однотипных объектов, имеющих общее имя и различающихся местоположением в этом наборе (или индексом, присвоенным каждому элементу массива). Элементы массива занимают один непрерывный участок памяти компьютера и располагаются последовательно друг за другом.

Пример 1. Описания массивов.

```
int mas1[492];              // внешний массив из 492 элементов
void main(void)
  {
    double mas2[250];       // массив из 250 чисел типа double
    static char mas3[20];   // статическая строка из 20 символов
    extern mas1[];          // внешний массив, размер указан выше
    int mas4[2][4];         // двумерный массив из чисел типа int
  }
```

В данном примере квадратные скобки [ ] обозначают, что все идентификаторы, после которых они следуют, являются именами массивов. Число, заключенное в скобки, определяет количество элементов массива. Доступ к отдельному элементу массива организуется с использованием номера этого элемента, или индекса. Нумерация элементов массива начинается с нуля и заканчивается $n-1$, где n — число элементов массива.

Пример 2. Описание массива и присваивание начальных значений его элементам.

```
. . .
int mas[2];                 //объявление массива
int a=10,b=5;               //объявление переменных
. . .
mas[0]=a;
mas[1]=b;
. . .
```

4.2. Инициализация массивов

Инициализация массива означает присвоение начальных значений его элементам при объявлении. Массивы можно инициализировать списком значений или выражений, отделенных запятой, заключенным в квадратные скобки.

Пример 1. Инициализация массива, элементы которого содержат количество дней в каждом месяце года:

```
. . .
int days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
. . .
```

Если список инициализируемых значений короче длины массива, то инициализации подвергаются первые элементы массива, а остальные инициализируются нулем.

Массив также можно инициализировать списком без указания в скобках длины массива. При этом массиву присваивается длина по количеству инициализаторов.

Пример 2. Определение длины массива при инициализации.

```
. . .
char code[] = {'a', 'b','c'};
. . .
```

В данном примере массив *code* будет иметь длину 3.

Если же массив явно не проинициализирован, то внешние и статические массивы инициализируются нулями. Автоматические же массивы после объявления ничем не инициализируются и содержат неизвестную информацию.

4.3. Применение указателей

Указатель — это символическое представление адреса. Он используется для косвенной адресации переменных и объектов. Указатели тесным образом связаны с обработкой строк и массивов.

В C++ имеется операция определения адреса — &, с помощью которой определяется адрес ячейки памяти, содержащей заданную переменную. Например, если *vr* — имя переменной, то *&vr* — адрес этой переменной. В данном случае запись *&vr* означает: «указатель на переменную *vr*». Указатель на переменную — это число, которое представляет собой содержимое регистров CS:IP процессора. Символическое представление адреса *&vr* является константой типа указатель.

В C++ также существуют и переменные типа указатель. Например, значением переменной типа *char* является целое число длиной 1 байт, переменной типа *int* — целое число, а значением переменной типа указатель служит адрес переменной или объекта. Пусть переменная типа указатель имеет имя *ptr*, тогда в качестве значения ей можно присвоить адрес с помощью следующего оператора:

```
ptr=&vr;
```

В языке C++ при работе с указателями важное значение имеет операция ***косвенной адресации*** – *. Операция * позволяет обратиться к переменной не напрямую, а через указатель, содержащий адрес этой переменной. Данная операция является одноместной и имеет ассоциативность слева направо. Эту операцию не следует путать с бинарной операцией умножения. Пусть *ptr* – указатель, тогда **ptr* – это значение переменной, на которую указывает *ptr*.

Описание переменных типа указатель выполняется с помощью операторов следующей формы:

```
<тип> *<имя указателя на переменную заданного типа>;
```

Пример 1. Описание указателей.

```
int *ptri;                  //указатель на переменную целого типа
char *ptrc;                 //указатель на переменную символьного типа
float *ptrf;                //указатель на переменную с плавающей точкой
```

Такой способ объявления указателей возник вследствие того, что переменные разных типов занимают различное число ячеек памяти. При этом для некоторых операций с указателями требуется знать объем отведенной памяти. Операция * в некотором смысле является обратной операции &.

Пример 2. Использование указателей.

```
int a=10, b;
int *ptr=&a;                //инициализация указателя адресом переменной a
. . .
cout<<"указатель = "<<ptr<<"\nПеременная ="<<*ptr;
ptr=&b;                     //теперь ptr указывает на переменную b
. . .
```

Отметим, что без указателей невозможна обработка объектов. Например, передать объект некоторой функции можно только через указатель.

4.4. Ссылки

Ссылка (*reference*) представляет собой видоизмененную форму указателя, которая используется в качестве псевдонима (другого имени) переменной. В связи с этим для ссылок не требуется дополнительной памяти. Для определения ссылки используется символ & (амперсэнд), указываемый перед переменной-ссылкой.

Переменные ссылочного типа могут использоваться в следующих целях:

- вместо передачи в функцию объекта по значению;
- для определения конструктора копии;
- для перегрузки унарных операций;
- для предотвращения неявного вызова конструктора копии при передаче в функцию по значению объекта определенного пользователем класса.

Пример 1. Использование ссылок.

```
#include <iostream.h>                          // cout
int main()
  {
    int
      t = 13,
      &r = t;                                  // инициализация ссылки на t
                                               // теперь r синоним имени t
    cout << "Начальное значение t:" << t;      // выводит 13
    r += 10;                 // изменение значения t через ссылку
    cout << "\nКонечное значение t:" << t;     // выводит 23
    return 0;
  }
```

В данном случае мы использовали ссылку в качестве псевдонима переменной. В этой ситуации она называется *независимой ссылкой (independent reference)* и должна быть инициализирована при объявлении. Такой способ использования ссылок может привести к фатальным и трудно обнаруживаемым ошибкам по причине возникновения путаницы в использовании переменных.

Другое применение ссылок — возможность создания параметров функции, передаваемых по ссылке, при этом перед этим параметром ставится знак &.

Пример 2. Функция с параметром-ссылкой.

```
#include <iostream.h>                          // cout
void sqr(int &);                               // прототип функции
int main()
  {
    int t = 3;
    cout << "Начальное значение t:" <<  t;     // выводит 3
    sqr(x);
    cout << "\nКонечное значение t:" << t;     // выводит 9
    return 0;
  }
void sqr (int &x)
  {
    x *= x;
  }
```

В приведенной программе при объявлении параметра-ссылки мы указываем, что в функцию будет передаваться адрес переменной. Это дает возможность модифицировать фактические параметры, не указывая в явном виде при вызове функции передачу в нее адреса. Другой особенностью является то, что внутри функции отпадает необходимость в использовании операции разыменования * (звездочка) при обращении к параметрам. Однако для избежания ошибок при использовании параметров ссылочного типа целесообразно соблюдать следующее правило: аргументы-ссылки всегда должны передаваться в функцию как константы (*const*). Это позволяет обезопаситься от многих неприятностей.

Рассмотрим пример объявления двух функций, у которых параметр *x* является выходным:

```
int func1(int &x);
int func2(int *x);
```

Вызовы этих функций будут выглядеть соответственно так:

```
func1(param);
func2(&param);
```

Вызов *func1()* не указывает на то, что параметр *param* будет изменен при выходе из этой функции, в отличие от вызова *func2()*, где операция & (получения адреса) указывает на то, что в функцию передается адрес переменной для изменения ее значения. Поэтому если вы используете параметры ссылочного типа, то не модифицируйте их внутри функции и, для подстраховки, применяйте к ним модификатор доступа *const*, а выходные параметры должны использовать указатели для хранения результатов.

Ссылки можно использовать в качестве возвращаемых значений. Функция, которая возвращает ссылку, может находиться в левой части оператора присваивания.

Пример 3. Использование ссылки в качестве возвращаемого значения.

```
#include <iostream.h>                  // cout
int &f(int i);                         // прототип функции
int a[3] = {3,4,5};                    // объявление и инициализация массива

int main()
  {
    f(1) = 23;                         // присвоение: a[1] = 23
    for(int j = 0; j < 3; ++j)         // вывод результатов
      cout << "\na[" << j << "]= " << a[j];
    return 0;
  }

int &f(int i)
  {
    return a[i];
  }
```

Эта программа заменяет значение первого элемента массива (нумерация с нуля) на число 23. Функция *f()* возвращает ссылку на элемент массива *a*, определяемый аргументом *i*. Далее ссылка используется для присвоения элементу массива значения 23. Отметим, что нельзя возвращать ссылку (указатель) на локальную переменную, так как эта переменная перестает существовать после этого возврата. Пример ошибки такого рода приведен ниже:

```
// этот код не работает!
int &func()                            // функция, возвращающая ссылку
{
```

```
    int x;
    // . . .
    return x;
}
```

На применение переменных ссылочного типа накладывается ряд ограничений:

- нельзя взять адрес переменной ссылочного типа;
- запрещается использовать массивы ссылок;
- не допускается использовать ссылки на битовые поля;
- нельзя создать указатель на ссылку.

4.5. Указатели и массивы

Поскольку указатели представляют собой символические адреса объектов, то это дает возможность программисту применять адреса подобно тому, как это делается в компьютере, и тем самым повысить эффективность программы.

Пусть *mas[6]* — массив из шести элементов, тогда записи *mas* и *&mas[0]* эквивалентны и определяют адрес 1-го элемента массива. Оба значения являются константами типа указатель, поскольку они не изменяются на протяжении всей работы программы. Однако эти значения можно присваивать переменным типа указатель. Например:

```
  . . .
int a[4], *ptra , i;
float b[4], *ptrb;
  . . .
ptra=a;                    //присваивает указателю адрес массива
ptrb=b;                    //аналогично
for(i=0; i<4; i++)
cout<<"указатель+"<<i<<": "<<(ptra+i)<<" "<<(ptrb+i)<<"\n";
  . . .
```

На экране монитора появится, например, следующий результат:

```
указатель+0: 0x2e2112b2 0x2e2112ee
указатель+1: 0x2e2112b4 0x2e2112f2
указатель+2: 0x2e2112b6 0x2e2112f6
указатель+3: 0x2e2112b8 0x2e2112fa
```

Возникает вопрос: почему *0x2e2112b2+1=0x2e2112b4* или *0x2e2112ee+1= 0x2e2112f2*? Такая ситуация возникает из-за того, что единицей адресации в первом случае является тип *int* (размер — 2 байта). Для типа *float* единица адресации имеет размер 4 байта. Прибавление единицы к адресу означает переход к следующему элементу массива.

При объявлении указателей необходимо указывать тип объекта, чтобы компилятору было известно, сколько байтов добавлять к адресу при увеличении ука-

зателя на единицу. Работа с указателями из рассмотренного примера графически изображена на рис. 4.1.

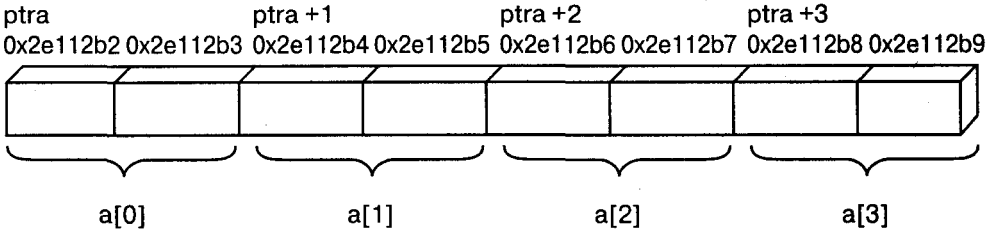


Рис. 4.1. Работа с указателями

Отметим, что прибавление к указателю 1 обеспечивает переход к следующему элементу массива, независимо от типа его элементов.

4.6. Указатели и многомерные массивы

Для начала рассмотрим пример объявления двумерного массива:

```
  . . .
int mas[4][2];
int *ptr;
  . . .
```

Тогда выражение *ptr=mas* указывает на первый столбец первой строки матрицы. Записи *mas* и *&mas[0][0]* равносильны. Выражение *ptr+1* указывает на *mas[0][1]*, далее идут элементы: *mas[1][0]*, *mas[1][1]*, *mas[2][0]* и т. д.; *ptr+5* указывает на *mas[2][1]*.

Двумерные массивы располагаются в памяти подобно одномерным массивам, занимая последовательные ячейки памяти (рис. 4.2).

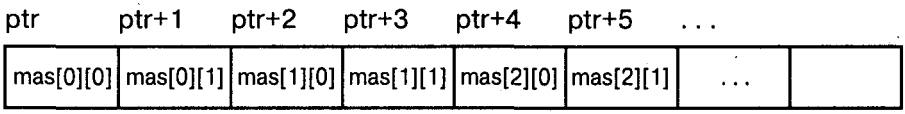


Рис. 4.2. Размещение двумерного массива в памяти

Элементы многомерного массива в общем случае располагаются на непрерывном участке памяти таким образом, что самый правый индекс изменяется первым.

4.7. Динамические массивы

Динамическим называется массив, размерность которого становится известной в процессе выполнения программы.

В C++ для работы с динамическими объектами применяются специальные операции *new* и *delete*. С помощью операции ***new*** выделяется память под динамический объект (создаваемый в процессе выполнения программы), а с помощью операции ***delete*** созданный объект удаляется из памяти.

Пример 1. Выделение памяти под динамический массив.

Пусть размерность динамического массива вводится с клавиатуры. Необходимо выделить память под этот массив, а затем созданный динамический массив надо удалить. Соответствующий фрагмент программы имеет вид:

```
  . . .
int n;
  . . .
cin>>n;                    // n – размерность массива
int *mas=new int[n];       // выделение памяти под массив
  . . .
delete mas;                // освобождение памяти
  . . .
```

В данном примере *mas* является указателем на массив из *n* элементов. Оператор *int *mas=new int[n];* производит два действия: *объявляется* переменная типа указатель, а затем *указателю присваивается адрес* выделенной области памяти в соответствии с заданным типом объекта. Для лучшего понимания работы с динамическими массивами эти две операции можно разделить.

Для рассмотренного примера можно задать следующую эквивалентную последовательность операторов:

```
. . .
int n, *mas;
. . .

cin>>n;                    // n – размерность массива
mas=new int[n];            // выделение памяти под массив
. . .
delete mas;                // освобождение памяти
. . .
```

Если с помощью операции *new* невозможно выделить требуемый объем памяти, то результатом операции *new* является значение 0.

Иногда при программировании возникает необходимость создания *многомерных динамических объектов*. Начинающие программисты по аналогии с описанным способом создания одномерных динамических массивов для двумерного динамического массива размерности n*k могут пытаться выделить память с помощью следующей записи операции *new*:

```
mas=new int[n][k]; // Неверно! Ошибка!
```

Такой способ выделения памяти не дает верного результата. Будет выдаваться ошибка о несоответствии типов указателей. Попытка устранить подобную ошибку с помощью операции явного преобразования типов приводит к ошибкам

при обращении к элементам массива из-за неверного типа выделенной области памяти.

Пример 2. Выделение памяти под двумерные массивы.

Пусть требуется создать двумерный динамический массив целых чисел размерностью *n*k*:

```
. . .
int n, k, i, * *mas;
. . .
cin>>n;                                  // n – число строк массива
cin>>k;                                  // k – число столбцов массива
mas=new * int[n];                        // выделение памяти под n
                                         // указателей на строку
for(i=0;i<n;i++) mas[i]=new int[k];      // выделение памяти для каждой
                                         // строки по числу столбцов k
. . .
for(i=0;i<n;i++) delete mas[i];          // освобождение памяти
delete [] mas;
. . .
```

Сначала необходимо с помощью операции *new* выделить память под *n* указателей. Выделенная память будет представлять собой вектор, элементом которого является указатель. При этом все указатели располагаются в памяти последовательно друг за другом. После этого необходимо в цикле каждому указателю присвоить адрес выделенной области памяти размером, равным второй границе массива (рис. 4.3).

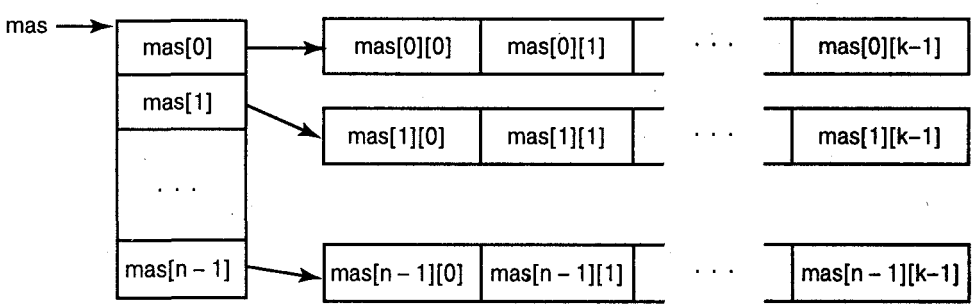


Рис. 4.3. Размещение динамического двумерного массива в памяти

Созданный таким образом двумерный динамический массив существенно отличается от обычного двумерного массива. Под обычный двумерный массив при объявлении выделяется сплошной участок памяти размером, равным произведению его границ. Для двумерного динамического массива выделенная память не представляет собой сплошной участок, поскольку она выделяется с помощью нескольких операций *new*.

В C++ определено понятие «вектор». Любой массив при объявлении представляет собой ***вектор***. Динамически память может быть выделена как вектор. Из этого следует невозможность явного выделения памяти под многомерный массив. Поэтому при работе с многомерными динамическими объектами следует пользоваться вышеописанным способом.

4.8. Пример использования указателей и массивов

Пусть необходимо упорядочить массив целых чисел по возрастанию. Размер массива задается пользователем во время выполнения программы.

В данном случае при упорядочивании массива возникает задача *сортировки*. Для решения подобной задачи существует много способов и разработана хорошая теоретическая база. Выберем один из способов сортировки. Для понимания этого способа рассмотрим в качестве примера алгоритм сортировки массива из 4-х целых чисел. Пусть заданы начальные значения элементов массива (рис. 4.4 а). Возьмем самый плохой вариант, когда числа упорядочены по убыванию.

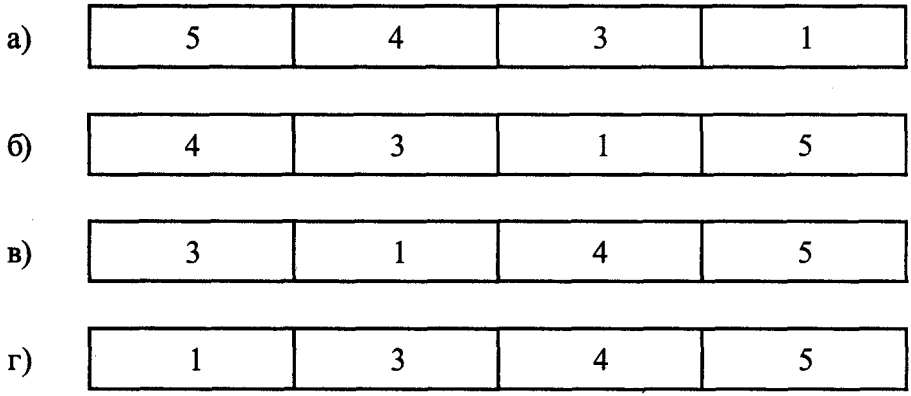


Рис. 4.4. Этапы сортировки массива из 4-х чисел

Проведем сравнение смежных элементов массива начиная с первого и до последнего и в случае неупорядоченности поменяем их местами. Очевидно, что для массива из 4-х чисел необходимо провести три сравнения. Результат показан на рис. 4.4 б. Ясно, что после 1-го прохода сравнений смежных элементов максимальный элемент переместится на последнее место в массиве. Поэтому привлекать данный элемент для последующих смежных сравнений не имеет смысла. Так как массив еще не упорядочен, необходимы дальнейшие сравнения смежных элементов с перестановками.

Для полного упорядочивания массива будем повторять проходы сравнений смежных элементов, начиная с 1-го элемента массива и до элемента, номер которого на 1 меньше последнего элемента, участвовавшего в сравнениях смежных элементов при предыдущем проходе.

Результаты сравнений смежных элементов с перестановками изображены на рис. 4.4 в и г. Очевидно, что для полного упорядочивания массива необходимо провести три прохода смежных сравнений.

Используем полученные результаты для общего случая сортировки. Пусть задан массив из n элементов. Тогда для его сортировки по возрастанию необходимо провести $n-1$ проходов сравнений смежных элементов. Для 1-го прохода должно быть произведено $n-1$ сравнений элементов, начиная с 1-го элемента. Для $(n-1)$-го прохода должно быть проведено одно сравнение.

Используем данный алгоритм сортировки для нашего примера. Законченная программа имеет следующий вид:

```
#include <iostream.h>
void main(void)
  {
    int n, i, j, *mas, buf;
    cout<<"\nВведите размерность массива:";
    cin>>n;
    mas=new int[n];
    cout<<"Вводите элементы массива.\n";
    for(i=0; i<n; i++)
      {
        cout<<"mas["<<i<<"]=";
        cin>>mas[i];
      }
    for(i=1; i<n; i++)
      for(j=1; j<n-i+1; j++)
        if(mas[j-1]>mas[j])
          {
            buf=mas[j-1]; mas[j-1]=mas[j]; mas[j]=buf;
          }
    cout<<"Результирующий массив:\n";
    for(i=0; i<n; i++) cout<<mas[i]<<" ";
    delete mas;
  }
```

В данной программе с помощью директивы препроцессора *#include* подключается стандартная библиотека потокового ввода-вывода. В разделе переменных объявлены следующие переменные: n — количество элементов массива, i и j — переменные счетчики циклов, *mas* — указатель на динамический массив, *buf* — переменная для временного хранения элементов массива.

В начале программы вводится число элементов массива. Далее с помощью операции *new* под массив выделяется требуемый участок памяти с учетом длины массива. Затем пользователем в цикле вводятся элементы массива. Далее с помощью 2-х вложенных циклов производится упорядочивание массива по возрастанию в соответствии с описанным алгоритмом. Во внутреннем цикле организовано сравнение смежных элементов массива, и в случае необходимости элементы массива меняются местами. Во внешнем цикле производится $n-1$ проходов сравнений смежных элементов массива.

Затем в цикле организован вывод результирующего массива. Далее с помощью операции *delete* из памяти удаляется созданный динамический массив. На этом работа программы заканчивается.

Контрольные вопросы и задания

1. Что такое массив? Дайте определение.
2. В чем состоят преимущества представления информации в виде массива?
3. Каким образом определяется длина массива при его инициализации?
4. Что такое указатель? Для чего он применяется?
5. Как объявляется переменная типа указатель?
6. Почему при объявлении указателей необходимо указывать тип адресуемой с его помощью переменной?
7. Раскройте понятие ссылки.
8. Для чего используются переменные ссылочного типа?
9. Перечислите основные сходства и различия ссылок и указателей.
10. В каких случаях нецелесообразно использовать переменные ссылочного типа?
11. Перечислите ограничения, накладываемые на ссылки.
12. Приведите пример инициализации указателя адресом переменной.
13. Каково назначение операции &?
14. Что такое динамический массив?
15. Каково назначение операций *new* и *delete*?
16. Опишите порядок создания динамического массива.
17. Как располагаются в памяти многомерные массивы?
18. Приведите пример инициализации матрицы целых чисел размерности 2×3.
19. Составьте программу нахождения суммы

$$z=\sum_{i=1}^{10} y_i$$, где $y_1, ..., y_{10}$ — массив из 10 чисел.

20. Составьте программу вычисления произведения

$$P = \prod_{j=1}^{n}(1+x_i)$$, где $x_1, ..., x_n$ — динамический массив из n элементов. Значение n определяется при вводе.

21. Составьте программу ввода для обработки массива координат. Размер массива передается программе на этапе ее выполнения. Одна координата — это три значения.

5. ФУНКЦИИ

5.1. Общие сведения о функциях

Функции используются для упрощения процесса разработки программ в случаях, когда аналогичные преобразования над различными данными необходимо выполнять в нескольких местах программы.

Каждая программа в своем составе должна иметь ***главную функцию*** *main()*. Именно функция *main()* обеспечивает создание точки входа в объектный модуль.

Кроме функции *main()*, в программу может входить произвольное число функций, выполнение которых инициализируется либо прямо, либо опосредованно вызовами из функции *main()*. Каждая функция по отношению к другой является внешней. Для того, чтобы функция была доступной, необходимо, чтобы до ее вызова о ней было известно компилятору.

С понятием функции в языке C++ связано три следующих компонента:

- описание функции;
- прототип;
- вызов функции.

Описание функции состоит из двух частей: заголовка и тела. Описание функции имеет следующую форму записи:

```
/* заголовок функции*/
[тип_результата] <имя> ([список_параметров])
  {
    /* объявления и операторы */
    тело_функции
  }
```

Здесь *тип_результата* — тип возвращаемого значения. В случае отсутствия спецификатора типа предполагается, что функция возвращает целое значение (*int*). Если функция не возвращает никакого значения, то на месте типа записывается спецификатор *void*. В списке параметров для каждого параметра должен быть указан тип. При отсутствии параметров список может быть пустым или иметь спецификатор *void*.

Тело функции представляет собой последовательность объявлений и операторов, описывающих определенный алгоритм. Важным оператором тела функции является оператор возврата в точку вызова: *return [выражение];*. Оператор *return* имеет двойное

назначение. Он обеспечивает немедленный возврат в вызывающую функцию и может использоваться для передачи вычисленного значения функции. В теле функции может быть несколько операторов *return*, но может не быть и ни одного. В последнем случае возврат в вызывающую программу происходит после выполнения последнего оператора тела функции.

Прототип функции может указываться до вызова функции вместо описания функции для того, чтобы компилятор мог выполнить проверку соответствия типов аргументов и параметров. Прототип функции по форме такой же, как и заголовок функции, в конце его ставится «;». Параметры функции в прототипе могут иметь имена, но компилятору они не нужны.

Компилятор использует прототип функции для сравнения типов аргументов с типами параметров. Язык C++ не предусматривает автоматического преобразования типов в случаях, когда аргументы не совпадают по типам с соответствующими им параметрами, т. е. язык C++ обеспечивает строгий контроль типов.

При наличии прототипа вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией.

Вызов функции может быть оформлен в виде оператора, если у функции *отсутствует* возвращаемое значение, или в виде выражения, если *существует* возвращаемое значение.

В первом случае оператор имеет следующий формат:

```
имя_функции (список_аргументов);
```

Во втором случае выражение записывается следующим образом:

```
имя_функции (список_аргументов)
```

Значение вычисленного выражения является возвращаемым значением функции. Возвращаемое значение передается в место вызова функции и является результатом ее работы.

Число и типы аргументов должны совпадать с числом и типом параметров функции. При вызове функции параметры подставляются вместо аргументов. Ниже приводится пример функции с возвращаемым значением.

Пример 1. Составить программу, содержащую обращение к функции вычисления максимума из двух чисел:

Возможное решение данной задачи имеет вид:

```
#include<iostream.h>
int max(int,int);          /* прототип функции */
void main()
  {
    int x,y,z;
    cout << "\n поочередно введите x и y \n";
    cin >> x; cin >> y;
    z=max(x,y);
    cout << "z=" << z;
  }
#include "d:\max.cpp"      // включение файла max.cpp с функцией max
```

Описание функции *max* находится в файле *max.cpp*, находящемся в корневом каталоге диска *d:*, и имеет следующий вид:

```
int max (int a,int b)
{
 int c;                    /*рабочая переменная */
 if (a>=b) c=a;
 else c=b;
 return c;
}
```

Во второй строке программы записан прототип функции *max()*, который необходим компилятору для проверки соответствия типов аргументов и параметров при вызове функции. Вызов функции является выражением в правой части оператора присваивания *z=max(x,y);*, при выполнении которого значения аргументов *x* и *y* подставляются вместо параметров *a* и *b* соответственно. После выполнения тела функции возвращаемое значение передается в место вызова функции и присваивается переменной *z*.

Описание функции находится в файле, поэтому для включения файла в программу необходимо в тексте программы указать директиву препроцессора *#include "d:\max.cpp"*.

Описание функции может находиться в одном файле с главной программой. При этом директива *#include"max.cpp"* не указывается, а вместо нее помещается описание функции.

В рассмотренной программе функция имеет возвращаемое значение. Существуют задачи, которые не требуют передачи возвращаемого значения. Такой пример приведен ниже.

Пример 2. Использование функции без возвращаемого значения.

Составить программу, включающую в свой состав функцию, при обращении к которой обеспечивается выдача на экран символа, набранного на клавиатуре. При нажатии клавиши с символом С выполнение программы завершить.

```
#include <iostream.h>
void print(char);  // прототип функции
void main()
  {
    char x;
    do
      {
        cout << "\nВведите символ ";
        cin >> x;
        pr(x);
      }
    while (x!= "C');
  }
void pr(char a)
  {
    cout << "\tВведенный символ: " << a;
  }
```

В функции *pr()* отсутствует возвращаемое значение, поэтому обращение к функции осуществляется оператором вызова функции *pr(x);*. В результате обращения к функции на экран будет выведен символ, введенный с клавиатуры с помощью оператора *cin >> x;*.

5.2. Получение нескольких результатов

Во многих практических задачах ответных значений бывает несколько. Поскольку имя функции является носителем одного результирующего значения, для решения поставленной задачи напрямую его использовать нельзя. Рассмотрим, каким образом в таких случаях описываются и вызываются функции.

Аргументы функции передаются *по значению*, поэтому прямым путем значения соответствующих переменных в функции изменять нельзя (чтобы вернуть ответные значения в качестве результатов вычисления). Однако соответствующей цели можно достичь, если в качестве параметров и аргументов функции использовать адреса переменных, а не их значения.

Пример 1. Получение нескольких результатов функцией.

Пусть требуется составить программу, содержащую обращение к функции, которая обменивает переменные их значениями. Возможный вариант решения имеет вид:

```
#include < iostream.h>
void change(int*,int*);    // Прототип функции
void main()
  { int a,b;
    cout << "\nВведите a и b";
    cin >> a; cin >> b;
    change(&a,&b);
    cout << "\nНовые значения: a=" << a << "\tb=" << b);
  }
void change(int *x,int *y)/*заголовок функции*/
  {
    int z;                 // x и y - указатели,
    z=*x;*x=*y;*y=z;       // *x и *y - значения, на которые они указывают
  }
```

В приведенной программе указатели *&a* и *&b* используются в качестве аргументов функции *change()*. При обращении к функции значения адресов-аргументов передаются по значению указателям-параметрам *x* и *y*. При выполнении функции адресуемые указателями значения меняются местами и управление передается обратно в точку вызова функции.

Замечание.

В языке C++ имя массива является константным указателем адреса элемента массива с нулевым индексом. Отсюда следует, что при использовании имени массива в качестве параметра функции допускается изменять значения элементов массива.

Пример 2. Массив как параметр функции.

Пусть требуется составить программу, содержащую обращение к функции *der( )*, которая уменьшает на 1 значения всех элементов массива-параметра. Возможный вариант решения имеет вид:

```
#include <iostream.h>
void der(int,float[]);              // прототип функции
void main()
  {
    const int k=10;
    float a[k];
    int i;
    for (i=0;i<k;i++)
      {
        cout << "\nВведите элемент массива ";
        cin >> a[i];
      }
    der(k,a);                       // обращение к функции
    for (i=0;i<k;i++)
      cout << "\ta[" << i << "]=" << a[i];
  }
void der(int k1,float a1[])
  {
    int j;
    for (j=0;j<k1;j++)
      a1[j]--;
  }
```

5.3. Функции с переменным числом параметров

В языке C++ можно применять функции, у которых до выполнения программы число параметров заранее не известно. Количество и, соответственно, типы параметров становятся известными при вызове функции. Такие функции называются ***функциями с переменным числом параметров***. Формат описания функции с переменным числом параметров следующий:

```
тип имя (список_явных_ параметров, ...)
  {
    тело_функции
  }
```

Здесь:

тип — тип возвращаемого значения;

список_явных_параметров — список параметров, которые известны до вызова функции.

После списка явных параметров ставится необязательная запятая и многоточие, которое сообщает компилятору о том, что контроль типов и количества параметров при вызове функции проводить не следует.

При программировании функций с переменным числом параметров необходимо предусмотреть механизм определения количества параметров и их типов. При реализации этого механизма используется два следующих подхода:

- один из параметров определяет число дополнительных параметров функции;
- в список явных параметров последним задается параметр-индикатор, указывающий на окончание списка параметров.

Переход от одного параметра к другому в обоих случаях осуществляется с помощью указателей.

Пример 1. Задание числа дополнительных параметров функции с помощью одного из параметров.

Пусть требуется составить программу, содержащую функцию с переменным числом параметров. Функция должна вычислять сумму значений дополнительных параметров. Список явных параметров должен состоять из одного параметра, используемого для задания числа дополнительных параметров.

```
/* Функции с переменным числом параметров */
#include <iostream.h>
int sum(int, ...);   // прототип функции
void main()
  {
    cout << "\n 4+6=" << sum(2,4,6);
    cout << "\n 1+2+3+4+5+6=" << sum(6,1,2,3,4,5,6);
    cout << "\n Параметры отсутствуют. Сумма равна:" << sum(0);
  }
int sum(int n, ...)   // n - число суммируемых параметров
  {
    int *p=&n;  // выход на начало списка дополнительных параметров
    int s=0;
    for (int i=1;i<=n;i++)
      {
        if (n==0) break;
        s+=*(++p);
      }
    return s;
  }
```

Результат выполнения программы:

```
4+6=10
1+2+3+4+5+6=21
Параметры отсутствуют. Сумма равна: 0
```

В приведенном примере для организации анализа аргументов используется значение первого параметра. Для доступа к нему используется указатель *p*. Ему присваивается значение адреса первого аргумента, который соответствует явно

заданному параметру *n*. Таким образом указатель устанавливается на начало списка аргументов в памяти. Затем в цикле указатель *p* перемещается по адресам следующих аргументов, соответствующим дополнительным параметрам. С помощью операции разыменования **p* выполняется выборка и суммирование значений аргументов.

Замечание.

Особенность рассмотренного подхода заключается в том, что все параметры (явный и дополнительные) должны иметь одинаковый тип *int* или *double*.

Пример 2. Определение конца списка дополнительных параметров функции с помощью параметра-индикатора.

Пусть требуется составить программу, содержащую функцию с переменным числом параметров. Функция должна вычислять произведение значений дополнительных параметров. Для определения конца списка параметров используем параметр-индикатор.

```
#include <iostream.h>
double pr(double a...)
  {
    double b=1.0;        // b -  произведение
    double *p=&a;
    if(*p==0.0) return 0.0;
    for(; *p; p++)  b* =*p;
    return b;
  }
void main()
  {
    cout << "\n pr(6e0,5e0,3e0,0e0)=" << pr(6e0,5e0,3e0,0e0);
    cout << "\n pr(1.5,2.0,0.0,3.0,0.0)=" << pr(1.5,2.0,0.0,3.0,0.0);
    cout << "\n pr(0.0)=" << pr(0.0);
  }
```

Результат выполнения программы:

```
pr(6e0,5e0,3e0,0e0)=90
pr(1.5,2.0,0.0,3.0,0.0)=3
pr(0.0)=0
```

В функции *pr()* перемещение по списку аргументов осуществляется за счет изменения значения указателя *p*. При втором обращении к функции в середине списка аргументов находится аргумент с нулевым значением. Он воспринят функцией как индикатор, который сигнаизирует об окончании списка аргументов. В приведенной программе, так же как и в первом примере, аргументы должны иметь один тип, но не обязательно *int* или *double*.

Для использования в программе функций с переменным числом параметров, при вызове которых можно использовать аргументы разных типов, необходимо включать в текст программы специальный набор макроопределений. Эти макроопределения находятся заголовочном файле *stdarg.h*.

5.4. Рекурсивные и подставляемые функции

В инженерной практике приходится иногда программировать рекурсивные алгоритмы. Такая необходимость возникает при реализации динамических структур данных, таких как стеки, деревья, очереди. Для реализации рекурсивных алгоритмов в C++ предусмотрена возможность создания рекурсивных функций. ***Рекурсивная функция*** представляет собой функцию, в теле которой осуществляется вызов этой же функции.

Пример 1. Использование рекурсивной функции для вычисления факториала.

Пусть требуется составить программу вычисления факториала произвольного положительного числа.

```
#include <iostream.h>
int fact(int n)
  {
    int a;
    if (n<0) return 0;
    if (n==0) return 1;
    a =n * fact(n-1);
    return a;
  }
void main()
  {
    int m;
    cout << "\nВведите целое число:";
    cin >> m;
    cout << "\n Фактериал числа " << m << " равен " << fact(m);
  }
```

Для отрицательного аргумента факториала не существует, поэтому функция в этом случае возвращает нулевое значение. Так как факториал 0 равен 1 по определению, то в теле функции предусмотрен и этот вариант. В случае когда аргумент функции *fact()* отличен от 0 и 1, то вызывается функция *fact()* с уменьшенным на единицу значением параметра и результат умножается на значение текущего параметра. Таким образом, в результате встроенных вызовов функций будет возвращен следующий результат:

```
n * (n-1) * (n-2) * . . . * 2 * 1 * 1
```

Последняя единица при формировании результата принадлежит вызову *fact(0)*.

Поскольку в языке C++ отсутствует операция возведения в степень, то для выполнения возведения в степень вещественного ненулевого числа можно использовать рекурсивную функцию.

Пример 2. Использование рекурсивной функции для возведения в степень.

Пусть требуется составить программу возведения в положительную целую степень вещественного ненулевого числа.

```
/* Возведение основания x в степень n */
#include <iostream.h>
float st(float x,int n)
```

```
  {
    if (n==0) return 1;
    if (x==0) return 0;
    if (n>0) return x*st(x,n-1);
    if (n<0) return st(x,n+1)/x;
  }
void main()
  {
    int m;
    float a,y;
    cout << "\nВведите основание степени:";
    cin >> a;
    cout << "\nВведите степень числа:";
    cin >> m;
    y=st(a,m);
    cout << "\n Число " << a << " в степени " << m << "  равно: " << y;
  }
```

В функции предусмотрены четыре ветви выполнения возведения в степень вещественного числа:

- степень числа равна нулю;
- основание равно нулю;
- степень больше нуля;
- степень меньше нуля.

При вызове функции, когда основание степени больше нуля, например, если при вводе заданы значения переменных a=2.0 и m=4, тогда обращение *st(a,m)* приводит к следующему вычислению:

```
2.0 * 2.0 * 2.0 * 2.0 * 1
```

Вызов функции для отрицательной степени, например, когда значения аргументов равны a=3.0 и m=–2 приводит к вычислению следующего выражения:

```
1/3.0/3.0
```

Некоторые функции языка C++ целесообразно определять как ***подставляемые***. *Подставляемые* функции определяются с помощью ключевого слова ***inline***.

Например:

```
inline float module (float x=0, float y=0)
  {
    return sqrt(x*x+y*y);
  }
```

Функция *module()* определена как подставляемая. Отличие подставляемых от других функций в том, что при вызове функции компилятор будет встраивать тело функции в место вызова функции. Таким образом, при многократных вызовах подставляемой функции размеры программы будут увеличиваться, однако исключаются затраты на передачу управления в вызываемую функцию и затраты на возврат из нее.

Наиболее эффективно использование подставляемой функции в том случае, когда ее тело состоит из небольшого числа операторов.

5.5. Области действия переменных

При объявлении переменных в программе большое значение имеет то место, где она объявлена. От того, где объявлена переменная, зависит возможность ее использования.

В языке C++ возможны три места объявления переменных.

Во-первых, вне каких либо функций, в том числе и *main()*. Такая переменная называется ***глобальной*** и может использоваться в любом месте программы от места объявления и до конца программы.

Во-вторых, переменная может быть объявлена внутри блока, в том числе и внутри тела функции. Объявленная таким образом переменная называется ***локальной*** и может использоваться только внутри блока. Такая переменная вне блока, в котором она объявлена, не известна.

В третьих, переменная может быть объявлена как ***параметр функции***. Кроме специального назначения, а именно для передачи данных в функцию, параметр можно рассматривать как локальную переменную для тела функции. Для иллюстрации указанных положений рассмотрим пример.

Пример. Составить программу вычисления суммы *k* чисел.

```
#include <iostream.h>
void sum(int );          // прототип функции
  int s=0;               // глобальная переменная
void main()
 {
   int i,b,k;            // локальные переменные
   cout << "\nВведите число слагаемых ";
   cin >> k;
   for (i=0;i<k;i++)
     {
       cout << "\nВведите новое слагаемое ";
       cin >> b;
       sum(b);           // обращение к функции
     }
   cout << "\ns=" << s;
 }
void sum(int c)
  {
    s=s+c;
  }
```

В данной программе переменная *s* является глобальной, она доступна из обеих функций программы — *main()* и *sum()*, а переменные *i*, *b*, *k* и *c* — локальные, доступны только в тех функциях, где они объявлены.

Если глобальная и локальная переменные имеют одно и то же имя, тогда считается, что объявлены две различные переменные со своими областями использования. При этом локальная переменная будет видима в той функции, где она объявлена, а глобальная во всей программе за исключением функции, в которой объявлена локальная переменная.

5.6. Библиотечные функции

При появлении нового компилятора фирма-разработчик обеспечивает пользователя довольно большим перечнем библиотечных функций. Типичным является следующий набор функций:

- манипулирование строками;
- сравнение и преобразование символов;
- преобразование строк в числа и обратно;
- дата и время дня;
- математические функции;
- функции ввода-вывода;
- сортировка;
- графические.

Здесь мы рассмотрим *математические* и *графические* функции как наиболее часто используемые при решении различных задач. Обычно прототипы математическихфункций входят в файл *math.h*, а графических – в файл *graphics.h*. Если такой файл отсутствует, то прототип функции необходимо записать следующим образом:

```
extern double имя_функции ( );
```

Функции будут работать корректно и для переменных типа *float*. Попытки использовать функции с данными типа *long* связаны с некорректным преобразованием типов и приводят к ошибкам. Перечень основных математических функций приведен в табл. 5.1.

Пример 1. Использование математических функций.

Пусть требуется составить программу для вычисления $y = x^2 - \cos z$.

Возможный вариант программы, предназначенной для решения поставленной задачи, имеет вид:

```
#include <iostream.h>
#include <math.h>
void main()
  {
    float x;
    double z,y;
    cout << "\nВведите x ";
    cin >> x;
    cout << "\nВведите z ";
    cin >> z;
    y=pow(x,2)-cos(z);
    cout << "\ny=" << y;
  }
```

Таблица 5.1

Математические функции

Имя	Результат выполнения функции
abs	абсолютное значение аргумента типа *int*
cos	косинус угла, заданного в радианах
exp	экспонента в степени с аргументом типа *double*
fabs	абсолютное значение аргумента типа *double*
log	натуральный логарифм аргумента типа *double*
log10	десятичный логарифм аргумента типа *double*
max	максимальный из двух аргументов типа *int*
min	минимальный из двух аргументов типа *int*
pow	значение аргумента 1 в степени аргумента 2
sin	синус угла, заданного в радианах
sqrt	квадратный корень положительного аргумента типа *double*
tan	тангенс угла, заданного в радианах

В табл. 5.2 приведен перечень наиболее часто используемых графических функций для работы под управлением MS DOS.

Рассмотрим пример использования графических функций.

Пример 2. Использование графических функций.

Вывести на экран компьютера линию и прямоугольник. Координаты начала и окончания линии и вершин прямоугольника задать самостоятельно.

```
#include <iostream.h>          // прототипы функций ввода-вывода
#include <graphics.h>          // прототипы графических функций
#include <conio.h>             // прототип функции getch()
void main()
{                              // Инициализация графики
  int a=DETECT,b;
   initgraph(&a,&b,"F:\\BC5\\bgi");
   int d,c;
   int x1,x2,y1,y2;
   cout << "\nВведите коодинаты начала прямой x и y\n";
   cin >> x1 >> y1;
   cout << "\nВведите коодинаты окончания прямой x и y\n";
   cin >> x2 >> y2;
   line(x1,y1,x2,y2);
   getch();                    // ждать нажатия клавиши
```

Таблица 5.2

Графические функции

Имя	Результат выполнения функции
arc	Вычерчивание дуги окружности
bar	Вычерчивание закрашенного прямоугольника
circle	Вычерчивание окружности
cleardevise	Очистка экрана цветом фона
closegraph	Перевод системы в текстовый режим
getcolor	Возвращение номера текущего цвета изображения
getbcolor	Возвращение номера текущего цвета фона
getpixel	Возвращение цвета заданной точки
initgraph	Перевод системы в графический режим
line	Вычерчивание линии
outtextxy	Вывод текстовой строки
putpixel	Вычерчивание точки
rectangle	Вычерчивание прямоугольника
setcolor	Установка цвета изображения
setbcolor	Установка цвета фона

```
    rectangle (x1,y1,x2,y2);
    getch();                    // ждать нажатия клавиши
    d=getbkcolor();             // запоминание текущего цвета фона
    c=getcolor();               // запоминание цвета изображения
    setcolor(d);                // установить цвет изображения
    line(x1,y1,x2,y2);          // нарисовать линию цветом фона
    setcolor(c);                // восстановление цвета изображения
    getch();                    // ждать нажатия клавиши
    closegraph();               // закрыть графический режим
}
```

В программе используются следующие графические функции:

- *void line*(*int x*1, *int y*1, *int x*2, *int y*2); — вычерчивание линии с координатами концов (*x*1, *y*1) и (*x*2, *y*2).
- *void rectangle* (*int x*1, *int y*1, *int x*2, *int y*2); — вычерчивание прямоугольника с координатами вершин (*x*1, *y*1) и (*x*2, *y*2);
- *int getbkcolor*(*void*) — возвращение номера текущего цвета фона;

- *int getcolor(void)* — возвращение номера текущего цвета изображения;
- *void setcolor(int color)* — установка цвета изображения.

Перед использованием функций графической библиотеки необходимо перевести монитор компьютера в графический режим. Этот переход выполняет функция initgraph(), а выход из графического режима осуществляет функция *closegraph*().

В результате выполнения программы на экране вычерчивается линия и прямоугольник. Линия является диагональю прямоугольника. Координаты прямоугольника и линии задаются по запросу. В конце программы линия рисуется цветом фона, тем самым она стирается с экрана.

Контрольные вопросы и задания

1. Поясните назначение прототипа функции.
2. Какова связь между параметрами функции и аргументами?
3. Каким образом можно обеспечить получение с помощью функции нескольких результатов?
4. Что представляют собой функция с переменным числом параметров?
5. Укажите формат описания функций с переменным числом параметров.
6. Поясните механизм определения числа дополнительных параметров функции с помощью одного параметра.
7. Как выполняется определение конца списка параметров с помощью параметра-индикатора?
8. Укажите особенности описания рекурсивных функций.
9. Каковы особенности описания и в чем смысл использования подставляемых функций?
10. Какие переменные называются глобальными?
11. В чем отличие глобальных переменных от локальных?
12. Охарактеризуйте математические функции.
13. В каком файле размещаются прототипы математических функций?
14. Привидите примеры графических функций.
15. С помощью какой функции система переводится в графический режим?
10. Поясните назначение возвращаемого значения.
11. Как оформляется при описании функции отсутствие возвращаемого значения?
12. В каком случае в программе прототип функции не обязателен?
13. Укажите возможные варианты размещения описания функций.
14. Составьте программу с использованием функций для выполнения вычислений по следующим формулам:

$z=max(x, y)$;

$c=min(n, m)$;

При составлении программы описание функции поместите:

а) в основной программе;

б) в отдельном файле.

6. МАССИВЫ В КАЧЕСТВЕ ПАРАМЕТРОВ ФУНКЦИЙ

В настоящем разделе рассматриваются вопросы использования в качестве параметров функций одномерных, многомерных статических и динамических массивов. Для случая одномерных массивов приводятся примеры использования строк, статических и динамических массивов.

6.1. Одномерные массивы

Массивы могут быть параметрами функций и функции в качестве результата могут возвращать указатель на массив. Рассмотрим эти возможности.

При использовании массивов в качестве параметров функции возникает необходимость определения в теле функции количества элементов массива, который является аргументом при обращении к функции.

При работе со ***строками***, т. е. с массивами типа *char[]*, проблема решается просто, поскольку последний элемент строки имеет значение '\0'. Поэтому при обработке массива-аргумента каждый его элемент анализируется на наличие символа конца строки.

Пример 1. Строки как параметры функций.

Требуется составить программу, содержащую функцию, которая подсчитывает число элементов в строке. Возможный вариант программы имеет вид:

```
// Массивы в функциях
#include <iostream.h>
int dl(char[]);              // прототип функции dl

void main()
  {
    char p[]="кафедра";
    cout << "\n Длина строки равна:" << dl(p);
  }
 int dl(char c[])
  {
    int i;
```

```
    for(i=0;;i++)
    if (c[i]=='\0') break;
    return i;
  }
```

В результате выполнения программы на экран будет выдано сообщение:

```
Длина строки равна: 7.
```

Если массив-параметр функции не является *символьной строкой*, то необходимо либо использовать массивы с заранее известным числом элементов, либо передавать размер массива с помощью дополнительного параметра. Следующие два примера иллюстрируют эти возможности.

Пример 2. Одномерный массив в качестве параметра функции.

Пусть требуется составить программу, в которой функция вычисляет сумму элементов массива, состоящего из 5 элементов. Возможный вариант программы имеет вид:

```
// Одномерный массив в качестве параметра
#include <iostream.h>
int sum(float x[5])
  {
    float s=0;
    for(int i=0;i<5;i++)
    s=s+x[i];
    return s;
  }
void main()
  {
    float z[5],y;
    for(int i=0;i<5;i++)
      {
        cout <<"\nВведите очередной элемент массива:";
        cin >> z[i];
      }
    y=sum(z);
    cout <<"\nСумма элементов массива:"<< y;
  }
```

В результате выполнения программы на экран будет выведено значение суммы элементов массива *z*.

В данном примере отсутствует прототип функции *sum()*, поскольку описание функции *sum()* транслятор анализирует раньше, чем обращение к ней. В приведенном примере заранее известно число элементов — 5, поэтому в функции всего один параметр — массив *x*. Поскольку в функции существует возвращаемое значение, то вызов функции может быть только выражением или частью выражения. В программе оператор присваивания *y=sum(z);* содержит такое выражение в правой части. Здесь аргументом функции является массив *z*.

Пример 3. Одномерный массив с произвольным числом элементов в качестве параметра функции.

Пусть требуется составить программу с обращением к функции, которая возвращает максимальный элемент одномерного массива с произвольным числом членов.

```
// Поиск максимального элемента
#include <iostream.h>
float max(int n,float a[])
  {
    float m=a[0];
    for(int i=1;i<n;i++)
    if (m<a[i]) m=a[i];
    return m;
  }
void main()
  {
    float z[6];
    for(int i=0;i<6;i++)
      {
        cout <<"\nВведите очередной элемент массива:";
        cin >> z[i];
      }
    cout <<"\nМаксимальный элемент массива:"<< max(6,z);
  }
```

В результате выполнения программы на экран монитора будет выдано сообщение со значением максимального элемента массива *z*, который в данном примере состоит из шести элементов.

В приведенной программе прототип отсутствует, поскольку описание функции следует раньше ее вызова. В функции *max()* используется два параметра: первый указывает число элементов в массиве, а второй — имя и размерность массива. Аргументы указаны при вызове функции *max(6, z)*. Здесь 6 — размер массива, а *z* — имя массива.

Имя массива представляет собой ***константный указатель адреса*** нулевого элемента массива. Поэтому у любого массива, используемого в качестве параметра функции, можно осуществлять изменение его элементов при выполнении тела функции.

Пример 4. Массивы с произвольным числом элементов как параметры функции.

Пусть требуется составить программу с функцией, формирующей в качестве результата массив, каждый элемент которого является максимальным из соответствующих значений элементов двух других массивов-параметров. Возможный вариант программы имеет вид:

```
// Указатели на массив в качестве параметров
#include <iostream.h>
void max1(int,int*,int*,int*); // прототип функции
void main()
```

```
{
  int a[]={0,1,2,3,4};
  int b[]={5,6,0,7,1};
  int d[5];
  max1(5,a,b,d);
  cout << "\n";
  for (int i=0;i<5;i++)
  cout << "\t" << d[i];
}
void max1(int n,int *x,int *y,int *z)
{
  for (int i=0;i<n;i++)
  z[i]=x[i]>y[i] ? x[i]:y[i];
}
```

В результате выполнения программы на экран монитора будут выданы элементы результирующего массива d: 5 6 2 7 4.

В приведенной программе у функции имеется четыре параметра: число *n* элементов в массивах и три указателя *x*, *y*, *z*. При вызове функции *max1(5,a,b,d);* (с помощью оператора, поскольку отсутствует возвращаемое значение) в качестве аргументов используются имена исходных массивов *a* и *b* и результирующего *d*.

6.2. Многомерные массивы

Особенностью языка C++ является несамоопределенность массивов, т. е. по имени массива невозможно узнать его размерность и размеры по каждому измерению. Кроме того, в C++ многомерные массивы не определены. Например, если объявлен массив *float d[3][4][5]*, то это не трехмерный, а одномерный массив *d*, включающий три элемента, каждый из которых имеет тип *float [4][5]*. В свою очередь, каждый из четырех элементов типа *float [5]*. И, соответственно, каждый из этих элементов является массивом из пяти элементов типа *float*. Эти особенности затрудняют использование массивов в качестве параметров функций.

При передаче массивов в качестве параметров через заголовок функции следует учитывать, что передавать массивы можно только с одной неопределенной границей мерности (эта мерность должна быть самой левой).

Пример 1. Двумерный массив как параметр функции.

Пусть требуется составить программу с функцией, которая подсчитывает сумму элементов матрицы.

```
#include <iostream.h>
float summa(int n,float a[][3])
 {
   float s=0;
   for(int i=0;i<n;i++)
     for(int j=0;j<3;j++)
```

```
    s=s+a[i][j];
    return s;
  }
void main()
  {
    float z[4][3]={ 0, 1, 2, 3,
                    4, 5, 6, 7,
                    7, 6, 5, 4 };
    cout << "\n Сумма элементов матрицы равна" << summa(4,z);
  }
```

В результате выполнения программы на экран будет выведено сообщение: «Сумма элементов матрицы равна 50».

В приведенной программе параметром функции *summa()* является матрица *a[][3]*, у которой не определено число строк. Число столбцов должно быть заранее определено, оно равно 3. Таким образом, при использовании данной функции вводится существенное ограничение — фиксированное число столбцов. Указанное ограничение можно обойти с помощью использования вспомогательных массивов указателей на массивы.

Пример 2. Вспомогательный массив указателей на массив как параметр функции.

Требуется составить программу с функцией, которая возвращает в качестве результата минимальный элемент матрицы *d* размером *m*×*n*.

```
#include <iostream.h>
float min(int m,int n,float *p[]);
void main()
  {
    float d[3][4]={ 1,2,-2,4,
                    5,0,-3,18,
                   -9,6, 7,9 };
    float *r[]=
      {
        (float *) &d[0], (float *) &d[1],(float *) &d[2]
      };
    int m=3;
    int n=4;
    cout << "\n Минимальный элемент матрицы равен " << min(m,n,r);
  }
float min(int m,int n,float *p[])
  {
    float x=p[0][0];
    for (int i=0;i<m;i++)
      for (int j=0;j<n;j++)
        if (x>p[i][j]) x=p[i][j];
    return x;
  }
```

В результате выполнения программы на экран будет выведено сообщение: «Минимальный элемент матрицы равен –9».

В функции *min()* в качестве параметров используются *int m* — число строк, *int n* — число столбцов и *float *p[]* — массив указателей на одномерные массивы элементов типа float, причем размеры двухмерного массива заранее не известны.

В теле функции обращение к элементам матрицы осуществляется с помощью двойного индексирования. Здесь *p[i][j]* — значение элемента двухмерного массива.

В функции *main()* объявлена и инициализирована матрица *d[3][4]*, имеющая фиксированные размеры. Такой размер нельзя использовать непосредственно в качестве аргумента функции, поэтому объявлен дополнительный вспомогательный массив указателей *float *r[]*. В качестве значений элементам этого массива присваиваются адреса первых элементов строк матрицы, т. е. *&d[0], &d[1], &d[2]*, преобразованные к типу *float ** (указатель на данные типа *float*).

6.3. Динамические массивы

Часто при решении инженерных задач размеры массивов заранее не известны. Они определяются в ходе решения поставленной задачи. Такие массивы называются ***динамическими***. Организовать динамические массивы можно с помощью операции *new*, которая позволяет выделить в динамической области памяти участок для размещения массива соответствующего типа.

Пример 1. Динамические массивы как параметры функций.

Требуется составить программу с использованием функции, которая заполняет элементы матрицы размером *m×n* последовательно значениями целых чисел: 0, 1, 2, 3, ... Для формирования матрицы в основной программе использованы динамические массивы, так как размеры матрицы заранее не известны.

```
// Матрица как набор одномерных динамических массивов
#include <iostream.h>
void fun(int m,int n,int **uc)
 {
   int k=0;
   for (int i=0;i<m;i++)
   for (int j=0;j<n;j++)
   uc[i][j]=k++;
 }


void main()
 {
   int **pi;              // указатель на массив указателей
   int m1;                // число строк в матрице
```

```
    cout << "\n Введите число строк матрицы : ";
    cin >> m1;
    int n1;                  // число столбцов в матрице
    cout << "\n Введите число столбцов матрицы : ";
    cin >> n1;
    int i,j;
    pi=new int* [m1];        // выделение вспомогательного массива указателей
    for (i=0;i<m1;i++)
      pi[i]=new int [n1];    // формирование i-й строки матрицы
    fun(m1,n1,pi);           // обращение к функции
    for (i=0;i<m1;i++)       // цикл перебора строк
      {
        cout << "\n строка " << i+1 << ":";
        for (j=0;j<n1;j++)
          cout << "\t" << pi[i][j];
      }
    for (i=0;i<m1;i++)
    delete pi[i];
    delete pi;
}
```

Если при вводе задать размерность матрицы 3×4, то в результате выполнения программы на экран будет выдано три следующие строки:.

```
строка 1: 0 1 2 3
строка 2: 4 5 6 7
строка 3: 8 9 10 11
```

В функции *fun()* есть три следующих параметра: *int m* — число строк, *int n* — число столбцов матрицы и *int* ***uc* — указатель на массив указателей.

В основной функции для определенности значение числа строк *m1* задано 3, а число столбцов матрицы *n1* задано 4. При выполнении оператора присваивания *pi=new int* [m1];* операцией *new* запрашивается память для вспомогательного массива указателей. В результате выполнения этой операции для массива указателей выделяется требуемое количество памяти, а адрес первого элемента выделенной памяти присваивается указателю *pi*. При выполнении в теле цикла второй операции *new* формируются *m1* строк матрицы (см. раздел, посвященный массивам и указателям).

При обращении к функции происходит присваивание элементам матрицы последовательно значений целых чисел от 0 до 11. В конце программы с помощью *delete* выделенная память освобождается.

Многомерный массив с переменными размерами, который формируется в теле вызываемой функции, невозможно целиком вернуть в качестве результата в вызывающую функцию. Однако возвращаемым значением функции может быть указатель на одномерный массив указателей на одномерные массивы с элементами заданного типа. Следующий пример демонстрирует такую возможность.

Пример 2. Формирование многомерного динамического массива в функции.

Пусть требуется составить программу с функцией, формирующую квадратную матрицу порядка *n* с единичными диагональными элементами и остальными нулевыми. Возможный вариант программы имеет вид:

```
// Единичная диагональная матрица порядка n
#include <iostream.h>
#include <process.h>       // для exit()

// функция формирования матрицы
int ** matr(int n)
  {
    int **p;               // вспомогательный указатель
    p=new int* [n];        // массив указателей на строки
    if (p==NULL)
      {
        cout << "\n Не создан динамический массив!!";
        exit(1);
      }
// цикл создания строк
    for (int i=0;i<n;i++)
      {
        p[i]=new int [n];  // выделение памяти для i-й строки
        if (p[i]==NULL)
          {
            cout << "\n Не создан динамический массив!";
            exit(1);
          }
// цикл заполнения строк
        for (int j=0;j<n;j++)
          if (j!=i) p[i][j]=0; else p[i][j]=1;
      }
    return p;
  }
void main()
  {
    int n1;                // порядок матрицы
    cout << "\n Введите порядок матрицы:";
    cin >> n1;
    int **ma;              // указатель для формирования матрицы
    ma=matr(n1);           // обращение к функции
// печать элементов матрицы
    for (int i=0;i<n1;i++)
      {
        cout << "\n строка " << i+1 << ":";
```

```
      for (int j=0;j<n1;j++)
      cout << "\t" << ma[i][j];
    }
  for (i=0;i<n1;i++)
    delete ma[i];
  delete ma;
}
```

При выполнении программы на экране появится подсказка: «Введите порядок матрицы:». Если в ответ на подсказку пользователь введет с клавиатуры порядок матрицы, равный 5, то на экране дисплея он получит сообщение из пяти строк вида:

```
строка 1: 1 0 0 0 0
строка 2: 0 1 0 0 0
строка 3: 0 0 1 0 0
строка 4: 0 0 0 1 0
строка 5: 0 0 0 0 1
```

В данной программе функция *matr()* имеет один параметр *int n* — порядок матрицы. В теле функции формируется квадратная матрица размером $n \times n$ (создаются n одномерных массивов с элементами типа *int* и массив указателей на эти одномерные массивы) и ее элементы заполняются необходимыми значениями. Возвращаемым значением функции *matr()* является значение указателя на сформированную матрицу.

В функции *main()* с клавиатуры задается значение *int n1*порядка матрицы.

Контрольные вопросы и задания

1. Каким образом можно определить число элементов в строке символов?
2. В чем отличие оформления списка параметров, когда известно и не известно число элементов во входном для функции массиве?
3. Что представляет собой имя массива и как оно связано с нулевым элементом массива?
4. Каким образом можно получить в качестве результата выполнения функции массив?
5. Как определяются в C++ многомерные массивы?
6. Поясните связь между массивами и указателями.
7. Как можно оформить использование динамических массивов в функциях?
8. Составить функцию, которая подсчитывает число повторений заданного символа в произвольной строке.
9. Составить программу, в которой функция находит минимальный элемент в произвольном одномерном массиве.
10. Составить программу, выполняющую с помощью функции упорядочение по возрастанию элементов одномерного динамического массива размером m.
11. Составить функцию, которая возвращает в качестве результата максимальный элемент матрицы d размером $m \times n$.

12. Составить функцию, подсчитывающую сумму элементов произвольной матрицы размером $m \times n$.

13. Составить программу, в которой с помощью функции подсчитывается количество k отрицательных элементов в массиве q размером $m \times n$.

14. Составить программу для подсчета числа точек x_i, y_i, $i = 1, \ldots, N$, попавших в круг радиуса R с центром x_0, y_0.

15. Составить программу, выполняющую с помощью функции вычисление элементов матрицы размером $m \times n$ по формуле

$$d_{ij} = (i + j)(x_i + y_j),$$

где $i = 1, \ldots, m; j = 1, \ldots, n$.

16. Составить программу, выполняющую с помощью функции транспонирование квадратной матрицы a размером $m \times n$. При транспонировании матрицы симметричные относительно главной диагонали элементы меняются местами.

7. ИСПОЛЬЗОВАНИЕ ПРЕПРОЦЕССОРА

7.1. Общие сведения

Препроцессор входит в состав системы программирования C++ и обеспечивает обработку исходного текста программы перед ее компиляцией. В ходе препроцессорной обработки программы реализуются следующие действия:

1. Выполняется подстановка макрорасширений вместо имен макросов и их аргументов.

2. В исходный текст программы включаются файлы, содержащие вспомогательную информацию (описания объектов, классов, прототипы функций и т. п.).

3. Выполняется условная компиляция программы – определение окончательного вида исходного текста программы путем оставления или исключения отдельных ее фрагментов.

Задание препроцессору описывается с помощью директив *препроцессора*, которые начинаются со знака #, размещаемого с первой позиции в строке.

Препроцессорная обработка позволяет сократить время разработки программ, повысить их наглядность и мобильность. В частности, при изменении характеристик аппаратно-программных средств достаточно изменить содержимое подключаемых файлов.

7.2. Определение и обработка макросов

Определение и отмена определения макроса осуществляется с помощью директив препроцессора ***#define*** и ***#undef*** соответственно. Директива определения макроса задается в следующем формате:

```
#define имя_макро(список параметров) тело_макро
```

Элемент *имя_макро* задает имя макроопределения, вместо которого в ходе препроцессорной обработки подставляется макрорасширение. В простейшем варианте (при отсутствии необязательных параметров) вместо имени подставляется просто *тело_макро*. Если в элементе *имя_макро* параметр присутствует (в обязательных круглых скобках), то вместо имени макроопределения подставляется *тело_макро*, в кото-

ром вместо параметра всюду подставляется аргумент обращения к макросу с указанным именем, например:

```
#define GM "good morning"
#define M 100
#define power2(a) a*a
. . .
char s[M];                  // расширяется в char s[100];
puts(GM);                   // расширяется в puts("good morning");
y=power2(x);                // расширяется в y=(x)*(x);
```

В сравнении с использованием функций макросредства позволяют уменьшить время выполнения программы путем исключения потерь на организацию передачи управления, но платой за это является увеличение размера загрузочного модуля.

Следует проявлять внимательность при задании аргументов для макро. В отличие от функций, при обращении к макросу нет контроля соответствия типов аргументов и параметров. Кроме того, здесь возможны отличия в правилах вычисления результата и изменения аргументов. В связи с отмеченным рекомендуется по возможности не использовать макросредства.

Рассмотрим возможные отличия в применении макроса и функции на примере возведения в квадрат:

```
#define power2(a) a*a . . .
int y,x=5,z,b=5;
y=power2(++x);              //расширяется в y=(++x)*(++x);
//в результате y=42 и x=7
z=pow(++b,2);               //в результате z=36 и b=6
```

При необходимости описание макроса можно продолжить в нескольких строках. Для этого в конце продолжаемой строки указывается символ \.

С помощью символов ## в макросе с параметрами можно обеспечить удаление всех символов между символами ##, в том числе и сами эти символы. В результате вызова макроса соответствующие аргументы объединяются в единую лексему, например:

```
#define macf(k,r) k##-##r
macf(y,2)                   //расширяется в y2
```

В теле макроса перед параметрами можно задавать символ #. Это означает, что соответствующие аргументы при расширении макроса должны в этом месте должны преобразовываться в строки символов. Например:

```
#define FP(rez) printf(#rez"=%d\n",rez);
. . .
int y=5; FP(y);             //Расширяется в printf("y""=%d\n",y);
                            //Расположенные подряд без разделителей
                            //строковые литералы рассматриваются как один
```

На печать будет выдано y=5.

В программе допускается организовывать *вложенные макросы*. Это обеспечивается тем, что после получения расширений макроса осуществляется повторный просмотр текста, например:

```
#define G good  #define GM G morning
```

Директива *отмены определения* макроса имеет формат:

```
#undef имя_макро
```

Действие макроса в программе распространяется от строки определения макроса до строки отмены определения макроса.

7.3. Включение файлов

Включение текстовых файлов в исходный текст программы перед компиляцией выполняется с помощью директивы ***#include***, имеющей два следующих формата:

```
#include <имя файла>
#include "спецификация_файла"
```

Например:

```
#include <stdio.h>
#include <iostream>
#include "a:\sys\prototip.txt"
```

Первый вариант применяется для *стандартных заголовочных файлов* с расширением *.h*. Поиск таких файлов выполняется в каталогах, местоположение которых задается в настройках системы программирования.

Напомним, что в стандарте C++ расширение имени заголовочного файла не обязательно. Именно такой вариант иллюстрируется во второй строке с директивой препроцессора.

Во третьем варианте поиск файла выполняется в соответствии с указанной спецификацией. Если файл по спецификации не найден, то поиск его продолжается в тех же каталогах, что и в первом варианте.

7.4. Условная компиляция

Условная компиляция позволяет *включать* или *исключать* отдельные фрагменты программы перед компиляцией. В частности, с помощью условной компиляции можно выполнить предварительную настройку программы на технические характеристики аппаратуры или особенности программного обеспечения, в среде которых планируется ее выполнение. В зависимости от задаваемых с помощью директив препроцессора условий принимается решение по оставлению или исключению фрагментов программы.

Условная компиляция реализуется с помощью следующих пяти директив:

#if константное_выражение — оставить фрагмент для компиляции, если выражение истинно (отлично от нуля);

#ifdef идентификатор — оставить фрагмент, если идентификатор определен как макрос;

#ifndef идентификатор — оставить фрагмент, если идентификатор не определен как макро;

#else — применяется в сочетании с указанными выше директивами как отрицание условия; *#elif* выражение — равносильно *#else*, в котором дополнительно проверяется истинность выражения;

#endif — завершает область действия любой из перечисленных директив.

Конструкции *#if* допускают вложение друг в друга произвольное число раз при условии соблюдения соответствия между *#if* и *#endif*.

Пример. Использование условной компиляции.

```
#ifndef BFSIZE #define BFSIZE 20 #endif
#include <stdio.h>
void main(void)
  {
   printf("BFSIZE=%d\n",BFSIZE);
  }
```

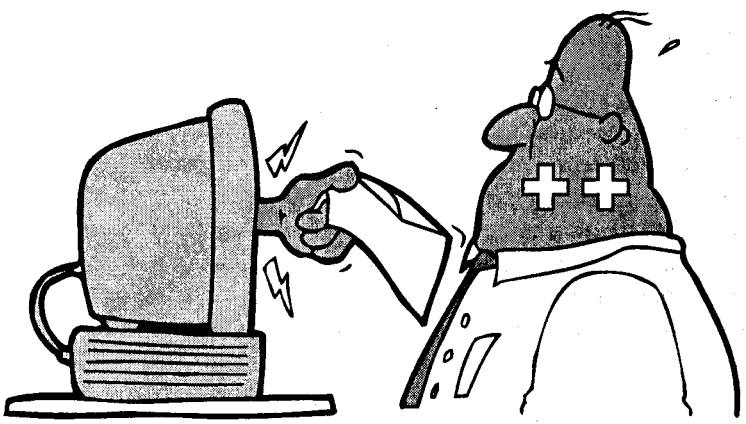
Выполнение приведенной программы после компиляции дает результат *BFSIZE*=20, если значение *BFSIZE* не определено в интегрированной среде системы программирования. В противном случае в качестве результата будет выдано значение, определенное для *BFSIZE* в настройках системы программирования.

Контрольные вопросы и задания

1. На каком этапе работы с программой осуществляется препроцессорная обработка?
2. Укажите формат директив препроцессора.
3. Укажите достоинства и недостатки использования макросов.
4. Назовите основные области применения директив препроцессора.
5. Приведите формат директив препроцессора для включения файлов.
6. Для чего предназначена директива препроцессора *#include*?
7. Каково назначение директив препроцессора условной компиляции?
8. Составьте программу, в которой с помощью директив условной компиляции обеспечивается вывод результатов на экран или в файл.

ЧАСТЬ 2

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



8. ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В настоящее время объектно-ориентированное программирование (ООП) является доминирующим стилем при создании больших программ. Рассмотрим основные этапы эволюции структурного подхода в программировании, которые помогают лучше понять взаимосвязь структурного подхода, модульного программирования и ООП.

8.1. Структурный подход в программировании

Первые программы для цифровых вычислительных машин редко превышали объем 1 кбайт. С той поры существенно изменилась архитектура и технические характеристики программируемых вычислительных средств (ВС), чрезвычайно снизилась стоимость хранения, пересылки и обработки 1 байта информации. Объемы используемых программ и программных систем измеряются не только десятками килобайтов, но и сотнями мегабайтов. Вместе с тем удельная стоимость создания программ (нормированная объемом программы) до последнего времени менялась мало. Более того, с ростом объема программы удельная стоимость ее создания могла нелинейно возрастать. Было обнаружено, что время создания сложных программ пропорционально квадрату или даже кубу объема программ. Поэтому не удивительно, что одним из основных факторов, определяющих развитие технологии программирования, является снижение стоимости проектирования и создания программных продуктов (ПП), или борьба со сложностью программирования.

Другими важнейшими факторами, влияющими на эволюцию методов проектирования и создания ПП, являются:

- изменение архитектур вычислительных средств (ВС) в интересах повышения производительности, надежности и коммуникативности;
- упрощение взаимодействия пользователей с ВС и интеллектуализация ВС.

Действие двух последних факторов, как правило, сопряжено с ростом сложности программного обеспечения ВС. ***Сложность*** представляет неотъемлемое свойство программирования и программ, которое *проявляется* во времени и стоимости создания

программ, в объеме или длине текста программы, характеристиках ее логической структуры, задаваемой операторами передачи управления (ветвления, циклы, вызовы подпрограмм и др.).

Можно выделить 5 следующих источников сложности программирования:

1) решаемая задача;
2) язык программирования;
3) среда выполнения программы;
4) технологический процесс коллективной разработки и создания ПП;
5) стремление к универсальности и эффективности алгоритмов и типов данных.

От свойства сложности нельзя избавиться, но можно изменять характеристики его проявления путем управления или организации.

В программировании широко используется фундаментальный принцип управления сложными системами, который известен человеку с глубокой древности — *devide et impera* (разделяй и властвуй, лат.) и широко им применяется при разработке и проектировании любых сложных технических систем. Согласно первой части этого принципа, при проектировании сложной программной системы проводится ***алгоритмическая декомпозиция*** решаемой задачи.

Целью декомпозиции является представление разрабатываемой системы в виде взаимодействующих небольших подсистем (модулей или блоков), каждую из которых можно отладить (испытать) независимо от других. В этом случае при разработке системы, разделенной на «крупные» элементы или подсистемы, необходимо держать в уме информацию о гораздо меньшем числе деталей, чем в отсутствие такого разделения.

Наряду с термином *декомпозиция*, также используется термин ***структуризация*** проблемы, задачи или программы. Идеи разделения программ на относительно самостоятельные крупные части, реализующие определенные процедуры и функции и образующие определенную иерархию взаимосвязей, нашли отражение в ***структурном подходе*** к разработке и созданию программных средств. В программировании структурный подход появился с возникновением первых подпрограмм, процедур и функций, написанных в так называемом ***процедурно-ориентированном стиле***. Данный стиль опирается на простое правило: определить переменные и константы, которые понадобится хранить в памяти компьютера, и описать или использовать алгоритм их обработки.

Теоретическое оформление структурный подход получил в конце 60-х — начале 70-х годов в работах Э. Дейкстры, А.П. Ершова, Э. Йодана, Н. Вирта, Э. Брукса и других теоретиков и практиков программирования. Следует отметить появление структурного программирования, в котором нашла определенное отражение идея упорядочивания структуры программы. ***Структурное программирование*** ориентирует на составление программ, структура которых близка к «дереву» операторов или блоков. Использование структуры типа «дерево» в качестве своеобразного эталона объясняется тем, что это простая для анализа и реализации структура.

Дальнейшее развитие структурного подхода привело к ***модульному программированию***. Оно предусматривает декомпозицию прикладной задачи в виде иерархии взаимодействующих модулей или программ. Модуль, содержащий данные и

процедуры их обработки, удобен для автономной разработки и создания. Специализация модулей по видам обработки и наличие в них данных определенных типов — это свойства, которые отражают генетическую связь модульного программирования и ООП.

Важнейшими инструментами производителей ПП, в которых находят отражение практически все аспекты эволюции, являются языки программирования. Язык программирования изначально ориентирован на компьютер и содержит набор типов данных, операторов, операций, функций и других операционных единиц языка, которые достаточно просто могут быть переведены в инструкции (команды) по управлению аппаратным и программным обеспечением компьютера. При этом желательно максимизировать эффективность трансляции предложений языка в машинные коды в смысле минимизации требуемой памяти, времени выполнения программы и стоимости создания транслятора. Вместе с тем язык программирования ориентирован на программиста и предоставляет средства для моделирования объектов, их свойств и поведения при решении прикладных задач в некоторой предметной области в виде программ.

Развитие языков в направлении повышения эффективности составления прикладных программ привело к разделению языков по следующим уровням.

1. Низкий уровень (машинно-ориентированные языки — языки ассемблера).
2. Высокий уровень (процедурно-ориентированные языки: FORTRAN, ALGOL, PL/1, Pascal).
3. Уровень решаемой задачи (проблемно-ориентированные языки — GPS, SQL).

Введение ***типов данных*** обозначило еще одно направление развития технологии программирования. Типизация данных предназначена как для облегчения составления программ, так и для автоматизации выявления ошибок использования данных в виде операндов и фактических параметров при вызове функций. Использование ***структурных типов данных*** позволяет, во-первых, упростить работу алгоритмиста при сопоставлении структур данных прикладной задачи и данных, обрабатываемых процедурами и функциями программных модулей, и, во-вторых, сократить объем рутинной работы программиста при кодировании алгоритма обработки.

Результатом обобщения понятия «тип данных» являются классы объектов (C++) или объектные типы (Pascal), которые могут содержать в качестве элементов не только данные определенного типа, но и методы их обработки (функции и процедуры).

Таким образом, по мере развития технологии программирования и в программах, и в типах данных все адекватнее отражалась структура решаемой прикладной задачи и осуществлялась соответствующая интеграция данных и программ в модулях. Одновременно с этим языки программирования пополнились средствами, необходимыми для описания подобных структур. Развитие идей абстрагирования и модульности привело к появлению в программировании объектного подхода.

Человек мыслит образами или объектами, он знает их свойства и манипулирует ими, сообразуясь с определенными событиями. Еще древним грекам принадлежит мысль о том, что мир можно рассматривать в виде объектов и событий. Рене Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир. Так, по-

думав о телефонном аппарате, человек может представить не только его форму и цвет, но и возможность позвонить, характер звучания звонка и ряд других свойств (в зависимости от его побуждений, технических знаний, фантазии).

Язык программирования позволяет описать свойства моделируемых объектов и порядок манипуляции с объектами или порядок их взаимодействия, сообразуясь с условиями решаемой задачи. Первые языки программирования ориентировались, с одной стороны, на математические объекты, а с другой — на определенную модель вычислителя (ЭВМ с архитектурой фон Неймана). Поэтому они содержали такие конструкции как переменная, константа, процедура, функция, формальные и фактические параметры. Программисты представляли свои программы в виде взаимодействующих подпрограмм (блоков, процедур, функций) и модулей. Характер программирования был процедурно-ориентированным, поскольку первостепенное внимание уделялось последовательностям действий с данными (процедурам). Соответственно такие языки программирования, как FORTRAN, ALGOL, COBOL, PL-1, C называют ***процедурно-ориентированными.***

Данные и подпрограммы объединялись в модули в соответствии с логикой проектировщиков, создающих сложные программные системы для определенных областей их применения. Логика интеграции в модули определялась рядом факторов, среди которых следует отметить свойства предметной области: данные и подпрограммы их обработки, соответствующие определенному классу объектов предметной области, объединялись в модуль. Например, модуль обработки строк содержал подпрограммы выполнения основных операций со строками: объединения, сравнения, копирования, нахождения вхождения подстроки в строку, вычисления длины строки.

Развитием идеи модульного программирования является сопоставление объектам предметной области (моделируемым объектам) программных конструкций, называемых ***объектами***, *объектными типами* или ***классами*** (моделирующими объектами). Моделирующие объекты содержат данные и подпрограммы, которые описывают свойства моделируемых объектов. Так, данные могут отражать признаковые или количественные свойства (масса, длина, мощность, цена, наличие, видимость и т. п.), а подпрограммы отражают поведенческие или операционные свойства (изменить массу, вычислить мощность, установить цену, проверить наличие, сделать видимым и т. п.). Таким образом, при объектном подходе интеграция данных и процедур их обработки определяется структурой предметной области, т.е набором моделируемых объектов, их взаимосвязью или взаимодействием в рамках решаемой задачи.

Моделируемый объект всегда представляется человеку чем-то единым, целостным, хотя может состоять из частей или других объектов. Например, телефонный аппарат состоит из трубки, провода, корпуса с номеронабирателем, а трубка содержит микрофон, динамик и провода и т. д. Целостное представление объекта в виде взаимосвязанной совокупности его свойств или компонентов является базовым принципом объектного подхода.

Объектный подход начал развиваться в программировании со второй половины 60-х годов (язык Simula-67) и нашел свое отражение в языках Smalltalk, CLOS, Ada и в ряде других. Эти языки называются объектными. Иерархическая классификация объектов и наследование свойств являются отправными идеями появившегося в на-

чале 80-х годов объектно-ориентированного подхода. Одной из причин сравнительно медленного становления объектно-ориентированного стиля программирования является его существенное отличие от господствовавшего процедурно-ориентированного стиля.

8.2. Концепции объектно-ориентированного программирования

ООП является третьим крупным этапом (после структурного и модульного программирования) в процессе развития структурного подхода. Создаваемые в середине 70-х годов большие программные системы продемонстрировали, что в рамках процедурно-ориентированного стиля использование структурного подхода не дает желаемого эффекта. По мере увеличения числа компонентов в создаваемых программных системах число ошибок, связанных с неправильным использованием процедур и некорректным учетом взаимосвязей между компонентами, стало нелинейно расти. Сроки ввода в эксплуатацию этих систем постоянно срывались. Уменьшить число подобных ошибок и упростить их обнаружение могла позволить алгоритмическая декомпозиция, ориентирующаяся на «естественные» элементы (компоненты или объекты) пространства решаемой задачи. В этом случае на этапе кодирования и отладки упрощалось сопоставление программистских конструкций с моделируемыми объектами.

Такую декомпозицию будем называть объектно-ориентированным анализом пространства решаемой задачи или предметной области. Для описания результатов объектно-ориентированного анализа и последующего программного синтеза необходимы адекватные языковые средства, которые построены на определенных принципах. Рассмотрим их.

Основным понятием ООП является *объект* или ***класс*** в C++, который можно рассматривать с двух позиций. Во-первых, с позиции предметной области: *класс* соответствует определенному характерному (типичному) объекту этой области. Во-вторых, с позиции технологии программирования, реализующей это соответствие: «класс» в ООП — это определенная программная структура, которая обладает тремя важнейшими свойствами:

- инкапсуляции;
- наследования;
- полиморфизма.

Эти свойства используются программистом, а обеспечиваются объектно-ориентированным языком программирования (транслятором, реализующим это язык). Они позволяют адекватно отражать структуру предметной области.

Некоторые авторы называют данные свойства объектов принципами ООП. Но, по-видимому, это слишком узкое понимание сущности ООП, поскольку в стороне остались отнюдь не детали:

- обьектно-ориентированный анализ предметной области и парадигма ООП;
- создание и уничтожение объектов,
- принципы организации взаимодействия объектов.

Объекты и классы

Концепция объектов предназначена для моделирования (отображения) понятий предметной области в виде программных единиц, объединяющих в себе атрибуты и поведение (состояние и функционирование) соответствующих объектов (сущностей) предметной области.

Класс объектов представляет собой программную структуру, в которой данные и функции образуют единое целое и отражают свойства и поведение этого целого в рамках моделируемой предметной области. В отличие от модуля, в котором на состав данных и функций накладывается меньше смысловых ограничений, в объекте присутствуют только те данные и функции, которые необходимы для описания свойств и поведения объекта определенного типа.

Объекты выделяются в процессе анализа предметной области с использованием идей абстрагирования от несущественного и классификации родственных объектов. Результатом объектно-ориентированного анализа являются ***классы объектов***, которые присутствуют или в перспективе могут присутствовать в пространстве решаемой задачи и образуют ***иерархии классов***, представляемые в виде *деревьев наследования свойств*.

Например, на основе анализа авиационной техники можно выделить класс объектов *Самолет*. При этом мы абстрагировались от таких свойств как: форма крыльев, длина флюзеляжа, используемые материалы конструкций, расположение крыльев и т. п. К числу основных свойств класса объектов *Самолет* можно отнести: скорость полета, размах крыльев, тип двигателя, грузоподъемность, высота полета, функциональное назначение и т. д.

Для примера приведем также иерархию классов *Авиамодели*, которую можно представить в следующем виде:

Авиамодель:

- кордовые модели:
  - гоночные;
  - скоростные;
  - пилотажные;
  - воздушного боя;
  - копии;
- модели свободного полета:
  - двигательные:
    - — таймерные;
    - — радиоуправляемые;
    - — копии;
  - резиномоторные;
  - планерные.

Класс объектов характеризуется уникальным набором свойств и ему присваивается уникальное имя, как и любому типу данных. В качестве переменных программы используются ***объекты*** определенного класса. Создаваемые объекты, даже одного класса, могут отличаться значениями (степенью проявления) свойств и, конечно, должны отличаться именами.

Инкапсуляция свойств объектов

Инкапсуляция (дословно — «содержание в оболочке») представляет собой объединение и локализацию в рамках объекта, как единого целого, данных и функций, обрабатывающих эти данные. В совокупности они отражают свойства объекта.

В C++ данные класса и объекта называются ***элементами данных*** или ***полями***, а функции — ***методами*** или ***элементами-функциями***.

Доступ к полям и методам объекта осуществляется через имя объекта и соответствующие имена полей и методов при помощи операций выбора «.» и «->». Это позволяет в максимальной степени изолировать содержание объекта от внешнего окружения, т. е. ограничить и наглядно контролировать доступ к элементам объекта. В результате замена или модификация полей и методов, инкапсулированных в объект, как правило, не влечет за собой плохо контролируемых последствий для программы в целом. При необходимости указания имени объекта в теле описания этого объекта в C++ используется зарезервированное слово ***this***, которое в рамках объекта является специальным синонимом имени объекта — указателем на объект.

Отметим, что именование классов, элементов данных и методов имеет большое значение в ООП. Названия должны либо совпадать с названиями, использующимися в предметной области, либо ясно отражать смысл или назначение (функциональность) именуемого класса, поля или метода. При этом не следует бояться длинных имен — затраты на написание сторицей окупятся при отладке и сопровождении прот. е..е.ного продукта. Текст подобной программы становится понятным без особых комментариев. Дополнительным средством т. е.е. е.ения доступа к данным и методам является описание элементов классов с помощью спецификаторов ***private***, ***protected*** и ***public***, которые определяют три соответствующих уровня доступа к компонентам класса: собственный, защищенный и общедоступный.

Для расширения доступа к элементам данных, имеющим атрибуты *private* или *protected*, в классе можно реализовать с атрибутом *public* специальные методы доступа к собственным и защищенным элементам данных.

Гибкое разграничение доступа позволяет уменьшить нежелательные (бесконтрольные) искажения свойств объекта или несанкционированное использование свойств классов.

Хорошим стилем в ООП считается организация доступа к элементам данных с помощью функций или методов без использования оператора присваивания. Конечно, это положение не должно являться догмой, но случаи отхода от него должны быть хорошо обдуманы.

Наследование свойств

Наследование есть свойство классов порождать своих потомков и наследовать свойства (элементы данных и методы) своих родителей. Класс-потомок автоматически наследует от родителя все *элементы данных* и *методы*, а также может содержать новые элементы данных и методы и даже заменять (перекрывать, переопределять) методы родителя или модифицировать (дополнять) их. Наследование в ООП позволяет адекватно отражать *родственные отношения*

объектов предметной области. Если класс B обладает всеми свойствами класса A и еще имеет дополнительные свойства, то класс A называется ***базовым*** (родительским), а класс B называется ***наследником*** класса A. В C++ возможно *одиночное* (с одним родителем) и *множественное* (с несколькими родителями) наследование.

Родственные отношения, или ***отношения включения свойств классов***, могут отражаться не только с помощью наследования, но и путем инкапсуляции в классе в качестве элементов данных других классов. Например, стек может быть построен, как наследник одного класса — «элемент стека». Возможно построение стека, как класса, содержащего в качестве элемента данных объект класса «элемент стека».

Свойство наследования упрощает модификацию свойств классов, обеспечивает ООП исключительную гибкость и сокращает затраты на написание новых классов на основе старых (родителей). Программист обычно определяет базовый класс, обладающий наиболее общими свойствами, а затем создает последовательность потомков, которые обладают своими специфическими свойствами. В результате получается иерархия наследования свойств классов.

Базовые классы, или корни подобных деревьев, обладают такими абстрактными свойствами, что часто непосредственно в программах не используются, а необходимы «всего лишь» для порождения требуемых классов. Правильный выбор корня обеспечивает удобство «выращивания» дерева, т. е. простоту развития библиотеки классов Применение правила ООП ***«наследуй и изменяй свойства объектов»*** хорошо согласуется с поэтапным подходом к разработке и созданию больших программных систем.

Примеры родственных классов: Координаты на экране → Цветная точка → Прямая → Прямоугольник. Здесь направление стрелки указывает порядок наследования свойств классов.

Полиморфизм поведенческих свойств объектов

Полиморфизм часто определяют как свойство объектов-родственников по-разному осуществлять однотипные (и даже одинаково поименованные) действия, т. е. однотипные действия у множества родственных классов имеют множество различных форм. Например, метод «нарисовать на экране» должен по-разному реализовываться для родственных классов «точка», «прямая», «ломанная», «прямоугольник». В ООП действия или поведение объекта определяется набором методов. Изменяя алгоритм метода в потомках класса, программист придает наследникам специфические поведенческие свойства. Для изменения метода необходимо перекрыть его в потомке, т. е. объявить в потомке одноименный метод и реализовать в нем нужные действия, отражающие специфику потомка.

Свойство полиморфизма реализуется не только в механизме замещения (перекрытия) одноименных методов при наследовании свойств, но и в механизме виртуализации методов, или позднем связывании методов. Если замещение метода реализуется на этапе компиляции (***раннее связывание*** объекта с методом), то замещет. е. е.етт. е.объявленного в описании класса виртуальным (virtual) происходит на этапе выпол-

нения (***позднее связывание***). Но ничего не дается даром: виртуализация требует дополнительных ресурсов памяти и времени. Виртуальные методы выполняются медленнее, из-за необходимости дополнительных действий по связыванию.

Создание и уничтожение объектов

Описание класса в ООП есть некоторая программная структура, которую используют при создании объектов, отличающихся именами и отдельными свойствами. Создание и удаление объектов осуществляется с помощью специальных методов, которые называются *конструкторами (constructor)* и *деструкторами (destructor)* соответственно.

Конструктор класса создает и инициализирует объекты, а также может выполнять подготовку механизма позднего связывания, необходимого для использования виртуальных функций. ***Деструктор*** класса выполняет действия, завершающие работу с объектом, например: освобождает динамическую память, восстанавливает экран, закрывает файловые переменные.

Объекты можно размещать и в динамической памяти. Для этого необходимы указатели на эти объекты.

Взаимодействие объектов и сообщения

Отношение родства или включения свойств — это только один из многих типов отношений между классами. Еще одним важнейшим типом отношений между классами и между объектами являются отношения взаимодействия или отношения «клиент-сервер». Отношения родства также подразумевают определенное отношение взаимодействия, ограниченное свойством инкапсуляции.

Благодаря инкапсуляции объекты так хорошо изолируются друг от друга, что необходимо специально заботится о механизме их взаимодействия в программе. Это касается, в первую очередь, независимых объектов, а не тех, которые погружены в другие объекты в виде элементов данных этих объектов. Но и в последнем случае взаимодействие может быть ограниченно синтаксисом языка и соглашениями о видимости (доступности) элементов данных и методов родственных объектов и инкапсулированных объектов. Поэтому для его расширения требуется специально заботиться об этом.

Часто связь устанавливают при помощи ***указателей***, что делает программу похожей на структуру данных в динамической памяти. Примерами такого способа являются программы, написанные с помощью библиотек Turbo Vision и Object Windows. Основным содержанием программы являются описание классов и совокупности взаимодействующих объектов. В библиотеках классов для доступа объекта к самому себе (внутри себя) используется зарезервированное слово ***this***, являющееся синонимом имени объекта, в контексте которого оно используется.

Вторым способом обмена информацией между объектами является передача через ***глобальную переменную*** (буфер сообщений), которой один объект присваивает значение, а другой — считывает. Данный способ прост в реализации, но требует тщательного контроля обмена сообщениями со стороны программиста,

поскольку структура взаимодействия более скрыта от программиста, чем при первом способе.

В ООП термин «послать объекту сообщение» часто трактуют, как вызов метода объекта, которому посылается сообщение, определяемое данным вызовом.

8.3. Этапы объектно-ориентированного программирования

Программа, решающая некоторую задачу, заключает в себе описание части мира, относящегося к этой задаче, или определенной предметной области. Описание действительности в форме взаимодействующих объектов, по-видимому, естественнее, чем в форме иерархии подпрограмм, и поэтому облегчает программное моделирование в некоторой предметной области.

В процессе программирования в объектно-ориентированном стиле можно выделить следующие этапы:

1. Определение основных понятий предметной области и соответствующих им классов, имеющих определенные свойства (возможные состояния и действия). Обоснование вариантов создания объектов.

2. Определение или формулирование принципов взаимодействия классов и взаимодействия объектов в рамках программной системы.

3. Установление иерархии взаимосвязи свойств родственных классов.

4. Реализация иерархии классов с помощью механизмов инкапсуляции, наследования и полиморфизма.

5. Для каждого класса реализация полного набора методов для управления свойствами объектов.

В результате будет создана объектно-ориентированная среда, или библиотека классов, позволяющая решать задачи моделирования в определенной предметной области.

Первые три этапа и являются объектно-ориентированным анализом предметной области.

В заключение раздела отметим, что ООП обладает следующими достоинствами:

- использование более естественных понятий и имен из повседневной жизни или предметной области;
- простота введения новых понятий на основе старых;
- отображение в библиотеке классов наиболее общих свойств и отношений между объектами моделируемой предметной области;
- естественность отображения пространства решаемой задачи в пространство объектов программы;
- простота внесения изменений в классы, объекты и программу в целом;
- полиморфизм классов упрощает составление и понимание программ;
- локализация свойств и поведения на основе инкапсуляции и разграничения доступа упрощают понимание структуры программы и ее отладку;
- передача параметров в конструкторах объектов повышает гибкость создаваемых программных систем.

В качестве недостатков ООП можно отметить следующие:

• снижение быстродействия программ, особенно при использовании виртуальных методов;

• большие затраты времени на разработку библиотеки классов, поэтому ООП целесообразнот. е.. е.ятьт. е.роизводстве ПО или при создании больших программ, а не при написании маленьких единичных программ;

• необходимость анализа полной иерархии классов для правильного использования свойств базовых классов.

Целесообразность применения технологии ООП для создания конкретной программной системы определяется двумя главными факторами:

• спецификой предметной области и решаемой прикладной задачи;

• особенностями используемого языка программирования (транслятора).

Предметная область может быть в большей или меньшей степени разработанной для использования технологии ООП. В разработанных областях ясны классы, их иерархия и отношения взаимодействия. В неразработанных областях использование ООП предусматривает гораздо большие затраты на первоначальных этапах разработки (объектно-ориентированный анализ, проектирование и создание библиотеки классов).

Известны области, в которых требуется максимальная эффективность программного кода в смысле быстродействия и затрат памяти (системы реального времени, сверхминиатюрные средства). В таких областях применение ООП может быть оправдано или снижением стоимости разработки ПП, или удовлетворительным качеством генерируемого программного кода.

Контрольные вопросы и задания

1. Назовите основные черты эволюции структурного подхода в программировании.

2. Укажите важнейшие факторы, влияющие на эволюцию методов проектирования и создания программных продуктов.

3. Каковы различия модульного и объектно-ориентированного программирования?

4. Укажите основной принцип понижения сложности программ.

5. Назовите основные различия процедурно-ориентированного и объектно-ориентированного стилей программирования.

6. Укажите, для чего предназначена типизация данных.

Что дает использование структурных типов данных?

7. Перечислите этапы разработки программ в объектно-ориентированном стиле.

8. Охарактеризуйте свойства инкапсуляции.

9. Каким образом осуществляется доступ к элементам данных и методам класса или объекта?

10. Каким образом устанавливаются дополнительные ограничения к данным и методам классов?

11. Каково назначение зарезервированного слова *this*?

12. Как соотносятся между собой и что означают понятия *класс объекта* и *объект*?

13. Дайте определение полиморфизма.

14. Назовите достоинства применения ООП.
15. Какими факторами определяется целесообразность применения технологии ООП?
16. С помощью каких методов выполняется создание и удаление объектов?
17. Назовите способы установления связи и обмена информацией между объектами.
18. В чем суть наследования?

Какие разновидности наследования имеются в языках программирования?

19. Предложите вариант иерархии классов в области транспортных средств.
20. Предложите вариант иерархии классов в области вычислительной техники.
21. Предложите вариант иерархии классов в животном мире.

9. КЛАССЫ И ИНКАПСУЛЯЦИЯ

В данном разделе рассматривается описание и использование классов в C++ и инкапсуляция как одно из важнейших свойств классов. ***Инкапсуляция*** представляет собой объединение данных и обрабатывающих их функций в одном классе как типе объектов. Целью инкапсуляции является автономность модулей, что позволяет локализовать последствия изменения структур данных конкретного модуля. Для остальных модулей изменения будут незаметны. Инкапсуляция при описании класса обеспечивается заданием спецификаторов доступа для компонентов класса.

9.1. Описание класса

Класс представляет собой абстрактный тип (определяемый программистом), который создается на основе существующих типов. Отдельный класс включт. е.т. е.бя дт. е.е, называемые *элементами данных,* и функции, называемые *методами.* Элементы данных и методы являются равноправными компонентами класса.

Описание класса имеет следующий формат:

```
class | struct | union  имя_класса {список_компонентов};
```

В этом описании:

- одно из ключевых слов *class, struct* или *union* указывает на начало описания класса, определяет используемый по умолчанию статус доступа к компонентам класса, а также влияет на возможности наследования свойств этого класса;
- *имя_класса* — идентификатор;
- *список_компонентов* — перечень объявлений элементов данных и описаний методов класса.

В соответствии с синтаксисом языка C++ каждый компонент класса обладает статусом доступа. Таких статуса три: общедоступный, собственный и защищенный. В качестве ***спецификаторов доступа*** используются ключевые слова ***public*** (общедоступный), ***private*** (собственный), ***protected*** (защищенный), за которыми следует двоеточие. Действие спецификатора на компоненты класса начинается с момента его написания до нового спецификатора или до конца описания класса.

Спецификатор доступа *private* используется в основном для задания статуса доступа к элементам данных класса, что позволяет решить проблему защиты данных. Собственные данные являются доступными только для методов своего класса. Спецификатор доступа *public* часто используется для задания общедоступного доступа методам класса, которые организуют связь объекта данного класса с внешним миром. Статус защищенный *(protected)* используется в классах при применении механизма наследования классов. При отсутствии наследования спецификатор *protected* эквивалентен спецификатору *private*.

Все компоненты класса, введенные с помощью ключевых слов ***struct*** и ***union***, являются по умолчанию *общедоступными*, а с помощью ключевого слова ***class*** — *собственными*, т. е. недоступными для обращений извне. Для изменения статуса компонентов классов, описанных с помощью ключевых слов *class* и *struct*, необходимо использовать спецификаторы доступа. Классы, описанные с помощью ключевого слова *union*, не могут использоваться в качестве базовых классов при наследовании. Кроме того, у объектов, объявленных на основе подобного класса, для элементов данных выделяется общее место в памяти. Статус компонентов у таких классов изменить нельзя.

Пример 1. Описание класса.

Рассмотрим описание класса *Sum*, который обеспечивает суммирование двух целых чисел. Компонентами класса являются: два слагаемых x и y, сумма s и методы *getx()*, *gety()*, *summa()*, которые предназначены для инициализации компонентных данных x и y, а также для получения и вывода на экран компьютера результата.

Для того чтобы в классе *Sum* элементы данных определить *собственными*, а методы *общедоступными*, описание класса можно записать следующим образом:

```
class Sum
  {
    int x,y,s;                    // по умолчанию private
    public:
    void getx(int x1) { x=x1; } // описание метода
    void gety(int y1) { y=y1; } // описание метода
    void summa();                 // прототип метода
  };
// Описание метода:
void Sum :: summa()
  {
    s=x+y;
    cout << "\n Сумма "<< x << " и " << y << " равна:" << s;
  }
```

В приведенном классе *Sum* компонентные данные x, y и s являются *собственными* по умолчанию, а методы *getx()*, *gety()* и *summa()* *общедоступными*.

В описании класса методы *getx()* и *gety()* представлены полностью, а метод *summa()* — своим прототипом. Методы *getx()* и *gety()* обеспечивают ввод компонентных данных x и y соответственно, так как доступ к элементам данных класса можно обеспечить

только с помощью методов класса *Sum*. Другим функциям компонентные данные недоступны, т. к. данные *x* и *y* имеют статус доступа *private*.

Описания методов *getx()* и *gety()* размещены внутри класса. Такая форма описания делает метод встроенным (*inline*) по умолчанию. В этом случае тело метода будет размещено в самом классе в виде макрорасширения. Этим достигается экономия времени реализации метода при вызове функции и выходе из нее. Эту форму описания следует использовать лишь для небольших функций. Второй способ описания метода заключается в том, что внутри класса записывается прототип, а описание метода размещается в произвольном месте программы вне тела класса. В приведенном примере таким образом описан метод *summa()*.

9.2. Создание и использование объектов

После описания класса можно объявить один или несколько объектов как экземпляр этого класса. Для приведенного примера объявление объектов будет выглядеть следующим образом:

```
Sum k, z;
```

Здесь объявлены объекты *k* и *z* типа *Sum*.

Для доступа к компонентам объекта можно использовать два способа указания имени объекта:

1. Непосредственное указание имени объекта.
2. Косвенное задание имени объекта с помощью указателя и операции косвенного выбора (->).

При первом способе задания обращение к компонентам объекта имеет следующий формат:

```
имя_объекта . имя_класса :: обращение_к_компоненту
```

Например, запись *k.Sum* :: *x* означает обращение к компонентному данному *x* объекта *k* типа (класса) *Sum*.

Обращение к компонентам объекта в ряде случаев можно выполнить без указания имени класса, к которому принадлежит объект, в следующем формате:

```
имя_объекта . обращение_к_компоненту
```

Такая форма записи возможна в случае однозначности определения принадлежности объекта классу. Неоднозначность возникает, если имеются методы с одинаковыми именами, описанные в разных базовых классах.

Например: *k.x*, *k.y* и *k.s* представляют собой обращения к элементам данных *x*, *y* и *s* объекта *k*; *k.getx(3)* есть вызов метода *getx()* объекта *k* с аргументом *3*.

При использовании объекта *k* обращение *k.getx(8)*; к методу *getx()* осуществляет присвоение элементу данных *x* значение *8* и обращение *k.gety(5)*; к методу *gety()* осуществляет присвоение значения *5* элементу данных *y*.

При втором способе указания имени объекта для доступа к компонентам объекта используется следующий формат:

```
указатель_на_объект_класса -> обращение_к_компоненту_объекта
```

Если объявить и инициализировать указатель *b* следующим образом: *Sum *b=&z;*, то для объекта *z* выражения, позволяющие обратиться к элементам данных *x* и *y*, будут выглядеть следующим образом: *b –> x* и *b –> y*. Вызов метода для присвоения значения *6* элементу данных *x* объекта *z* можно записать таким образом: *b –> getx(6);*. Теперь можно написать программу с использованием класса.

Пример 1. Создание и использование объектов.

Пусть требуется составить программу, выполняющую суммирование двух произвольных чисел.

```
// Сумма двух целых чисел
#include <iostream.h>
class Sum
  {
    int x,y,s;                    // по умолчанию private
    public:
    void getx(int x1) { x=x1; } // описание метода
    void gety(int y1) { y=y1; } // описание метода
    void summa();                 // прототип метода:
  };
void Sum :: summa()
  {
    s=x+y;
    cout << "\n Сумма "<< x << " и " << y << " равна:" << s;
  }
void main()
  {
    Sum z,*b=&z;
    int x2,y2;
    cout << "\n Введите первое слагаемое:";
    cin >> x2;
    cout << "\n Введите второе слагаемое:";
    cin >> y2;
    z.getx(x2);
    z.gety(y2);
    b->summa();
    // cout << "\n Сумма "<< z.x << " и " << z.y << " равна:" << z.s;
  }
```

В данной программе введен тип *Sum*, компонентами которого являются три элемента данных *x*, *y*, *s* и три метода *putx()*, *puty()*, *summa()*. В функции *main()* объявлены объект *z* типа *Sum* и указатель *b* на объекты типа *Sum*, инициализированный адресом объекта *z*. Обращения к методам объекта *z*: *z.getx(x2)*; и *z.gety(y2)*; присваивают значения *x2* и *y2* элементам данных *x* и *y* объекта *z* соответственно. Вызов *b–>summa()*; метода *summa()* выполняет вычисление суммы значений элементов *x* и *y* объекта *z* и вывод результата на экран.

В предпоследней строке записан в качестве комментария оператор, позволяющий выдать на экран информацию о значении данных объекта *z* в случае, если доступ к компо-

нентным данным класса *Sum* будет изменен на *public*. В настоящий момент с этими данными могут работать только методы своего класса, т. к. статус доступа у данных *private*.

9.3. Конструкторы и деструкторы

В языке C++ определены два специальных типа методов — конструкторы и деструкторы. ***Конструкторы*** предназначены для инициализации элементов данных объектов. Описание конструктора имеет следующий формат:

```
имя_ конструктора (список_ параметров)  {тело_ конструктора}
```

Имя конструктора должно совпадать с именем класса. Для конструктора не указывается тип возвращаемого значения (даже *void* недопустим).

Для класса *Sum* описание конструктора может выглядеть следующим образом:

```
Sum (int x2=0, int y2)
  {
    x=x2;
    y=y2;
  }
```

Конструктор может иметь значения параметров по умолчанию, которые задаются в списке параметров. В приведенном описании конструктора *Sum* параметру *x2* задано значение 0 по умолчанию.

В вызове конструктора по сравнению с другими методами существуют следующие особенности:

- если конструктор не вызывается явно, то он будет вызван автоматически при создании объекта с использованием значений параметров по умолчанию;
- если конструктор не описан явно, то он генерируется транслятором без участия программиста.

Для передачи значений элементам данных объекта с помощью конструктора существует два формата записи. Первый из них:

```
имя_ класса  имя_ объекта = имя_ конструктора (список_ аргументов);
```

Второй, более короткий, формат вызова конструктора имеет вид:

```
имя_ конструктора имя_объекта (список_ аргументов);
```

В обоих случаях осуществляется создание объекта указанного класса и инициализация его элементов данных. Возможность второго варианта объясняется тем, что имя конструктора совпадает с именем класса.

Например, конструктор *Sum()* можно вызвать двумя способами: *Sum A=Sum(1,2)*; и *Sum A(1,2)*. В обоих случаях создается объект *A*, элементы данных *x* и *y* которого получают начальные значения *1* и *2* соответственно. Для обсуждаемого примера вызов конструктора можно осуществить без указания первого аргумента, т. е. *Sum A(,2)*;. В этом случае элементу данных *x* будет присвоено нулевое значение по умолчанию (*x2*=0). В классе могут быть несколько конструкторов.

Деструкторы уничтожают объекты класса и освобождают занимаемую этими объектами память. Деструктор представляет собой метод с именем, совпадающим с именем класса, перед которым стоит символ тильда (~). Деструктор не должен иметь ни параметров, ни типа возвращаемого значения. Описание деструктора имеет следующий формат:

```
~имя класса( ) {операторы_тела_деструктора}
```

Например, для класса *Sum* описание деструктора выглядит следующим образом:

```
~Sum() {}
```

Деструктор вызывается *явно* или *неявно*. Деструктор вызывается ***явно*** (как обычный вызов функции) при необходимости уничтожения объекта. Вызов деструктора выполняется ***неявно*** (автоматически) для локального объекта тогда, когда перестает быть активным блок, в котором локальный объект объявлен. Если значения указателей объектов выходят за пределы области действия объявления объекта, то неявный вызов деструктора не происходит, а для разрушения такого объекта необходимо явным образом выполнить операцию *delete*.

Пример 1. Использование конструктора и деструктора.

Пусть требуется составить программу, реализующую вычисления по следующей формуле: $s=a\times b+c\times k+a\times c$.

```
#include <iostream.h>
struct Pro
  {
    private:
    int x,y,z;
    public:
    // Прототипы методов:
    Pro (int,int);          // конструктор
    int putx();             // доступ к x
    int puty();             // доступ к y
    int putz();             // доступ к z
    void proizv();          // произведение
    ~Pro();                 // деструктор
  };
// Описания методов:
Pro :: Pro(int x1,int y1) { x=x1; y=y1;}
int  Pro :: putx() { return x;}
int  Pro :: puty() { return y;}
int  Pro :: putz() { return z;}
void Pro :: proizv() {z=x*y;}
Pro :: ~Pro() { }
void main()
  {
    int s,a,b,c,k;
    cout << "\n Введите a,b,c и k\n";
```

```
    cin >> a >> b >> c >> k;
    Pro D = Pro(a,b);        //создание и инициализация объекта D
    Pro E(c,k);              //создание и инициализация объекта E
    Pro F(a,c);              //создание и инициализация объекта F
    D.proizv();              //получение произведения a*b
    E.proizv();              //получение произведения c*k
    F.proizv();              //получение произведения a*c
    cout << "\n D.a=" << D.putx();
    cout << "\t D.b=" << D.puty();
    cout << "\t D.z=" << D.putz();
    s=D.putz()+E.putz()+F.putz();
    cout << "\n s=" << s;
    F.Pro :: ~Pro();         //уничтожение объекта F
    E.Pro :: ~Pro();         //уничтожение объекта E
    D.Pro :: ~Pro();         //уничтожение объекта D
}
```

В примере для получения произведения создан класс *Pro*, компонентными данными которого являются сомножители *x* и *y*, а также элемент *z*, предназначенный для хранения произведения. Для доступа к компонентным данным используются три метода *putx()*, *puty()* и *putz()*, а для получения произведения *proizv()*. Кроме того, для класса определены конструктор *Pro()* и деструктор *~Pro ()* — в классе содержатся их прототипы.

Описания всех методов, в том числе конструктора и деструктора, вынесены за пределы описания класса. В результате трех явных вызовов конструктора созданы и инициализированы объекты *D, E и F*. В результате работы метода *proizv()* получены три значения произведений для своих объектов. Результатом работы программы является получение суммы этих значений. В конце программы три раза вызван деструктор для уничтожения созданных ранее объектов.

9.4. Пример создания и использования класса

В качестве комплексного примера рассмотрим класс *Point*, который позволяет сформулировать точку на экране компьютера. Поместим описание класса в файл с именем *point.h*:

```
// Файл point.h
#ifndef POINT_H
#define POINT_H 1
class Point                  //класс для определения точки на экране
  {
    protected:               //защищенный статус доступа к элементам данных
    int x;                   //координата x точки
    int y;                   //координата y точки
  // Прототипы методов:
    public:                  //общедоступный статус доступа
```

```
    Point (int, int);      //конструктор
    int putx();            //доступ к x
    int puty();            //доступ к y
    void show();           //изобразить точку на экране
    void move (int,int);   //переместить точку
   private:                //собственный статус доступа
    void hide();           //убрать изображение точки
  };
#endif
```

Поскольку описание класса *Point* планируется использовать при описании других классов, то для предотвращения недопустимого дублирования описания класса в текст включены три директивы препроцессора *#ifndef POINT_H*, *#define POINT_H 1* и *#endif*.

Компонентами класса *Point* являются два элемента данных *x* и *y* с защищенным статусом доступа, пять общедоступных методов и один метод с собственным статусом доступа. Методы в описании класса представлены своими прототипами.

Выполним внешнее описание методов класса, разместив описания в файле *point.cpp*:

```
//Файл point.cpp – описание методов
#ifndef POINT_CPP
#define POINT_CPP 1
#include <graphics.h>// прототипы функций графической библиотеки
#include "f:\POS_C\PRIMER\point.h"    // описание класса Point
// Конструктор:
Point :: Point (int x1=0, int y1=0)
  {
    x=x1;
    y=y1;
  }
//Метод доступа к x:
int Point :: putx()
  {
    return x;
  }
//Метод доступ к y:
int Point :: puty()
  {
    return y;
  }
//Метод изображения точки на экране:
void Point :: show(void)
  {
    putpixel(x,y,getcolor());
  }
```

```
// Метод удаления точки с экрана:
void Point :: hide(void)
  {
    putpixel(x,y,getbkcolor());
  }
// Метод перемещения точки в новое место экрана:
void Point :: move(int xn=0, int yn=0)
  {
    hide();
    x=xn;
    y=yn;
    show();
  }
#endif
```

Для получения изображения точки на экране в программу должен быть включен заголовочный файл *graphics.h. В* данном файле находятся прототипы графических функций.

***Достоинство** внешнего описания методов класса* состоит в том, что оно позволяет при необходимости модифицировать содержание методов, причем эти изменения останутся незамеченными для остальных частей программы.

Программа, содержащая в своем составе главную функцию, будет выглядеть следующим образом:

```
// Точка на экране
#include <iostream.h>
#include <graphics.h>      //прототипы графических функций
#include <conio.h>         //прототип функции getch()
#include "f:\POS_C\PRIMER\point.cpp"  //описание класса Point

int main()
  {
    Point t(100,150);      //создана невидимая точка t(x=100,y=150)
   Point t1(200,200);      //создана невидимая точка t1(X=200,Y=200)
    // Инициализация графики
   int a=DETECT,b;
    initgraph(&a,&b,"F:\\BC5\\bgi");

    t.show();              // показать точку t
    cout << "\n коор-ты точки t: x=" << t.putx() << ",y=" << t.puty();
   getch();                //ждать нажатия клавиши
   t1.show();              //показать тоочку t1
   cout << "\n координаты точки t1: x=" << t1.putx() << ",y=" <<
        t1.puty();
   getch();
   t.move(150,300);        //переместить точку t(x=150,y=300)
```

```
    cout << "\n новые координаты t: x=" << t.putx() << ",y=" <<
         t.puty();
    getch();
    closegraph();          //закрыть графический режим
}
```

В программе используется функция *getch()*, которая позволяет остановить выполнение программы до нажатия на клавиатуре любой клавиши. Прототип этой функции находится в файле *conio.h*.

Контрольные вопросы и задания

1. Какова цель инкапсуляции?
2. Дайте определение понятию «класс».
3. Укажите формат описания класса.
4. Что такое метод?
5. Назовите статусы доступа к компонентам класса.
6. Какую область имеют действия спецификаторов на компоненты класса?
7. Укажите различие между компонентами класса, объявленными с помощью ключевых слов *strukt*, *union*, *class*.
8. Какие функции имеют доступ к собственным элементам данных класса?
9. Назовите варианты размещения описаний методов класса.
10. В чем заключается достоинство внешнего описания методов класса?
11. Каким образом можно обратиться к элементам данных объекта?
12. В каком случае обращение к компанентам объекта можновыполнить без указания имени класса?
13. В чем назначение конструктора?
14. В чем особенности вызова конструктора по сравнению с вызовом других методов?
15. Каково назначение деструктора?
16. Укажите формат описания деструктора.
17. Назовите варианты вызова деструктора.
18. Всегда ли вызывается конструктор при создании объекта?
19. Измените описанный выше класс *Point* таким образом, чтобы элементы данных x и y стали общедоступными.
20. Для класса *Point* создайте конструктор, у которого отсутствуют параметры по умолчанию.
21. Для измененного класса *Point*, у которого общедоступные данные, создайте объект этого класса и организуйте обращение к элементам данных x и y, не используя методы класса *Point*.
22. Составьте программу игры в морской бой. При этом предусмотреть 5 классов кораблей и класс поля боя для размещения кораблей одной из сторон.
23. Составьте программу, моделирующую поиск мышки кошкой в комнате с предметами. Педусмотреть классы подвижных и неподвижных объектов разных размеров.

10. НАСЛЕДОВАНИЕ

Обычно классы не существуют сами по себе. Они возникают с введением понятия *объект*, как тип этого объекта. Каждт. е.от. е.ный обът. е.взаимодействует с другими частями программы с помощью сообщений. В ответ на переданное ему сообщение *объект* выполняет такие действия, которые заложены в методах того класса, к которому принадлежит объект. Такими действиями могут быть либо изменения внутреннего состояния объекта, т. е. изменения компонентных данных объекта, либо передача сообщений другим частям программы.

Объекты разных классов и сами классы могут находиться в отношении *наследования*, в соответствии с которым формируется иерархия объектов и иерархия классов соответственно.

Механизм наследования позволяет определять новые классы на основе уже имеющихся. Класс, на основе которого создается новый класс, называют ***базовым (родительским)*** классом, а новый — ***производным (наследником)***. ***Непосредственным базовым классом*** называется такой класс, который входит в список базовых классов при определении класса. Любой производный класс может в свою очередь стать базовым для других создаваемых классов. Таким образом, формируется направленный граф ***иерархии классов***, а при объявлении объектов и ***иерархия объектов***. В иерархии объектов производный объект имеет возможность доступа к элементам данных и методам объектов, типизированных базовым классом.

В языке существует возможность одиночного и множественного наследования. При ***одиночном наследовании*** базовым является один класс, а при ***множественном наследовании*** базовыми классами должны быть несколько классов.

10.1. Управление доступом производных классов

При наследовании важную роль играет статус доступа к компонентам класса. Напомним, что в иерархии классов используются следующие соглашения о правах доступа к компонентам классов:

- *собственные* (*private*) компоненты доступны только внутри того класса, где они определены;
- *защищенные* (*protected*) методы и элементы данных доступны внутри класса, в котором они определены, и во всех производных классах;
- *общедоступные* (*public*) компоненты класса видимы из любой точки программы.

Таким образом, для *объекта*, который *обменивается сообщениями* с другими объектами и обрабатывает их, доступными являются:

- общедоступные компоненты всех объектов программы;
- защищенные данные и методы объектов, являющиеся представителями базовых классов;
- собственные компоненты объекта.

При описании производного класса можно изменить статус доступа к наследуемым компонентам класса с помощью ***модификатора статуса доступа***. Формат описания производного класса выглядит следующим образом:

```
class | struct имя_производного_класса: [модификатор] имя_базового_класса
{компоненты_класса};
```

В качестве модификатора статуса доступа используются ключевые слова ***private***, ***protected***, ***public***. В табл. 10.1 приводятся значения статуса доступа к компонентам производного класса в зависимости от статуса доступа к компонентам базового класса и значения модификатора статуса доступа.

Таблица 10.1

Статусы доступа производных классов

Статус доступа в базовом классе	Модификатор доступа	Статус доступа в производном классе	
		struct	class
public	–	public	private
protected	–	public	private
private	–	недоступны	недоступны
public	public	public	public
protected	public	protected	protected
private	public	недоступны	недоступны
public	protected	protected	protected
protected	protected	protected	protected
private	protected	недоступны	недоступны
public	private	private	private
protected	private	private	private
private	private	недоступны	недоступны

Из таблицы видно, что в производных классах статус доступа к компонентам класса может быть только ужесточен. Во избежание ошибок целесообразно всегда явно указывать статус доступа для каждого компонента класса вне зависимости от установок по умолчанию.

10.2. Одиночное наследование

Напомним, что при одиночном наследовании базовым для производного класса служит один класс. Формат описаия производного класса при одиночном наследовании приведен в предыдущем подразделе. Рассмотрим пример одиночного наследования.

Пример 1. Одиночное наследование классов.

Требуется составить программу, которая позволяет получить на экране окружность. Для иллюстрации механизма наследования на основе класса *Point* построим производный класс *Circle* (окружность). Для производного класса из класса *Point* выберем следующие элементы данных и методы:

- *int x* — координата *x* точки;
- *int y* — координата *y* точки;
- *int putx()* — доступ к *x*;
- *int puty()* — доступ к *y*.

Дополнительно для класса *Circle* введем следующие компоненты:

- *int radius* — радиус окружности;
- *int vis* — индикатор видимости окружности на экране;
- *void show()* — изобразить окружность на экране;
- *void hide()* — убрать изображение окружности;
- *void move()* — переместить окружность на новое место экрана;
- *void vary()* — изменить размер окружности;
- *int putradius()* — обеспечить доступ к радиусу окружности.

В соответствии с изложенным создадим класс *Circle* и его описание поместим в файл *circle.h*:

```
// Файл circle.h - описание производного класса
#include "f:\POS_C\PRIMER\point.cpp" // описание класса Point

class Circle: public Point      //класс для определения окружности
  {  // модификатор public позволяет сделать элементы x и y класса
     // Point в описании класса Circle защищенными (protected)
    protected:              //защищенный статус доступа к элементам данных
     int radius;                //радиус окружности
     int vis;                   //видимость окружности на экране
  //Прототипы методов:
     public:                    //общедоступный статус доступа
     Circle(int,int,int);       //конструктор
     ~Circle();                 //деструктор
     void show();               //изобразить окружность на экране
     void hide();               //убрать изображение окружности
     void move (int,int);       //переместить окружность
     void vary(int);            //изменить размер окружности
    int putradius();            // доступ к радиусу
  };
```

В созданном классе *Circle* явно определены конструктор *Circle()* и деструктор *~Circle()*. Из класса *Point* наследуются два метода *putx()* и *puty()*, так как методы *show()* и *move()* заменяются одноименными методами класса *Circle*, а метод *hide()* не наследуется, поскольку имеет статус доступа *private*.

Используем внешние описания методов, поместив их в отдельный файл с именем *circle.cpp*:

```
//Файл circle.cpp
#include <graphics.h> // прототипы функций графической библиотеки
#include "f:\POS_C\PRIMER\circle.h" // описание класса Circle
//Описания методов класса Circle:
//Конструктор
Circle :: Circle (int xc,int yc,int radc=0) : Point(xc,yc)
  {
    radius=radc;
    vis=1;
  }
//Изобразить окружность на экране
void Circle :: show(void)
  {
    circle(x,y,radius);     //рисование окружности
  }
//Убрать окружность с экрана
 void Circle :: hide(void)
  {
    unsigned int col;       //объявление переменной для текущего цвета
    if (vis==0) return;     //окружности нет
    col=getcolor();         //запоминание текущего цвета
    setcolor(getbkcolor());      //выявление текущего цвета фона
    circle(x,y,radius);     //рисование окружности цветом фона
    vis=0;
    setcolor(col);          //восстановление текущего цвета
  }
  // Переместить окружность в новое место экрана
 void Circle :: move(int xn, int yn)
  {
    hide();                 //стирание старой окружности
    x=xn;
    y=yn;
    show();                 //рисование новой окружности
  }
   // Изменить размер окружности
 void Circle :: vary(int d)
  {
    hide();                 //стирание старой окружности
    radius+=d;              //изменение радиуса
```

```
    if(radius<0) radius=0;
    show();                    //рисование новой окружности
  }
    // Доступ к радиусу
 int Circle :: putradius()
  {
    return radius;
  }
  // Деструктор
 Circle :: ~Circle()
  {
     hide();                   //стирание окружности
  }
```

Конструктор *Circle()* имеет три параметра: координаты центра *(xc, yc)* и радиус окружности *radc*. При создании объекта класса *Circle* вначале вызывается конструктор класса *Point*, который по значениям фактических параметров определяет центр окружности. Эта точка создается как объект класса *Point* без имени. Затем выполняется тело конструктора *Circle()*.

Деструктор *~Circle()* уничтожает окружность и вызывает деструктор базового класса *~Point()*, который, несмотря на отсутствие его описания в классе, формируется компилятором по умолчанию.

Теперь можно составить программу для работы с объектами класса *Circle*.

```
#include <iostream.h>
#include <graphics.h>              //прототипы графических функций
#include <conio.h>                 //прототип функции getch()
#include "f:\POS_C\PRIMER\circle.cpp"  //описание класса Circle
int main()
  {
    // Инициализация графики
    int a=DETECT,b;
    initgraph(&a,&b,"F:\\BC5\\bgi");

    Circle A(150,250,30);          //создана невидимая окружность A
    Circle B(300,100,50);          //создана невидимая окружность B
    cout<<"\n A: x="<<A.putx()<<", y="<<A.puty()<<",
      радиус="<<A.putradius();
    cout<<"\n B: x="<<B.putx()<<", y="<<B.puty()<<",
      радиус="<<B.putradius();
    getch();                       //ждать нажатия клавиши
   A.show();                       //показать на экране окружность A
   getch();                        //ждать нажатия клавиши
   B.show();                       //показать на экране окружность B
   getch();                        //ждать нажатия клавиши
   Point C(300,100);               //создана точка C
```

```
    C.show();                    // показать на экране точку C
    getch();                     // ждать нажатия клавиши
    A.move(150,150);             // переместить окружность A
    cout<<"\n A: x="<<A.putx()<<", y="<<A.puty()<<",
      радиус="<<A.putradius();
    getch();
    B.vary(20);                  // изменить размеры окружности B
    cout<<"\n B: x="<<B.putx()<<", y="<<B.puty()<<",
      радиус="<<B.putradius();
    getch();
    A.Circle :: ~Circle();       // уничтожение объекта A
    B.Circle :: ~Circle();       // уничтожение объекта B
    closegraph();                // закрыть графический режим
}
```

В результате выполнения программы вначале конструктором *Circle()* создаются два невидимых объекта: окружность *A* с координатами (150, 250) и радиусом 30 и окружность *B* с координатами (300, 100) и радиусом 50. В результате двукратной работы метода *show()* объекты *A* и *B* становятся видимыми. При явном вызове конструктора *Point()* создается объект *C* с координатами центра окружности *B*. В результате работы метода *show()* класса *Point* точка *C* становится видимой. Метод *move()* стирает старую окружность *A*, изменяет в объекте *A* значения защищенных координат объекта (150, 150) (функции, которые не принадлежат к классам *Point* и *Circle*, этого сделать не могут) и рисует окружность прежнего радиуса на новом месте экрана. Метод *vary()* стирает окружность объекта *B* с экрана (при этом объект *B* не уничтожается) и после изменения значения радиуса рисует окружность с другим радиусом.

10.3. Множественное наследование

Множественное наследование представляет собой такое наследование, при котором создание производного класса основывается на использовании нескольких непосредственных базовых классов. При множественном наследовании используется следующий формат описания класса:

```
class | struct имя_производного_класса: [модификатор] имя_базового_класса_1
,.., [модификатор] имя_базового_класса_n
{ компоненты_класса };
```

В качестве примера множественного наследования рассмотрим производный класс *Reccircle* — «прямоугольник и окружность». Для описания класса *Reccircle* создадим новый непосредственный базовый класс *Rectangle* — «прямоугольник» и используем ранее созданный класс *Circle* — «окружность». Описание класса *Rectangle* поместим в отдельный файл *rectang.h*:

```
// Файл rectang.h - описание класса прямоугольник
#include <graphics.h>  //прототипы функций графической библиотеки
```

```
#include "f:\POS_C\PRIMER\point.cpp" //описание класса Point

class Rectangle: public Point          //класс для определения прямоугольника
  {
   protected              //защищенный статус доступа к элементам данных
   int lx;                //длина по горизонтали
   int ly;                //длина по вертикали
   // Прототипы методов:
   void risrec();         //прорисовка прямоугольника
   public:                //общедоступный статус доступа
   Rectangle(int,int,int,int)       // конструктор
   void show(void);       //прорисовка прямоугольника
   void hide(void);       //убрать изображение прямоугольника с экрана
 };
```

Непосредственным базовым классом для класса *Rectangle* является класс *Point*. Кроме наследуемых элементов данных *x* и *y* (координаты центра прямоугольника) класс *Rectangle* в своем составе имеет два элемента данных *lx* — длина по горизонтали и *ly* — длина по вертикали. Четыре метода класса позволяют создавать объекты, их инициализировать, получать изображение на экране и убирать его в случае необходимости. Описания методов класса *Rectangle* разместим в отдельном файле *rectang.cpp*:

```
// Файл rectang.cpp - описания методов класса прямоугольник
#include <graphics.h>              // прототипы функций графической библио-
теки
#include "f:\POS_C\PRIMER\rectang.h"  // описание класса Point

// Описания методов:
void Rectangle :: risrec()         // прорисовка прямоугольника
  {
    int g=lx/2;
    int v=ly/2;
    line(x-g,y-v,x+g,y-v);
    line(x-g,y+v,x+g,y+v);
    line(x-g,y-v,x-g,y+v);
    line(x+g,y-v,x+g,y+v);
  }
// Конструктор:
Rectangle :: Rectangle(int xi,int yi,int lxi=0,int lyi=0) :
Point(xi,yi)
  {
    lx=lxi;
    ly=lyi;
  }
// Изобразить прямоугольник на экране:
```

```
void Rectangle :: show(void)
  {
    risrec();                    // прорисовка прямоугольника
  }

// Убрать изображение прямоугольника с экрана:
void Rectangle :: hide(void)
  {
    int b,c;
    b=getbkcolor();              // запоминание текущего цвета фона
    c=getcolor();                // запоминание цвета изображения
    setcolor(b);                 // установить цвет изображения
    risrec();                    // нарисовать прямоугольник цветом фона
    setcolor(c);                 // восстановление цвета изображения
  }
```

Теперь можно создать класс «прямоугольник и окружность» на основании двух непосредственных базовых классов. Описание класса *Reccircle* поместим в отдельный файл *reccirc.h*:

```
// Файл reccirc.cpp - описание класса "прямоугольник и окружность"
#include "f:\POS_C\PRIMER\rectang.cpp"      // описание класса Rectangle
#include "f:\POS_C\PRIMER\circle.cpp"       // описание класса Circle

// Класс "прямоугольник и окружность":
class Reccircle: public Rectangle, public Circle
  {
    public:
    // Конструктор:
    Reccircle(int xi,int yi,int ri,int lxi,int lyi) :
    Rectangle(xi,yi,lxi,lyi),    //  Явный вызов конструкторов
    Circle(xi,yi,ri)             //  базовых классов
      {  }
    // Изобразить прямоугольник и окружность на экране:
    void show(void)
      {
        Rectangle :: show();     // изобразить прямоугольник
        Circle    :: show();     // изобразить окружность
      }
    // Убрать изображение с экрана:
    void hide(void)
      {
        Rectangle :: hide();     // убрать изображение прямоугольника
        Circle    :: hide();     // убрать изображение окружности
      }
  };
```

В классе описано три метода: конструктор, который явным образом вызывает конструкторы непосредственных базовых классов, и методы изображения на экране и стирания с экрана прямоугольника и окружности.

Теперь можно составить программу, которая позволит работать с объектами класса *Reccircle*.

```
// Пример множественного наследования
 #include <iostream.h>     // прототипы функций ввода-вывода
 #include <graphics.h>     // прототипы графических функций
 #include <conio.h>        // прототип функции getch()
 #include "f:\POS_C\PRIMER\reccirc.cpp"     // описание класса Reccircle

 void main()
 {
   // Инициализация графики
   int a=DETECT,b;
   initgraph(&a,&b,"F:\\BC5\\bgi");

   Reccircle A(250,100,100,50,50);   // создан невидимый объект A
   Reccircle B(400,300,50,120,140);  // создан невидимый объект B
   cout<<"\n A: x="<<A.Rectangle :: putx()<<", y="<<A.Rectangle::puty();
   cout<<"\n B: x="<<B.Circle :: putx()<<", y="<<B.Circle :: puty();
   getch();                  // ждать нажатия клавиши
   A.show();                 // показать на экране объект A
   getch();                  // ждать нажатия клавиши
   B.show();                 // показать на экране объект B
   getch();                  // ждать нажатия клавиши
   A.hide();                 // стереть объект A
   getch();                  // ждать нажатия клавиши
   A.Circle :: show();       // показать на экране окружность объекта A
   getch();                  // ждать нажатия клавиши
   B.hide();                 // стереть объект B
   getch();                  // ждать нажатия клавиши
   B.Rectangle :: show(); // показать на экране прямоугольник объекта B
   getch();                  // ждать нажатия клавиши
   closegraph();             // закрыть графический режим
 }
```

В программе посредством вызова конструктора формируются два объекта *A* и *B* с координатами центров (250, 100) и (400, 300) соответственно. В результате работы методов *A.show()* и *B.show()* на экране высвечиваются объекты *A* и *B*, каждый из которых состоит из прямоугольника и окружности. Размеры прямоугольника для объекта A: 50 — по горизонтали, 50 — по вертикали, а для *B*: 120 — по горизонтали, 140 — по вертикали. Радиус окружности объекта *A* — 100, а *B* — 50. После стирания с экрана объекта *A* с помощью оператора *A.Circle :: show();* на экране высвечивается окружность объекта *A*. Подобным же образом высвечивается на экран прямоугольник объекта *B*.

Контрольные вопросы и задания

1. Каким образом можно изменить статус доступа к наследуемым компонентам класса?
2. Назовите и охарактеризуйте ключевые слова, используемые в качестве модификаторов статуса доступа.
3. Назовите соглашения о правах доступа к компонентам класса.
4. В базовом классе установлен статус доступа *public*, модификатор доступа имеет значение *private*. Назовите каков статус доступа в производном классе?
5. В базовом классе установлен статус доступа *public*. Какое нужно выбрать значение модификатора доступа, чтобы в производном классе получить статус доступа *protected*?
6. Какой класс называется базовым?
7. В чем отличие между базовым и непосредственным базовым классами?
8. Каковы особенности одиночного и множественного наследования?
9. Приведите формат описания класса, используемый при множественном наследовании.
10. Создайте класс, описывающий квадрат, используя в качестве базового класс *Point*.
11. На основе созданных классов создайте класс «окружность и текст», который позволит изображать окружность и писать текст внутри окружности или за ее пределами.
12. Используя существующие классы, создайте класс «прямоугольник и текст», который позволит изображать прямоугольник и писать текст внутри прямоугольника и за его пределами.

11. ПОЛИМОРФИЗМ

Дословно *полиморфизм* в переводе с греческого означает многообразие форм (*poly* — много, *morphos* — форм). ***Полиморфизм*** можно определить как свойство, позволяющее использовать одно имя для обозначения действий, общих для родственных классов. При этом конкретизация выполняемых действий осуществляется в зависимости от типа обрабатываемых данных. К важнейшим формам полиморфизма в C++ можно отнести следующее:

- перегрузку функций и операций;
- виртуальные функции;
- обобщенные функции, или шаблоны.

Перегрузку функций и операций можно определить как *статический* (*static*) полиморфизм, поскольку он поддерживается на этапе компиляции. Виртуальные функции относятся к *динамическому* (*run time*) полиморфизму, поскольку он реализуется при выполнении программ.

Достоинством полиморфизма является то, что позволяет использовать многократно один раз составленные алгоритмы и, как следствие, обеспечивает уменьшение избыточного кода.

11.1. Перегрузка функций

В C++ допускается наличие нескольких *одноименных* функций, выполняющих аналогичные действия над данными *разных типов*. Например, пусть в программе определены две функции с прототипами:

```
int max(int,int);
float max(float,float);
```

В этом случае в программе допустимо указание следующих операторов:

```
float x,y;
cout<<"max="<<max(5,8)<<endl;
x=12.5; y=24.8;
cout<<"max="<<max(x,y);
```

В процессе компиляции программы при обращении к функциям *max()* в зависимости от типа и числа аргументов будет осуществляться загрузка требуе-

мого экземпляра функции. Описанный механизм называется перегрузкой функций.

Если в программе описано несколько одноименных функций, то при компиляции возможны следующие ситуации:

1. Если *возвращаемые значения и сигнатуры* (тип и число параметров) нескольких функций *совпадают*, то второе и последующие объявления трактуются как переобъявления первого. Например:

```
extern double max(double a,double b);
double max(double c,double d);
```

2. Если *сигнатуры* нескольких одноименных функций *совпадают*, но *возвращаемые значения различны*, то второе и последующие объявления при компиляции рассматриваются как ошибочные. Например, при наличии двух одноименных функций со следующими прототипами

```
unsigned int max(unsigned,unsigned);
extern long max(unsigned,unsigned);
```

при компиляции будет выдано сообщение об ошибке. Причина его в том, что при определении возможности перезагрузки функции тип возвращаемого значения не принимается во внимание, а сигнатуры в данном случае совпадают.

3. Если *сигнатуры* нескольких одноименных функций *различны* (параметры имеют различия по типам или количеству), тогда функция считается ***перегружаемой*** при условии объявления ее экземпляров в одной области видимости. Например:

```
#include <iostream.h>
int max(int,int);
float max(float,float);

void main()
  {
  . . .
  }

float max(float c,float d)
  {
     if (c>d) return(c);
     return(d);
  }

int max(int a,int b)
  {
     if (a>b) return(a);
     if (a<=b) return(b);
  }
```

В данном случае мы имеем два экземпляра перегружаемой функции *max()*.

11.2. Выбор экземпляра функции

При *выборе требуемого экземпляра функции* осуществляется сравнение типов и числа параметров и аргументов. При этом возможны три следующие варианта:

1. Имеет место *точное соответствие типов* аргументов параметрам одного из экземпляров функции. В этом случае осуществляется вызов этого экземпляра функции.

2. *Соответствие* аргументов параметрам *может быть достигнуто* путем преобразования типов аргументов к типам параметров *только для одного экземпляра функции*. В этом случае компилятор пытается сначала выполнить подходящие стандартные преобразования типов данных, а затем преобразования, определенные пользователем.

3. *Соответствие* может быть достигнуто *более чем для одного экземпляра* функции. В этой ситуации следует иметь в виду, что преобразования типов, определенные пользователем, имеют меньший приоритет в сравнении со стандартными преобразованиями. Если соответствие типов достигается путем равноприоритетных преобразований, то компилятором выдается сообщение об ошибке. Это связано с тем, здесь невозможно определить однозначно требуемый экземпляр функции.

Например, пусть имеются следующие прототипы перегружаемых экземпляров функций:

```
void f1(char);
void f1(unsigned int);
void f1(char*);
```

В этом случае ***точные соответствия*** будут иметь место при вызовах

```
f1('b'); //точное соответствие с f1(char)
f1("D"); //точное соответствие с f1(char*)
f1(52u); //точное соответствие с f1(unsigned)
```

Точное соответствие имеет место в случае, когда аргументы точно соответствуют параметрам одного из объявленных экземпляров перегружаемой функции.

Замечания.

• Аргументы типа *char* и *short* и константа *0* считаются как точно соответствующие параметру типа *int*.

• Аргумент типа *float* считается точно соответствующим параметру типа *double*.

Если точное соответствие не имеет места, то компилятор пытается выполнить ***подходящие стандартные преобразования***.

Например, пусть имеются прототипы

```
void f2(long);
void f2(char*);
```

Тогда в т . д.д. *д.2(10);* будет соответствовт. д.д.д.. д.му экземпляру функции, так как аргумент со значением 10 типа *int* преобразуется к типу *long*.

Замечания.

• Аргумент любого числового типа можно преобразовать стандартным путем к любому числовому типу.

- Константа **0** может быть преобразована стандартным путем к указателю.
- Любой указатель может быть преобразован к указателю на *void*.

Преобразования, ***определенные пользователем***, применяются компилятором при отсутствии точного соответствия и подходящих стандартных преобразований. Например, пусть имеются описания:

```
class string
  {
    char *str; int zf;
    public:
    operator int();        //Приведение типа string в тип int
    ...
  };
string p;
void f3(char*);
void f3(int);
```

Тогда вызов *f3(p)*; будет соответствовать экземпляру функции *f3(int)*, потому что есть преобразование из *string* в *int*, определенное пользователем в классе *string*: метод *operator int()* определяет операцию приведения из типа *string* в тип *int*.

Если нет точного соответствия аргументов параметрам и нет подходящих преобразований (стандартных или пользователя), то выдается сообщение об ошибке.

Замечание.

Если у компилятора нет возможности отдания предпочтения между двумя преобразованиями — различные стандартные преобразования приводят к различным экземплярам перегружаемых функций или определено более одного пользовательского преобразования, приводящего к одной цели, — то фиксируется ошибка «двусмысленность». Например:

```
void f4(long);
void f4(double);
f4('a');                   //Ошибка – двусмысленность
```

В данном случае точного соответствия для аргумента типа *char* нет. Для достижения соответствия используются стандартные преобразования. Тип *char* можно преобразовать к типу *long* и к типу *double*. Ввиду равноприоритетности этих преобразований возникает ошибка «двусмысленность».

11.3. Перегрузка стандартных операций

Перегрузку стандартных операций можно рассматривать как разновидность перегрузки функций, при которой в зависимости от типов данных, участвующих в выражениях, вызывается требуемый экземпляр операции. Например, стандартная операция «+» предназначена для сложения обычных числовых данных (с фиксированной или плавающей точкой). C++ позволяет расширить область применимости этой операции, например, для сложения данных комплексного типа или для конкатенации

символьных строк. Для этого нужно переопределить новое поведение стандартной операции «+» применительно к новым типам данных. Это допускается, если хотя бы один из операндов операции является объектом определенного пользователем класса.

Переопределение, и как следствие, перегрузка может выполняться для следующих стандартных операций и встроенных функциональных вызовов:

```
+, -, *, /, =, <, >, +=, -=, *=, /=, <<, >>, <<=, >>=, ==, !=, <=, >=,
++, --, %, &, ^, !, ~, &=, ^=, |=, |, &&, ||, %=, [], (), new, delete.
```

Не могут быть переопределены следующие операции: . , .* , ?: , :: , *sizeof*.

Переопределение операции выполняется с помощью определения так называемой ***операторной функции*** следующего формата:

```
тип_результата  operator знак_операции(список параметров)
{операторы тела операторной функции}
```

Здесь *тип_результата* определяет тип возвращаемого значения при выполнении переопределённой операции; в круглых скобках задаётся список типов параметров, при наличии которых у аргументов обращения к операции с данным знаком будет производиться перегрузка (вызов) именно этого экземпляра операции.

Операторная функция может быть компонентной функцией класса, в этом случае ее внешнее (за рамками описания класса) определение имеет следующий формат:

```
тип_результата  класс :: operator знак_операции(список параметров)
{операторы тела операторной функции}
```

При необходимости можно указывать прототип операторной функции следующего формата:

```
тип_результата  operator знак_операции(список параметров);
```

Пример 1. Перегрузка операций.

Определим класс комплексных данных и для этого класса переопределим стандартные операции ==, < и >.

```
#include <iostream.h>
#include <math.h>
#define FALSE 0
#define TRUE 1
class Complex
{
  private:
    double real;
    double image;
  public:
    Complex(double r){real=r;image=0;} //конструктор для одного параметра
    Complex(double r,double i){real=r;image=i;} //конструктор для 2 параметров
    ~Complex(void){ }
    int operator == (Complex&); //прототип операторной функции ==
```

```
    int operator > (Complex&);   //прототип операторной функции  >
    int operator < (Complex&);   //прототип операторной функции  <
  };
int Complex :: operator ==(Complex &c)  //определение операторной функции ==
  {
  return ((sqrt(real*real+image*image)==
   sqrt(c.real*c.real+c.image*c.image)) ? TRUE:FALSE);
  }
int Complex :: operator >(Complex &C) //определение операторной функции >
  {
    return((sqrt(real*real+image*image)>
    sqrt(C.real*C.real+C.image*C.image)) ?
    TRUE:FALSE);
  }
int Complex :: operator <(Complex &C) //определение операторной функции <
  {
    return((sqrt(real*real+image*image)<
    sqrt(C.real*C.real+C.image*C.image)) ?
    TRUE:FALSE);
  }
main()
  {
    Complex c1(5.2,10.1),c2(5.2,10.1);
    if(c1==c2)cout<<"c1 equal c2"<<endl;
    if(c1 < c2)cout<<"c1 < c2"<<endl;
    if(c1 > c2)cout<<"c1 > c2"<<endl;
  }
```

В приведенном примере мы выполнили переопределение стандартных операций ==, < и > (стандартно используемых для численных данных), обеспечив возможность их применения для комплексных данных. Вызов требуемых экземпляров операций будет осуществляться в зависимости от типа операндов (в нашем примере операнды являются комплексными).

Использование знака операции представляет собой *сокращенную форму записи* вызова операторной функции, которая может вызываться как любая другая функция. Например:

```
c1==c2;                          //сокращенная форма вызова
c1.operator==(c2);-              //полная форма вызова функции-операции
```

На выполнение перегрузки стандартных операций накладываются следующие ограничения:

- не могут быть перегружены следующие символы препроцессора: # и ##;
- операторы =, *[ ]*, *( )* и –> могут быть перегружены только как нестатические компонентные функции. Эти операторы не могут быть перегружены для перечисляемых типов *enum*;
- обычные приоритеты операций и порядок соответствия параметров и аргументов (правила ассоциации) остаются неизменными при использовании перегруженной операции;

• перегруженная операция не может изменить поведение по отношению к встроенным типам данных.

Пример 2. Использование перегрузки операций.

Рассмотрим пример, иллюстрирующий преимущества объектно-ориентированного подхода для вычислительных задач, основанных на матричных преобразованиях. Определим класс *matrix* (динамический двумерный массив). Для указанного класса перегрузим стандартные операции умножения, деления, сложения и вычитания. Такая перегрузка операторов позволит в дальнейшем писать программы с матричными вычислениями на C++ подобно тому, как пишутся программы для обработки простых числовых данных. Пример использования класса *matrix* содержится в конце программы.

```
// ================================================
// Файл matrix.hpp
// ================================================
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
const int  ERR_EXIT = -1;


const double IS_ZERO = 1E-5;                    // если число меньше IS_ZERO,
вым                                                 // оно считается нуле-
class matrix
  {
    private:
      unsigned int
        ROWS,                                   // число строк
        COLS;                                   // число столбцов
      float **point;                            // собственно данные матрицы
    protected:
      float Det2x2();                           // особый случай - матрица 2x2
    public:
      matrix (unsigned int rows,unsigned int cols);// конструктор
      matrix(const matrix &REF);                    // конструктор копиро-
вания
       ~matrix();                               // деструктор
      matrix operator + (matrix &B);
      matrix operator - (matrix &B);
      matrix operator * (matrix &B);
      matrix operator !();                      // транспонирование
      matrix operator =(matrix &B);                 // равенство
    inline matrix operator * (float a);         // Умножение на скаляр
    inline matrix operator / (float a);         // Деление на скаляр
    inline matrix operator * (int a);
    inline matrix operator / (int a);

                                                // обнуление
```

```
    inline matrix set_ZERO();
    inline matrix swap_rows(unsigned int row1,unsigned int row2);
    inline matrix swap_cols(unsigned int col1,unsigned int col2);
    matrix Menor (unsigned int row,unsigned int col);// вычисление минора
      void Out();                          // Вывод содержимого
      matrix input();                      // ввод значений
    public:
      friend float Det(matrix &B);         // определитель
      friend matrix Adj(matrix &B);              // приведение
      friend matrix Inv(matrix &B);              // инвертирование
  };
//=================================================================
// конструктор
matrix::matrix(unsigned int rows,unsigned int cols)
  {
    unsigned int i;
    ROWS=rows;COLS=cols;
    if ( !( point = new float*[ROWS] ) )
      {
        cerr << "Невозможно разместить матрицу в памяти";
        exit(ERR_EXIT);
      }
    for (i=0;i<ROWS;i++)
    if ( !( point [i]=new float[COLS] ) )
      {
        cerr << "Невозможно разместить матрицу в памяти";
        exit(ERR_EXIT);
      }
  }
//=================================================================
//  Конструктор копирования
matrix::matrix(const matrix &REF)
  {
    unsigned int i,j;
    // Размещаем новую матрицу
    ROWS=REF.ROWS;COLS=REF.COLS;
    if ( !( point = new float*[ROWS] ) )
      {
        cerr << "Невозможно разместить матрицу в памяти";
        exit(ERR_EXIT);
      }
    for (i=0;i<ROWS;i++)
      if ( !( point [i]=new float[COLS] ) )
        {
          cerr << "Невозможно разместить матрицу в памяти";
```

```
          exit(ERR_EXIT);
        }
    // Копируем матрицу в новое расположение
    for (j=0;j<COLS;j++)
      for (i=0;i<ROWS;i++)
        point[i][j]=REF.point[i][j];
  }
//=============================================================
/*  Присваивание (=) */
matrix matrix::operator=(matrix &B)
  {
    unsigned int i,j;
    for (i=0;i<COLS;i++)
      delete(point[i]);
    delete (point);
    ROWS=B.ROWS; COLS=B.COLS;
    if ( !( point = new float*[ROWS] ) )
      {
        cerr << "Невозможно разместить матрицу в памяти";
        exit(ERR_EXIT);
      }
    for (i=0;i<ROWS;i++)
      if ( !( point [i]=new float[COLS] ) )
        {
          cerr << "Невозможно разместить матрицу в памяти";
          exit(ERR_EXIT);
        }
    for (j=0;j<B.ROWS;j++)
      for (i=0;i<B.COLS;i++)
        point[i][j]=B.point[i][j];
    return *this;
  }
//=============================================================
/* Деструктор */
matrix::~matrix()
  {
    unsigned int i;
    for (i=0;i<ROWS;i++)
      delete(point[i]);
    delete (point);
  }
//=============================================================
/*  Транспонирование (!) */
matrix matrix::operator !()
  {
```

```
    unsigned int i,j;
    matrix TEMP(ROWS,COLS);
    for (j=0;j<COLS;j++)
    for (i=0;i<ROWS;i++)
      TEMP.point[j][i]=point[i][j];
    return TEMP;
  }
//=================================================================
/* Обнуление */
inline matrix matrix:: set_ZERO ()
  {
    unsigned int i,j;
    for (j=0;j<COLS;j++)
      for (i=0;i<ROWS;i++)
        point[i][j]=0;
     return *this;
  }
//=================================================================
/*    Операция вычитания  (-)  */
matrix matrix :: operator -(matrix &B)
  {
    unsigned int i,j;
      if ((COLS!=B.COLS)||(ROWS!=B.ROWS))
        {
          cerr << "Матрица не совместима с операцией -.";
          exit (ERR_EXIT);
        }
    matrix C(ROWS,COLS);
    for (j=0;j<COLS;j++)
      for (i=0;i<ROWS;i++)
        {
          C.point[i][j]=point[i][j]-B.point[i][j];
        }
    return C;
}
//=================================================================
/*   Операция сложения (+)  */
matrix matrix :: operator +(matrix &B)
  {
    unsigned int i,j;

    if ((COLS!=B.COLS)||(ROWS!=B.ROWS))
      {
        cerr << "Matrix aren't compatible bellow +.";
        exit (ERR_EXIT);
```

```
    }
  matrix C(ROWS,COLS);
  for (j=0;j<COLS;j++)
    for (i=0;i<ROWS;i++)
      {
        C.point[i][j]=B.point[i][j]+point[i][j];
      }
  return C;
 }
//=============================================================
/*  Операция умножения (*)  */
matrix matrix :: operator * (matrix &B)
 {
  unsigned int i,j,k;
  if (COLS!=B.ROWS)
    {
      cerr << "Матрица не совместима с операцией *.";
      exit (ERR_EXIT);
    }
  matrix M(ROWS,B.COLS);
  M.set_ZERO();          // обнуляем M
  // умножение:
  for (j=0;j<B.COLS;j++)
    for (i=0;i<ROWS;i++)
      for (k=0;k<B.ROWS;k++)
        M.point[i][j]=M.point[i][j]+point[i][k]*B.point[k][j];
  return M;
 }
//=============================================================
/* Инвертирование  Inv(M) */
matrix Inv(matrix &B)
 {
  unsigned int i,j;
  float det;
  matrix InvM(B.ROWS,B.COLS);
  det=Det(B);
  if (det==0)
    {
      cerr<<"Матрица не имеет обратной";
      exit(ERR_EXIT);
    }
  // инвертируем
  InvM=(Adj(!B))/det;
  return InvM;
 }
```

```
//=============================================================
/* Детерминант матрицы 2x2 (Det2x2) */
float matrix::Det2x2()
  {    // особый случай
    float det;
    det=point[0][0]*point[1][1]-point[0][1]*point[1][0];
    return det;
  }
//=============================================================
/* Детерминант матрицы Det(M) */
float Det(matrix &B)
  {
    unsigned int n;
    int signo;
    float det=0;

    if (B.ROWS!=B.COLS)
      {
        cerr<<"Матрица должна быти квадратной!";
        exit(ERR_EXIT);
      }
    else
      if (B.ROWS==1)
        return B.point[0][0];
      else
        if (B.ROWS==2)
          return B.Det2x2();
        else
          for (n=0;n<B.COLS;n++)
            {
              // проверка на четность, для четных столбцов
              // знак = +, нечетных - знак = -
              (n&1)==0 ? (signo=1) : (signo=-1);
              det=det+signo*B.point[0][n]*Det(B.Menor(0,n));
            }
    return det;
  }
//=============================================================
/* Умножение матрицы на скаляр */
matrix matrix :: operator *(float a)
  {
    unsigned int i,j;
    for (j=0;j<COLS;j++)
      for (i=0;i<ROWS;i++)
        point[i][j]=a*point[i][j];
```

```
    return *this;
  }
/* Умножение матрицы на целочисленный скаляр */
matrix matrix :: operator *(int a)
  {
    unsigned int i,j;
    for (j=0;j<COLS;j++)
      for (i=0;i<ROWS;i++)
        point[i][j]=a*point[i][j];
    return *this;
  }

//=============================================================
/* Деление матрицы на скаляр */
matrix matrix :: operator /(float a)
  {
    unsigned int i,j;
    for (j=0;j<COLS;j++)
      for (i=0;i<ROWS;i++)
        point[i][j]=point[i][j]/a;
    return *this;
  }
/* Деление матрицы на целочисленный скаляр */
matrix matrix :: operator /(int a)
  {
    unsigned int i,j;
    for (j=0;j<COLS;j++)
      for (i=0;i<ROWS;i++)
        point[i][j]=point[i][j]/a;
    return *this;
  }

//=============================================================
/* Вычисление минора матрицы */
matrix matrix::Menor(unsigned int a,unsigned int b)
  {
    unsigned int i,j,p,q;
    matrix MEN(ROWS-1,COLS-1);
    for (j=0,q=0;q<MEN.COLS;j++,q++)
      for (i=0,p=0;p<MEN.ROWS;i++,p++)
        {
          if (i==a) i++;
          if (j==b) j++;
          MEN.point[p][q]=point[i][j];
        }
```

```
    return MEN;
  }
//=============================================================
/* Приведение матрицы */
matrix Adj (matrix &B)
  {
    unsigned int i,j;
    float signo;
    matrix ADJ(B.ROWS,B.COLS);

    for (j=0;j<B.COLS;j++)
      for (i=0;i<B.ROWS;i++)
        {
          // проверка на четность, для четных знак=+, нечетных - знак=-
          ((i+j)&1)==0 ? (signo=1) : (signo=-1);
          ADJ.point[i][j]=signo*Det(B.Menor(i,j));
        }
    return ADJ;
  }

//=============================================================
/* Ввод данных */
matrix matrix::input()
  {
    unsigned int i,j;
    cout << "Введите матрицу ";
    cout << ROWS << 'X'<< COLS <<" :"<<'\n';
    flush(cout);
    for (i=0;i<ROWS;i++)
      for (j=0;j<COLS;j++)
        {
          cin>>point[i][j];
        }
    return *this;
  }
//=============================================================
/* Вывод */
void matrix::Out()
  {
    unsigned int i,j;
    cout << endl;
    for (i=0;i<ROWS;i++)
      {
        for (j=0;j<COLS;j++)
          cout<< point[i][j] << " ";
```

```
        cout<< endl;
      }
    flush(cout);
  }
//=============================================================
/* Перестановка строк и столбцов */
matrix matrix::swap_rows(unsigned int row1,unsigned int row2)
  {
    unsigned int j;
    float *FILA=new (float);
    if ( (row1>ROWS)||(row2>ROWS) )
      {
        cerr<<"Невозможно переставить несуществующие строки!";
        exit(ERR_EXIT);
      }
    FILA=point[row1];
    point[row1]=point[row2];
    point[row2]=FILA;

    return *this;
  }
matrix matrix::swap_cols(unsigned int col1,unsigned int col2)
  {
    unsigned int i;
    float COLUMN;
    if ( (col1>COLS)||(col2>COLS) )
      {
        cerr<<"Невозможно переставить несуществующие столбцы!";
        exit(ERR_EXIT);
      }
    for (i=0;i<ROWS;i++)
      {
        COLUMN=point[i][col1];
        point[i][col1]=point[i][col2];
        point[i][col2]=COLUMN;
      }
    return *this;
  }
// ============================================================
// Пример работы программы
// ============================================================
#include "matrix.hpp"
#include <stdlib.h>

void main ()
```

```
{
  unsigned int i, j;
  matrix A(4,4),B(4,4),C(4,4);
  A.input();
  cout<<"A = \n";
  A.Out();

  B=Inv(A);
  cout<<"B = \n";
  B.Out();

  C=A*B;
  cout<<"Результат умножения A*B: \n";
  C.Out();

  cout<<"Введите множитель i = \n";
  cin >> i;
  C = C * i ;
  cout<<"Результат умножения: \n";
  C.Out();

  C = !C ;
  cout<<"Результат транспонирования: \n";
  C.Out();
  cout<<"Детерминант A:"<<Det(A)<<"\n";
}
```

Как следует из приведенного примера, на основе использования определенного нами класса работа с матрицами (в сравнении с определением обычных процедур и функций с передачей параметров) оказывается очень простой и может доставить удовольствие.

11.4. Виртуальные функции

Виртуальная функция (*virtual function*) представляет собой компонентную функцию базового класса, которая переопределяется в производном классе. Использование виртуальных функций, в отличие от перегрузки функций, обеспечивает *динамический полиморфизм*, реализуемый на этапе выполнения программы.

При объявлении виртуальной функции в базовом классе перед ее именем указывается ключевое слово ***virtual***. В производном классе виртуальная функция переопределяется. Каждое такое переопределение (*overriding*) виртуальной функции в производном классе означает создание конкретного метода. При переопределении виртуальной функции в производном классе ключевое слово *virtual* не указывается.

Виртуальную функцию можно вызывать как любую другую компонентную фун-

кцию. Однако для поддержки динамического полиморфизма виртуальные функции вызывают через *указатель базового класса*, используемый в качестве ссылки на объект производного класса. Если адресуемый таким образом объект производного класса содержит виртуальную функцию и виртуальная функция вызывается с помощью этого указателя, то при компиляции определяется версия вызываемой виртуальной функции с учетом типа объекта, на который ссылается указатель. Определение конкретной версии виртуальной функции осуществляется в процессе выполнения программы.

Если имеется несколько производных классов от содержащего виртуальную функцию базового класса, то при ссылке указателя базового класса на разные объекты этих производных классов будут выполняться различные версии виртуальной функции.

Переопределение виртуальной функции в производном классе имеет существенные отличия от механизма перегрузки функций. Прежде всего, переопределяемая виртуальная функция должна иметь те же тип, число параметров и тип возвращаемого значения, тогда как перегружаемая функция должна иметь отличия в типе и/или числе параметров. Во-вторых, виртуальная функция должна быть компонентом класса, что не относится к перегружаемым функциям.

Чтобы прояснить смысл использования виртуальных функций, приведем пример, иллюстрирующий разницу между обычными и виртуальными функциями.

Пример. Виртуальные и обычные функции.

```
#include <iostream.h>
class parent
  {                                    //объявление базового класса
   public:
     int k;
     parent(int a)                     // конструктор класса
       {
         k = a;
       }
     virtual void fv()                 // виртуальная функция
       {
         cout << "call fv() of base class: ";
         cout << k << endl;
       }
     void f()                          //обычная функция
       {
         cout << "call f() of base class: ";
         cout << k*k << endl;
       }
  };
class child1: public parent
  {
    public:
      child1(int x): parent(x) {}
```

```
    void fv()
      {
        cout << "call fv() of derived class child1: ";
        cout << (k + 1) << endl;
      }
    void f()          //обычная функция
      {
        cout << "call f() of derived class child1: ";
        cout << (k-1) << endl;
      }
  };
int main ()
  {
    parent *p;          // указатель на базовый класс
    parent ez(5);       // объявление объекта базового класса
    child1 chez1(15);   // объявление объекта производного класса
    p = &ez;            // указатель ссылается на объект базового класса
    p->fv();            // вызов функции fv() базового класса
    p->f();             // вызов функции f() базового класса
    p = &chez1;         // указатель ссылается на объект производного класса
    p->fv();            // вызов виртуальной функции fv() производного класса
    p->f();             // вызов обычной функции f() производного класса
    return 0;
  }
```

При выполнении приведенной программы на экран будут выданы следующие результаты:

```
call fv() of base class: 5
call f() of base class: 25
call fv() of derived class child1: 16
call f() of base class: 225
```

Полученные результаты показывают, что при обращении к виртуальной функции *fv()* производного класса происходит вызов функции производного класса. Обращение к обычной функции *f()* производного класса приводит к вызову одноименной функции базового класса, а не производного.

В языке C++ существует понятие ***чистых (pure) виртуальных функций***. Такие функции используются в случае, когда в виртуальной функции базового класса *отсутствует значимое действие*. При этом в каждом производном классе от заданного класса такая функция должна быть обязательно переопределена.

Чистые виртуальные функции в базовом классс не определяются, вместо этого помещаются их прототипы со следующей формой записи:

```
virtual тип имя_функции (список  параметров) = 0;
```

Присвоение имени функции значения 0 указывает на отсутствие тела функции. При таком объявлении виртуальной функции в базовом классе в каждом производ-

ном классе должно выполняться ее переопределение, в противном случае при компиляции программы будет выявлена ошибка.

Рассмотренные нами *статический* и *динамический* варианты полиморфизма (перегрузка функций и использование виртуальных функции соответственно) соотносят с двумя следующими понятиями: раннее связывание и позднее связывание.

Раннее связывание касается событий этапа компиляции программы, таких как: настройка при вызове обычных функций, перегружаемых функций, невиртуальных компонентных функций и дружественных функций. При вызове перечисленных функий вся необходимая адресная информация известна при компиляции. *Достоинством* раннего связывания является высокое быстродействие получаемых выполнимых программ. *Недостатком* раннего связывания является снижение гибкости программ.

Позднее связывание касается событий, происходящих в процессе выполнения программы. При вызове функций с использованием позднего связывания адрес вызываемой функции до начала выполнения программы неизвестен. В частности, объектом позднего связывания являются виртуальные функции. При доступе к виртуальной фукции через указатель базового класса при выполнении программы определяется тип ссылаемого объекта и выбирается версия виртуальной функции для вызова. *Достоинством* позднего связывания является высокая гибкость выполняемой программы, возможность реакции на события. *Недостатком* является относительно низкое быстродействие программы.

Контрольные вопросы и задания

1. Дайте определение понятия полиморфизма.
2. Назовите примеры проявления полиморфизма в языке C++.
3. Поясните понятия статического и динамического полимирфизма.
4. В чем смысл перегрузки функций, на чем основан механизм перегрузки функций?
5. Приведите формат записи операторной функции.
6. Укажите полную и сокращенную формы вызова операторной функции.
7. Укажите ограничения, накладываемые на перегрузку стандартных операций.
8. Дайте определение и укажите достоинства и недостатки раннего связывания.
9. Дайте определение и укажите достоинства и недостатки позднего связвания.
10. В чем смысл виртуальных функций?
11. Как осуществляется вызов виртуальных функций?
12. Перечислите отличия виртуальных и перегружаемых функций.
13. Укажите, какие результаты будут выведены на экран, если в главной функции (Пример 1 подраздела 11.4) поместить следующие строки:

```
parent ob(10);
child1 chob(8);
p = & ob;
p -> fv();
p -> f();
p = & chob;
p -> fv();
```

```
p -> f();
```

14. Сравните работу с динамическими массивами на основе класса *matrix* из подраздела 11.3 и пример работы с динамическими массивами, рассмотренный в четвертом разделе 4. Какой из них проще для неоднократного использования?

15. Составить программу с использование перегрузки функций, основанную на различии типов аргументов *int* и *float*.

16. Составить программу с использование перегрузки стандартных операций сложения и умножения для вещественных и комплексных переменных.

17. Составьте программу с использованием виртуальных функций для вычислений по следующей формуле:

$$z=\begin{cases}2+a, \text{ если } x>0;\\ a*a, \text{ если } x\le 0.\end{cases}$$

При условии, что значение x вводится при выполнении программы.

12. ОСНОВЫ ОРГАНИЗАЦИИ ВВОДА-ВЫВОДА

Практически любая программа обменивается информацией с ***внешними*** устройствами ПЭВМ, которые используются для ввода данных в программу и вывода результатов. Управлять вводом-выводом можно из программы и на уровне операционной системы (за рамками языка программирования). Мы рассмотрим способы работы с устройствами ввода-вывода из программы и соответствующие средства языка C++.

12.1. Классификация средств ввода-вывода

В основе классификации лежит характер использования средств ввода-вывода языка, который определяется тремя главными факторами:

- уровнем абстрагирования от деталей организации взаимодействия с внешними устройствами;
- стилем программирования, который поддерживают средства ввода-вывода;
- принципами управления внешними устройствами и организацией файловой системы.

Можно выделить ***три уровня абстрагирования***:

- высокий, или уровень потоков и объектов типа «файл» (*TFile*);
- средний, или уровень стандартных системных устройств ввода-вывода (консоль — клавиатура и дисплей, системный принтер);
- низкий, или уровень операционной системы при работе с двоичными файлами и портами ввода-вывода.

С позиций ***стиля программирования*** в языке C++ можно выделить *объектно-ориентированную систему ввода-вывода* и три унаследованные *процедурно-ориентированные системы ввода-вывода* языка C: *потоковые средства стандарта ANSI C, систему ввода-вывода типа UNIX* и *средства низкоуровневого ввода-вывода.*

Все средства ввода-вывода в языке C++ реализованы в виде ***библиотечных*** перегруженных операций << и >> (извлечения и вставки), манипуляторов, классов потоков, констант, глобальных переменных, функций и типов данных.

Базовые ***объектно-ориентированные*** средства ввода-вывода, обеспечивающие работу с потоками (*streams*), объявлены в файле ***iostream.h.*** При подключении файла

fstream.h становятся доступны объектно-ориентированные средства обмена данными между программой и файлами через потоки. В файле ***sstream.h (strstrea. h*** — в ранних версиях Borland C++) объявлены средства для обмена данными с областью памяти, которая рассматривается как массив символов или как строка типа *string*.

Процедурно-ориентированные средства высокого уровня реализованы с помощью библиотеки стандартного ввода-вывода (ANSI C) и становятся доступными при подключении заголовочного файла ***stdio.h***.

Во многих реализациях C++ средства работы со стандартными системными устройствами ввода-вывода ***среднего уровня*** для DOS доступны при подключении файла ***conio.h*** (от англ. *console* — клавиатура и монитор).

Низкоуровневые средства ввода-вывода объявлены в файлах ***bios.h***, ***direct.h***, ***dirent.h***, ***dos.h***, ***io.h*** и ряде других. Примеры организации ввода-вывода на низком уровне можно найти в системе помощи (Help) интегрированной среды програмирования.

В дальнейшем мы будем рассматривать *стандартные средства* ***высокоуровнего*** ввода-вывода. Их использование предполагает усвоение двух важнейших абстрактных понятий «поток» и «файл» и принципов организации обмена данными программы с файлами через потоки.

Отметим, что в основе всех абстрактных понятий ввода-вывода на языке C++ лежат принципы организации файловой системы и порядок работы с файловыми устройствами.

12.2. Принципы работы с потоками и файлами

Поток (*stream*) можно определить как абстрактный канал связи, который создается в программе для обмена данными с файлами. Это понятие введено для того, чтобы можно было не учитывать детали физической организации канала связи между источником и приемником информации.

У всех создаваемых в программе потоков имеются общие поведенческие свойства. Поэтому одинаковые средства (операции и функции) могут применяться для работы с различными потоками. Отличия потоков определяются спецификой их создания и использования.

Вторым важнейшим понятием является файл. ***Файл*** (*file*) представляет собой поименованную совокупность данных, находящуюся на внешнем устройстве и имеющую определенные атрибуты (характеристики). Правила именования файлов определяются файловой системой.

Обычно файл рассматривается как *последовательность* байтов или символов, поэтому файл имеет начало (перед первым байтом), конец (после последнего байта) и при продвижении от начала к концу каждый байт находится в определенной *позиции*. Начало файла и первый байт имеют нулевую позицию, каждый последующий байт имеет позицию на единицу больше предыдущего. Позиция конца файла равна размеру файла в байтах.

Высокоуровневые средства ввода-вывода C++ позволяют работать с *текстовыми и бинарными файлами* после того, как в программе установлена их взаимосвязь с

текстовыми и *бинарными потоками* соответственно. Файл, рассматриваемый как последовательность строк символов, разделеных *непробельными* символами, называется ***текстовым***. Кроме символа «пробел» пробельными символами являются специальные символы *'\t'*, *'\v'*, *'\n'*, *'\r'*, *'\f'* — табуляция (горизонтальная и вертикальная), новая строка, возврат каретки и перевод формата (страницы) соответственно. Среди пробельных символов выделяется символ *«новая строка»*, который используется для стандартного разделения совокупностей строк на «файловые строки» («линии» — *line*), в файле он представляется двумя символами *'\r'* и *'\n'*. *Строки символов* в *текстовых файлах* представляют собой последовательности непробельных символов, которые могут интерпретироваться при вводе как данные определенного типа, представленные в некотором формате. Например, последовательность символов 125, ограниченная с двух сторон пробельными символами, может быть преобразована при вводе в вещественное число типа *double* или целое число типа типа *int*.

Файл называется ***бинарным***, если с ним работают как с последовательностью байтов или символов. Бинарные потоки и файлы обычно используются при перегрузке операций извлечения (>>) и вставки (<<) для пользовательских типов данных или реализации специфических методов ввода-вывода для пользовательских классов.

В качестве файлов рассматриваются не только файлы на дисках, но и любые устройства (файловые), с которыми можно осуществлять операции ввода-вывода. Так, файлами являются клавиатура и дисплей, а также модем, принтер и другие подключенные внешние устройства.

Не все файлы имеют одинаковые поведенческие свойства. Это связано с назначением и физической реализацией файла. Например, для файлов на CD ROM корректны только операции ввода. Поэтому при *связывании* потока с определенным файлом *поток приобретает свойства* этого файла. Такое поведение потока позволяет говорить о записи в поток и чтении из потока, что эквивалентно записи в файл и чтению из файла. Если свойства файла конфликтуют со свойствами связываемого потока, то возникает исключительная ситуация — ошибка открытия файла.

При работе с потоками и файлами различают ***буферизированный*** и ***небуферизированный*** (без использования буферов) ввод-вывод. ***Буфер*** (*buffer*) представляет собой область оперативной памяти, которую используют средства ввода-вывода для промежуточного хранения данных, передаваемых между программой и внешним устройством. Буфер может быть системным. Область памяти, где размещаются *системные буферы*, принадлежит операционной системе, а не программе.

Вывод данных в поток с буфером приводит к выводу этих данных в соответствующий файл только после заполнения буфера. Вывод данных в небуферизованный поток приводит к немедленному выводу в файл. Использование буферов позволяет ускорить работу потока путем поблочного (не побайтного) обмена данными. Вместе с тем, наличие буфера приводит к накладным расходам. Во-первых, при вводе данных следует учитывать возможность ошибок и, следовательно, требуется контролировать и обеспечивать корректность содержимого буфера перед выполнением очередной операции ввода. Во-вторых, в условиях сбоев и возможности принудительного (аварийного) завершения программы при выводе данных необходимо обеспечивать своевременное сохранение содержимого буфера в файле. И наконец, если возникает необходимость синхронизации работы потоков, то наличие буферов также должно учитываться.

Рассмотрим подробнее виды и общие свойства потоков, которые позволяют создавать стандартные средства C++.

По ***направлению передачи данных*** различают следующие потоки:

- *входные*, из которых читаются (извлекаются) данные в переменные программы — *потоки ввода* (*input stream*);
- *выходные*, в которые записываются (вставляются) значения из программы — *потоки вывода* (*output stream*);
- *двунаправленные*, допускающие чтение и запись — *потоки ввода-вывода* (*input-output stream*).

Входной поток не может быть связан с файлом, который предусматривает только запись. Примером такого файла является принтер или монитор. *Выходной поток* не может быть связан с файлом, который имеет атрибут «только для чтения».

Двунаправленные потоки имеют свойства как входных, так и выходных потоков. Потоки такого класса имеет смысл использовать для файлов, к которым можно организовать *произвольный доступ*. С файлами, хранящимися на дискетах или жестких дисках и не имеющими атрибута «только для чтения», можно работать через поток следующим образом. Перед операцией записи-чтения очередной порции данных можно устанавить требуемую *позицию* в файле, начиная с которой и будет осуществлятся очередной обмен данными. Такое *позиционирование* обеспечивает доступ к произвольному байту, а не только к очередному следующему байту, как это имеет место при обычном *последовательном доступе* к файлу. Учитывая, что поток и связанный с ним файл представляют в программе единый объект, обычно говорят только о *позиционировании (seek) потока*.

По ***способу создания*** различают потоки следующих видов:

- *автоматически создаваемые* потоки, которые связаны со стандартными системными устройствами ввода-вывода (***стандартные*** *потоки ввода и вывода*);
- *явно создаваемые* потоки для организации обмена данными с файлами;
- *явно создаваемые* потоки для обмена данными со строкой в оперативной памяти.

Ввод-вывод, выполняемый с использованием стандартных потоков, обычно называют ***консольным*** вводом-выводом.

При подключении файла ***iostream.h*** в программе в начале ее выполнения автоматически создаются 4 ***стандартных потока***:

- ***cin*** — буферизированный поток для ввода данных со стандартного устройства (по умолчанию — с клавиатуры);
- ***cout*** — буферизированный поток для вывода данных на стандартное устройство (по умолчанию — на монитор);
- ***cerr*** — не буферизированный поток для стандартного вывода сообщений об ошибках (по умолчанию — на монитор);
- ***clog*** — буферизированный поток для стандартного вывода сообщений об ошибках (по умолчанию — на монитор).

Связи этих стандартных потоков с файловыми устройствами можно изменять (переназначать) на уровне операционной системы или в программе.

Например, при выполнении команды DOS

```
tstpgm < tstpgm.in > tstpgm.out
```

будут переопределены связи стандартных потоков ввода-вывода. Во время выполнения программы из файла *tstpgm.exe* ввод через *cin* будет производится из файла *tstpgm.in*, а вывод через *cout* – в файл *tstpgm.out*.

Перенаправление стандартных потоков можно выполнять также непосредственно в программе.

Использование потоков *cerr* и *clog* упрощает создание программ на языке C++ для сред, в которых имеются специальные внешние устройства для немедленного вывода сообщений об ошибках (*errors*) через *cerr* и сохранения этих сообщений в файле регистрации ошибок (*errors log*).

Потоки для работы с другими ***файловыми устройствами*** нужно в программе создавать явно, обеспечивая необходимые их свойства. Для большинства задач эти потоки могут быть объектами стандартных классов, объявленных в файлах *fstream.h* и *sstream.h (strstrea.h)*. Заметим, что базовые свойства всех потоков содержит класс *ios*, объявленный в файле *iostream.h*.

Если в программе требуется работать с файлами через потоки, то необходимо подключать файл *fstream.h*. Тогда можно создавать потоки следующих трех классов:

- ***ifstream*** – для ввода из файлов;
- ***ofstream*** – для вывода в файл;
- ***fstream*** – для обмена с файлом в двух направлениях.

В файле ***sstream.h*** (***strstrea.h***) объявлены классы потоков для обмена данными не с внешними устройствами, а с ***оперативной памятью***, в которой выделена специальная область. Эта область рассматривается как массив символов или строка типа *string*. При подключении файла *sstream.h (strstrea.h)* в программе можно создавать потоки 3 + 3 аналогичных классов:

- ***istrstream, istringstream***;
- ***ostrstream, ostringstream***;
- ***strstream, stringstream***

для работы с массивом символов и строкой типа *string* соответственно.

На рис. 12.1 приведен фрагмент иерархии стандартных классов потоков.

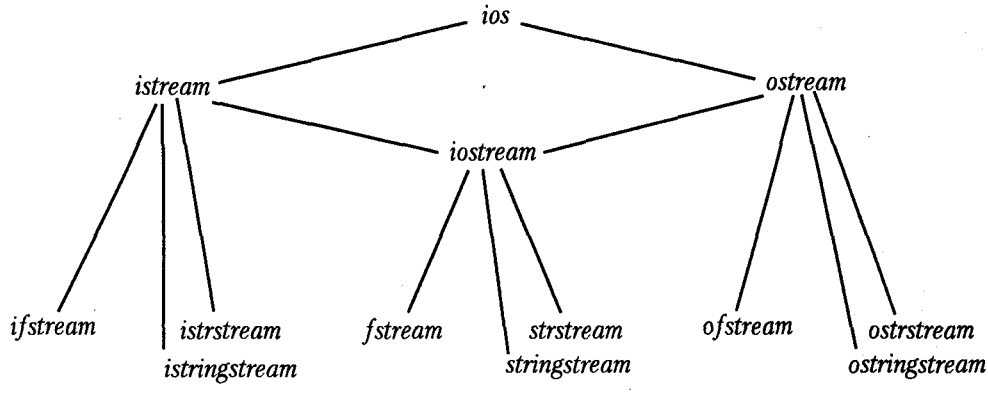


Рис. 12.1. Стандартные классы потоков

Процесс работы с файлом через потоки можно разбить на 4 этапа:

1. Создание потока (объявление переменной).

2. Связывание потока с файлом и открытие (*open*) файла для работы в определенном режиме.

3. Обмен данными с файлом через поток: запись в поток; чтение из потока; управление состоянием потока.

4. Разрыв связи потока с файлом: освобождение буфера («флэширование» — *flush*), закрытие (*close*) файла и разрыв его связи с потоком.

При работе со *стандартными* потоками действия, указанные в пунктах 1, 2, 4 выполняются автоматически. Стандартные потоки *cin*, *cout*, *cerr*, и *clog* называют также ***предопределенными*** *потоками*, поскольку связь их с файловыми устройствами определена до начала выполнения первого оператора программы. Обычно поток *cin* является объектом специального класса *istream_withassign*, а не класса *istream*, как можно было бы ожидать. Соответственно потоки *cout*, *cerr*, и *clog* — это объекты класса *ostream_withassign*. Классы стандартных потоков являются наследниками классов *istream* и *ostream* соответственно.

По умолчанию предполагается, что при корректном завершении программы или при выходе из области видимости потока освобождение буфера и закрытие файла осуществляется автоматически. Но из этого не следует делать однозначный вывод об избыточности пункта 4. Подобная избыточность повышает надежность программ при работе с файлами.

При работе с файлами многое зависит от пользователя, который «склонен» допускать ошибки, да и сами файловые устройства наименее надежный элемент архитектуры ЭВМ. Поэтому при выполнении пунктов 2, 3, 4 следует контролировать наличие ошибок ввода-вывода. Средства языка C++ предусматривают подобный контроль.

Каждый поток, как и всякий объект, в любой момент времени характеризуется состоянием, определяющим поведенческие свойства потока и зависящим от предшествующей работы с потоком. ***Состояние*** представляется набором трех групп переменных:

- *флагов* (признаков) *состояния* потока и указателя, определяющего связь потока с другим потоком;
- *флагов и переменных форматирования*, выполняемого в потоке;
- *переменной*, хранящей *текущую позицию* потока и *флаги режимов* работы с файлами.

Состояние потока определяется также содержимым буфера потока.

Анализ ***флагов состояния*** позволяет установить наличие ошибок ввода-вывода и возможность дальнейшей работы с потоком. Для этого используются функции-члены класса *ios*.

При анализе состояния потока важным понятием является событие ***конец файла*** (*End Of File*, *EOF*). Это событие возникает при попытке чтения из потока, в то время как текущая позиция потока равна размеру файла. То есть операция ввода выполняется после того, как уже был считан последний байт файла или файл пуст. При этом устанавливается соответствующий флаг состояния. Функции ввода-вывода могут возвращать значение *EOF* не только при возникновении события «конец файла», но и при возникновении ошибок ввода-вывода. Данное событие используют для завершения считывания данных из файлов.

Флаги форматирования определяют поведение потока при ***форматированном*** вводе-выводе, организуемом с помощью операций >> и << (извлечения и

вставки). *Форматирование* есть преобразование последовательности байтов потока в соответствии с установленными правилами. При выводе в поток форматирование позволяет получать в файле выводимое значение в определенном виде (формате). При вводе форматирование может позволить считывать из потока последовательность байтов как значение определенного типа. *Флаги* и *переменные* форматирования задают установленные в потоке правила форматирования, которые действуют при выполнении операций >> и <<. Управлять флагами и устанавливать значения переменных форматирования можно с помощью соответствующих методов класса *ios* или с помощью специальных функций — *манипуляторов*, часть из которых объявленна в файле *iomanip.h*, а остальные — в *iostream.h*. Отметим, что при создании любых потоков действуют единые правила форматирования (установленные по умолчанию).

Неформатированный ввод-вывод осуществляется с помощью функций-членов *get()*, *put()*, *read()* и *write()* соответствующих классов *istream*, *ostream*, *iostream*. При этом можно выделить два варианта организации: ***строко-ориентированный*** и ***символьный***. Первый вариант удобен для работы с текстовыми файлами. Символьный ввод-вывод используется, в первую очередь, для работы с бинарными файлами.

На рис. 12.2 приведены возможные варианты организации ввода-вывода. Из всех возможных вариантов мы рассмотрим шесть основных:

- консольный форматированный ввод-вывод базовых типов;
- консольный форматированный ввод-вывод пользовательских типов;
- файловый форматированный ввод-вывод базовых типов;
- файловый символьный ввод-вывод;
- файловый строко-ориентированный ввод-вывод;
- форматированный обмен данными базовых типов со строкой в памяти.

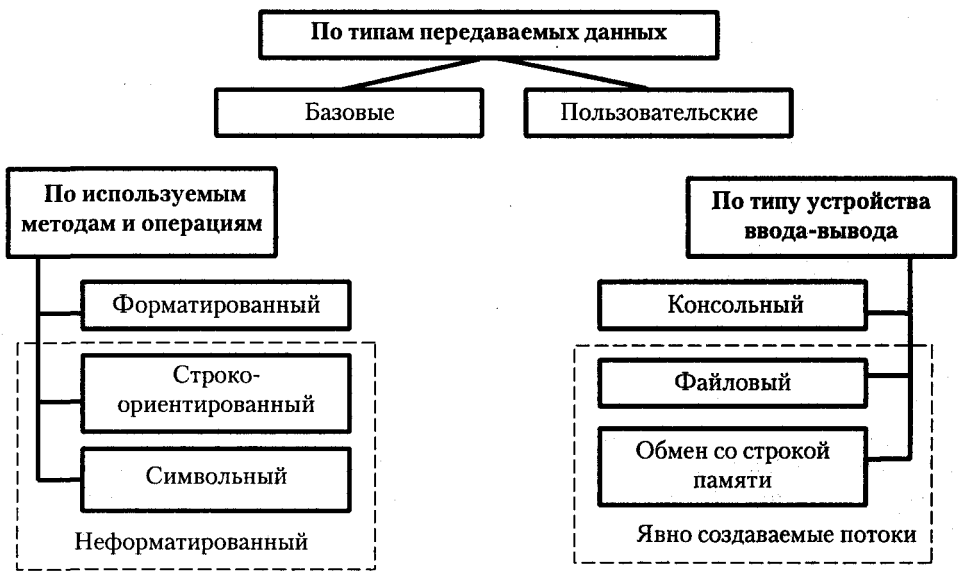


Рис. 12.2. Варианты организации ввода-вывода

В этих вариантах представлены все классы потоков стандартной библиотеки. Остальные возможные варианты можно реализовать по аналогии, поскольку работа ведется с однотипными объектами — потоками. Отличия, в первую очередь, проявляются в именах, используемых при создании потока и связывании его с файлом (строкой).

Отметим, что до настоящего раздела во всех примерах осуществлялся форматированный ввод-вывод через стандартные потоки *cin* и *cout*, при котором использовались параметры форматирования по умолчанию. Поэтому сначала рассмотрим возможности форматирования на примере консольного ввода-вывода (с использованием стандартных потоков), а затем перейдем к другим вариантам организации ввода-вывода.

12.3. Форматированный ввод-вывод базовых типов

Для выполнения форматированного ввода-вывода базовых типов используются перегруженные операции >> и << (правого и левого сдвига), называемые ***операциями извлечения*** (*extraction*) и ***вставки*** (*insertion*) соответственно. Перегрузка проведена в файле *iostream.h* таким образом, чтобы облегчить форматированный ввод-вывод значений переменных базовых типов. Операции вставки и извлечения распознают тип правого операнда и в соответствии с установленными правилами форматирования в потоке преобразуют значение этого операнда.

Пример 1. Консольный форматированный ввод-вывод базовых типов.

Следующая программа показывает свойства форматирования при работе со стандартными потоками *cin* и *cout*.

```
#include <iostream>
int main()
{
  char c = 't';
  char s[7] =  "string ";
  int i = 16;
  double df  = 12.3456789;
  ostream *pcout = &cout;
  cout << c <<'\n'<<s<<'\n'<<i<<'\n'<<df<<'\n'<<pcout<<'\n';
  return 0;
}
```

В результате выполнения программы на экран будут выведены следующие 5 строк.

```
t
string
16
12.3457
0x11ef1020
```

Последняя строка вывода может отличаться, поскольку шестнадцатеричный адрес размещения области памяти, выделенной для потока *cout* на вашем компьютере, может быть другим. Кроме того, формат вывода указателей по умолчанию зависит от реализации C++.

Из приведенного примера становится понятным, что поток *cout* является объектом класса *ostream*. Соответственно, поток *cin* — это объект класса *istream*.

В отношении базовых типов и символьных массивов по умолчанию установлены следующие *стандартные правила форматирования*:

- символы и строки символов выводятся в привычном виде;
- числа выводятся в десятичной системе счисления;
- знак у положительных чисел не выводится;
- у целых чисел выводятся только значащие цифры (незначащие старшие нули отбрасываются);
- вещественные числа выводятся с сохранением до 6 значащих цифр и указанием положения десятичной точки, если дробная часть не нулевая; при отбрасыват. е. млт. е.их значащит. е.фр производится округление.

Можно управлять форматом ввода-вывода с помощью *флагов форматирования* потока, которые объявлены в классе *ios*. Для установки флагов форматирования используются символические константы, которые приведены в табл. 12.1. Во втором столбце таблицы приведены правила форматирования, которые начинают действовать при установке соответствующих флагов.

Отметим, что флаг *boolalpha* отсутствует в ранних версиях C++; флаг *stdio*, наоборот, имеется в этих версиях, но не вошел в стандарт C++.

Флаги форматирования хранятся в защищенном члене класса *ios* — переменной *x_flags* типа *long*. Устанавливать флаги форматирования можно с помощью функции *flags()* — члена класса *ios*. Эта функция объявлена в двух вариантах:

```
long flags();
long flags(long new_fmtflags);
```

Новая совокупность флагов *new_fmtflags* может быть получена с помощью операции | (побитного ИЛИ) и констант, приведенных в табл. 12.1. Функции возвращают прежний набор флагов.

Пример 2. Использование флагов форматирования.

Рассмотрим сохранение флагов, установленных по умолчанию, и установку флагов для вывода вещественных чисел в научном формате и целых чисел в шестнадцатеричной системе счисления.

```
long oldfmtflags = cout.flags();
cout.flags(oldfmtflags|ios::scientific|ios::hex);
cout  <<'\n' <<1.2 << "; (dec) 255 = (hex)" << 255;
```

В результате выполнения этого фрагмента на экране может быть получена следующая строка:

```
1.200000e+00; (dec) 255 = (hex) ff
```

Сохранять значения флагов, установленных по умолчанию, необязательно. Для восстановления начальных установок можно воспользоваться вызовом *flags(0)*. С

Таблица 12.1

Константы установки флагов форматирования

Константа	Способ форматирования
boolalpha	Выводить данные типа *bool* в символьном виде, например, 0 — *false* и !(0) — *true*
dec	Использовать десятичное представление
fixed	Для вещественных чисел использовать представление с фиксированной точкой
hex	Использовать шестнадцатеричное представление
internal	Помещать символ символ-заполнитель (по умолчанию пробел) после знака числа или символа-признака основания системы счисления
left	Прижимать к левому краю при выводе
oct	Использовать восьмеричное представление
right	Прижимать к правому краю
scientific	Для вещественных чисел использовать «научное» представление: мантисса и порядок, разделенные символом *e* или *E*
showbase	Выводить признак системы счисления
showpoint	При выводе вещественных чисел выводить десятичную точку, даже если дробная часть нулевая
showpos	При выводе положительных чисел выводить знак «+»
skipws	Пропускать пробельные символы при вводе
stdio	Освобождать стандартные потоки *stdout*, *stderr* языка С после каждого вывода в поток
unitbuf	Освобождать буферы (выводить содержимое) всех потоков после каждого включения (вывода) в поток
uppercase	При выводе чисел использовать буквы верхнего регистра

помощью функции *setf()*, можно устанавливать отдельные флаги без использования переменной флагов форматирования.

Пример 3. Установка отдельных флагов форматирования.

Следующий фрагмент программы

```
cout.setf(ios::fixed|ios::hex|ios::showbase);
cout  <<'\n' <<12.0 << "; 255 = " << 127;
cout.setf(ios::showpos|ios::scientific|ios::oct);
cout  <<'\n' <<12.0 << "; 255 = " << 127;
```

позволяет получить на экране строки вида:

```
12.000000; 255 = 0x7f
+1.200000e+01; 255 = 0177
```

Функция *setf()* имеет следующие две формы:

```
long setf(long new_fmtflags);
long setf(long new_fmtflags, long reset_mask_fmtflags);
```

В последнем варианте второй операнд задает *маску поля флагов форматов*, которые переустанавливаются.

Функция *setf()* возвращает прежний набор флагов.

Не все флаги форматирования можно установить одновременно. Информацию о взаимосвязи флагов дают константы, используемые при образовании маски для функции *setf()*. В классе *ios* объявлено три таких константы:

basefield – поле флагов *dec, oct, hex*;
adjustfield – поле флагов *left, right, internal*;
floatfield – поле флагов *scientific* и *fixed*.

Эти константы следует использовать в случае, когда перед установкой флага требуется сбросить все альтернативные флаги. Например, при выполнении следующего фрагмента

```
cout.setf(ios::dec, ios::basefield);
cout <<'\n' <<0x7B;
```

на экране может быть получена следующая строка:

```
123
```

Поле оставшихся флагов форматирования можно задать с помощью поразрядных логических операций, например, следующим образом:

```
~ (ios::basefield | ios::adjustfield | ios:: floatfield)
```

Реализация второй формы функции *setf()* в классе *ios* может быть определена следующим образом:

```
setf(long new_fmtflags, long reset_mask_fmtflags)
  {
    return( flags( flags() | ( new_fmtflags & reset_mask_fmtflags )))
  }
```

С помощью функции ***unsetf()*** сбрасываются все флаги, которые помечены в параметре. Функция возвращает предыдущее значение переменной *x_flags*.

Отметим, что не все флаги действуют на входные и на выходные потоки. Учет свойств флагов может облегчить ввод некоторых специфических значений.

Пример 4. Ввод в шестнадцатеричной системе счисления.

Пусть в переменной *obj_state* типа *unsigned int* каждый бит соответствует определенному состоянию некоторого объекта. Тогда следующий фрагмент программы предполагает, что значения будут вводиться в шестнадцатеричной системе счисления.

```
cin.setf(ios::hex);
cout<<"\nВведите состояние объекта: ";
cin >> obj_state;
```

Поэтому, если *sizeof(int)==16*, то для установки старшего бита переменной *bitstate* требуется ввести с клавиатуры строку 8000, а ввод FFFF обеспечит установку всех битов в единичное состояние.

При выводе данных в виде таблиц часто требуется определять фиксированные размеры строки или символьного представления числа. В этом случае говорят о ***поле вывода*** или о числе символов, которые должны быть выведены. Если поле вывода больше числа символов в выводимой строке или числе, то незанятые места заполняются определенными символами. В качестве *символа-заполнителя* по умолчанию устанавливается «пробел». Флаги *left, right, internal* задают правило размещения числа или строки в этом пот. е.Кромт. е.го, для вещт. е.енных чисел часто требуется управлять и точностью представления выводимых чисел.

Значения ширины поля вывода, точности представления и символа заполнителя хранятся в следующих защищенных переменных класса *ios*:

```
int x_width;          // задает минимальную ширину поля вывода
int x_precision;      // определяет максимальное количество значащих
                      // цифр вещественного числа
int x_fill;           // задает символ-заполнитель поля вывода до
                      // минимальной ширины,
```

Доступ к переменным форматирования обеспечивают функции-члены того же класса, имеющие следующие заголовки:

```
char fill();          // возвращает используемый символ-заполнитель
char fill(char cf);   // устанавливает новый символ заполнитель и
                      // возвращает прежний символ
int precision();      // возвращает используемое значение переменной
                      // x_precision
int precision(int p); // устанавливает новое значение и возвращает
                      // прежнее значение
int width();          // возвращает используемый размер поля вывода
int width(int w);     // устанавливает  ширину поля вывода и возвра-
                      // щает прежнее значение
```

Замечание.

Новое значение ширины поля вывода действует только на очередную операцию вывода, после которой восстанавливается прежнее значение величины *x_width = 0*.

Пример 5. Управление параметрами форматирования.

```
...
cout.flags(0);
cout.width(40);
```

```
cout.precision(10);
cout.fill('%');
cout.setf(ios::showpoint|ios::showpos|ios::left);
cout<<17.77777<< " = " << 1.77777e1 << " > " << 17<<'\n';
cout.setf(ios::internal);
cout.precision(7);
cout.width(40);
cout.fill('*');
cout<<17.77777<< " != " << 1.77777e1 << " > " << 17<<'\n';
cout.setf(ios::right);
cout.unsetf(ios::showpos);
cout.precision(4);
cout.width(40);
cout.fill('$');
cout<<17.77777<< " = " << 1.77777e1 << " > " << 17<<'\n';
...
```

В результате выполнения указанного фрагмента программы на экране появятся три строки:

```
+17.77777000%%%%%%%%%%%%%%%%%%%%%%%%%%%% = +17.77770000 > +17
+****************************17.77777 != +17.77770 > +17
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$17.78 = 17.78 > 17
```

В приведенных примерах можно выделить два стиля работы с потоками:

- ***функциональный***, когда для управления состоянием потока используются вызовы вида

```
имя_потока.функция();
```

- ***операциональный***, который наглядно отражает порядок обмена данными, в виде:

```
имя_потока  << выводимое_1_значение << 2_значение <<...<< n_значение;
имя_потока  >> вводимое_1_значение >> 2_значение >>...>> n_значение;
```

Оба стиля равноправны, но второй из них представляется более наглядным и более привлекательным. И средства для широкого применения этого стиля имеются — это манипуляторы. Рассмотрим их подробнее.

12.4. Манипуляторы

Манипуляторами называют специальные функции, позволяющие изменять состояние потока и использующиеся совместно с операциями извлечения и вставки в одном операторе ввода или вывода данных. Отличие манипуляторов от обычных функций состоит в том, что их имена можно использовать в качестве правого операнда при выполнении форматированного обмена с помощью операций << и >>. В качестве самого левого операнда выражения с манипуляторами и операторами обмена всегда используется имя потока.

Например, используя манипуляторы *hex*, *oct* и *endl*, можно написать следующий оператор;

```
cout << "(dec) 1023 = (hex) " << hex << т. е. << "т. е.(oct) " << от. е.< 1023 << endl;
```

В результате выполнения на экране появится строка

```
(dec) 1023 = (hex) 3ff = (oct) 1777
```

Стандартные манипуляторы ввода-вывода делятся на две группы: манипуляторы с параметрами и манипуляторы без параметров. Манипуляторы с параметрами объявлены в файле **iomanip.h**, а без параметров — в файле ***iostream.h***. В табл. 12.2 приведены стандартные манипуляторы без параметров.

Таблица 12.2

Манипуляторы без параметров

Манипулятор	Действие в потоке
dec	Преобразование в десятичное представление
hex	Преобразование в шестнадцатеричное представление
oct	Преобразование в восьмеричное представление
endl	Вставка символа новой строки и выгружение буфера
ends	Вставка в поток нулевого признака конца строки
flush	Выгружение буфера выходного потока
ws	Извлечение и игнорирование пробельных символов *
showbase	Вставка признака системы счисления *
noshowbase	Изъятие признака системы счисления *
skipws	Пропуск пробельных символов при вводе *
noskipws	Отмена пропуска пробельных символов при вводе *
uppercase	Использование символов верхнего регистра при выводе чисел *
nouppercase	Отмена использования символов верхнего регистра при выводе чисел *
internal	Вставка заполнителей между знаком и модулем выводимого числа *
left	Вставка заполнителей после значения в поле вывода *
right	Вставка заполнителей перед значением в поле вывода *
fixed	Использование для вещественных чисел формата *dddd .dd* *
scientific	Использование для вещественных чисел формата *1.ddddd e dd* *
boolalpha	Вывод данных типа *bool* в символическом виде *
noboolalfa	Вывод данных типа *bool* как целых чисел *

Звездочкой отмечены манипуляторы, отсутствующие в ранних версиях C++. Отметим, что манипуляторы без параметров управляют флагами форматирования потока и буфером потока.

Манипулятор *endl* удобно использовать для немедленной выдачи сообщений программы.

Например:

```
...
cout << "\nИдет процесс моделирования." << endl;
...
```

Вывод с таким использование манипулятора *endl* избавляет от необходимости ожидания заполнения буфера до момента окончания процесса моделирования.

Чтобы избежать подобных ситуаций при выводе сообщений об ошибках, поток *cerr* не буферизирован. Поэтому сообщения об ошибках целесообразно направлять в поток *cerr*. При этом именем потока отмечается место обработки ошибок в программе.

Пример 1. Вывод сообщений об ошибках в поток *cerr*.

```
if (SizeFile > MaxSizeFile)
{
  cerr << "\nРазмер файла результатов слишком велик.";
... // операторы обработки ошибки
}
cout<<"\nСохраняются результаты моделирования."<<endl;
```

Если при выводе не требуется начинать новую строку и нужно обеспечить немедленное освобождение буфера, то следует использовать манипулятор *flush*.

Например,

```
cout <<  "\nВведите Tm =  " << flush;
cin >> Tm;
```

или

```
clog <<  "\nЗавершение работы. " << flush;
```

Для управления точностью, шириной поля вывода и другими параметрами форматирования используются *манипуляторы с параметрами*, приведенные в табл 12.3.

Отметим, что манипуляторы обеспечивают все возможности методов класса *ios* для управления форматированием.

Пример 2. Использование манипуляторов с параметрами.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
  double darray[]={6.7891, -0.0089, 5.6789, -0.0056, 4.5678};
  char col_name[]="значение";
  cout << setw(sizeof(col_name))<<col_name<<endl;
  cout << setprecision(3) << setiosflags(ios::fixed | ios::showpoint);
```

```
  cout << setfill('x')    << setiosflags(ios::showpos | ios::left);
  for (int i=0;i<sizeof(darray)/sizeof(darray[0]);i++)
  cout << setw(sizeof(col_name))<<darray[i]<<endl;
return 0;
}
```

В результате выполнения программы на экране появится столбец округленных значений элементов массива с заголовком.

Таблица 12.3

Манипуляторы с параметрами

Манипулятор и тип параметра	Действие в потоке
resetiosflags (long fflags)	Сброс флагов форматирования
setiosflags (long fflags)	Установка флагов форматирования
setbase (int b)	Установка системы счисления с заданным основанием
setfill (int cf)	Установка символа-заполнителя
setpresision (int p)	Установка точности вывода вещественных чисел
setw (int w)	Установка ширины поля для очередной операции вывода

12.5. Флаги состояния потока

При вводе данных с клавиатуры очень легко ошибиться. Поскольку оператор >> предназначен для считывания данных ожидаемого типа в ожидаемом формате, то очередная операция ввода может выполниться как нулевая (безрезультатно). Чтобы контролировать подобные ситуации, класс ios имеет защищенную переменную *state* типа *int*, в которой хранятся *флаги состояния потока*.

Для работы с состоянием потока используются следующие четыре *константы*, определяющие *флаги состояния*:

- *badbit* — поток испорчен;
- *eofbit* — достигнут конец файла;
- *failbit* — следующая операция не выполнится;
- *goodbit* — флаг, в отличие от остальных, установлен в ноль, если поток не испорчен, не достигнут конец файла и следующая операция может выполнится, т. е. поток «хороший» (*good*).

Для анализа состояния потока можно использовать методы *good()*, *eof()*, *fail()*, *bad()* и *rdstate()* без параметров и две перегруженные унарные операции ( ) и !( ).

Метод *good()* и операция ( ) возвращают не ноль, если установлен флаг *goodbit*. Метод *fail()* и операция !( ) возвращают не ноль, если установлен флаг *failbit*. Метод *rdstate()* возвращает значение переменной *state*.

Пример. Анализ состояния потока.

```
cout << "Введите d = "<<flush;
 cin >> d;
if (cin.fail())
{
  cerr<<"Ошибка ввода d";
  exit(1);
}
```

Анализ состояния потока с помощью операции !() можно выполнить следующим образом:

```
if (!(cin >> d))
{
  cerr<<"Ошибка ввода";
  exit(1);
}
```

Флаги состояния можно переустанавливать с помощью метода *clear()* или второй формы метода *rdstate()*, указав в качестве фактического параметра сбрасываемые или устанавливаемые флаги соответственно. Здесь есть аналогия с установкой и сбросом флагов форматирования. Вызов *clear()* без параметров эквивалентен вызову *rdstate(ios::goodbit)*.

12.6. Связывание потоков

Связывание потоков представляет собой установление связи между двумя потоками с помощью метода *tie()* класса *ios*, выполняемое *с целью синхронизации* операций ввода и вывода для связанных потоков. Поясним на примере, зачем требуется связывание двух потоков. Допустим, выполняется следующий фрагмент:

```
cout << "Введите любой символ:"
cin >> c;
```

Наличие буфера у потока *cout* может привести к тому, что сообщение на экране появится позже, чем будет выполнена операция считывания. Поэтому следует синхронизировать работу потоков таким образом, что когда нужен новый символ из одного потока, то во втором потоке освобождался бы буфер. Решение этой задачи с помощью манипулятора *endl* явно слишком грубое. Использование манипулятора *flush* или метода *flush()* целесообразно только при единичных синхронизациях. Метод *tie()* позволяет установить подобную синхронизацию между потоками на требуемом интервале выполнения следующим образом:

```
cin.tie(&cout);          //установление синхронизующей связи
...
cout << "Введите любой символ:"
cin >> c;
...
cin.tie(0);              // разрыв связи
```

Список параметров метода *tie()* может содержать указатель на поток, нулевое значение или быть пустым. Возвращаемое значение есть указатель на поток, с которым связан данный поток. В каждый момент времени любой поток может быть связан только с одним потоком.

Контрольные вопросы и задания

1. Дайте определение понятия «поток».
2. Чем отличаются текстовые и бинарные файлы?
3. Назовите основные классы потоков.
4. Нарисуйте дерево наследования свойств для стандартных классов потоков.
5. Объектами каких классов являются потоки *cin* и *cout*?
6. В каких целях используются потоки *cerr*, *clog*?
7. Опишите основные этапы работы с файлами через потоки.
8. Что такое форматированный ввод-вывод?
9. Чем отличается форматированный ввод-вывод от строко-ориентированного и символьного ввода-вывода?
10. Что общего между форматированным и строко-ориентированным вводом-выводом?
11. Какие средства используются для управлением форматированным вводом-выводом?
12. Какие имеются флаги и переменные форматирования?
13. Назовите флаги, которые влияют только на потоки вывода.
14. Какие флаги определяют только формат вывода?
15. В каких файлах объявлены средства форматирования?
16. Что такое манипуляторы и как они используются?
17. В каких целях используются манипуляторы *flash* и *endl*?
18. Назовите манипуляторы для управления переменными форматирования.
19. Что определяют флаги состояния потока?
20. Зачем и как анализируют флаги состояния потока?
21. В каких целях можно использовать метод *tie()*?
22. Перегрузите операции << и >> для форматированного ввода объектов класса «матрица вещественных чисел».
23. В каком файле объявлены классы файловых потоков?
24. Назовите основные методы файловых потоков.
25. Когда требуется явно разрывать связь потока с файлом?

13. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ ВВОДА-ВЫВОДА

В настоящем разделе рассматривается организация форматированного ввода-вывода пользовательских типов, файлового ввода-вывода пользовательских типов, файлового ввода вывода, обмена со строкой в оперативной памяти и средства пошагового ввода-вывода языка С.

13.1. Форматированный ввод-вывод пользовательских типов

В случае создания нового пользовательского типа данных для удобства работы с ним целесообразно перегрузить операции << и >> (вставки и извлечения). Рассмотрим вариант перегрузки на следующем примере.

Пример 1. Перегрузка операций << и >>.

Рассмотрим выполнение перегрузки операций << и >> (вставки и извлечения) для класса «точка трехмерного пространства *char×int×double*». Соответствующий фрагмент программы имеет следующий вид:

```
class cid_point
 {
   char c_p;
   int i_p;
   double d_p;
   public:
     cid_point(char c=' ', int i= 0, double d = 0){c_p = c; i_p = i; d_p = d;}
     char cp()                                    { return c_p; }
     int ip()                                     { return i_p; }
     double dp()                                  { return d_p; }
     void cp(char c)                              { c_p=c; }
     void ip(int i)                               { i_p=i; }
     void dp(double d)                            { d_p=d; }
   // остальное пока не существенно
 };
```

```
// cid_point insertion
ostream &operator << (ostream &stream, cid_point &p)
  {
    stream << '(';
    stream <<p.cp()<<',';
    stream <<p.ip()<<',';
    stream <<p.dp();
    stream << ')' ;
    return stream;
  };
// cid_point extraction
istream &operator >> (istream &stream, cid_point &p)
  {
    char cr=' ';int ir=0; double dr=0;
    stream.setf(ios::skipws);
    stream >> cr;  //   '('
    stream >> cr; p.cp(cr);
    stream >> cr;  //   ','
    stream >> ir; p.ip(ir);
    stream >> cr;  //   ','
    stream >> dr; p.dp(dr);
    stream >> cr;  //   ')'
    return stream;
  };
```

Здесь в качестве возвращаемого значения операций << и >> объявлена ссылка на объект соответствующего типа (*ostream* и *istream)*. Это необходимо для того, чтобы работать с классом *cid_point* как с базовым типом. Использование флага *skipws* позволяет снять ограничение на число пробелов между элементами (скобками и запятыми) формата внешнего представления объектов класса *cid_point*.

Теперь, в результате выполнения фрагмента

```
int main()
{
   cid_point ok('a',1,2.3);
   cout << "ok = " << ok<<endl;
   ...
```

получим строку

```
ok = (a,1,2.3)
```

Допустимыми значениями при вводе будут являтся не только строки вида (*j*, 7, 9.0), но и такие как (*j*, 7, 9.0) или (<Enter> j <Enter> , <Enter> 7 <Enter> , <Enter> 9.0<Enter>)<Enter>.

Если при перегрузке указать конкретный поток, например, *cout*, то тем самым мы уменьшим область применения перегруженного оператора вставки.

Перегружаемый оператор не должен быть членом класса *cid_point*, потому что в противном случае правым операндом будет объект этого класса, а не поток. Возникает вопрос: как обеспечить вывод в поток защищенных и частных членов класса. В рассмотреном примере это достигнуто с помощью соответствующих методов класса. Другой вариант — объявить в классе операцию >> с атрибутом *friend*, а затем определить ее. Если все элементы-данные класса имеют спецификатор доступа *public*, то перегрузка самая простая.

Перегрузка оператора извлечения из потока более хлопотное занятие из-за достаточно жестких ограничений на единый вид формата при вводе и выводе. Приведенное решение на основе форматированного ввода не универсально, но показывает возможное направление действий. В более сложных случаях на помощь здесь могут придти средства неформатированного ввода-вывода.

При вводе-выводе данных пользовательских типов может потребоваться и перегрузка манипуляторов или введение новых пользовательских манипуляторов.

Пример 2. Определение пользовательского манипулятора.

Реализуем манипулятор *bell* для вывода звукового сигнала, который можно использовать при выполнении операций << и >> для потоков ввода-вывода.

```
istream& bell(istream& stream)
{
   cout  <<  "\a" <<  flush;
   return stream;
}
ostream& bell(ostream& stream)
{
  cout  << "\a" << flush;
 return stream;
}
```

Теперь можно озвучить любое приглашение для ввода следующим образом

```
cout << "\nВведите имя: " << flash << bell;
cin   >> name;
```

Выполнение первой строки приведенного фрагмента сопровождается звуковым сигналом.

13.2. Файловый ввод-вывод

Для обмена данными с файлами, размещенными на дисках, используются три стандартных класса *ifstream*, *ofstream* и *iofstream* файловых потоков. Эти классы являются производными от классов *istream*, *ostream* и *iostream* соответственно. Поэтому они наследуют все свойства родительских потоков по форматированному вводу-выводу, связыванию потоков, управлению состоянием.

Для того чтобы начать работу с файлами, расположенными на логических дисках, необходимо «создать» соответствующие потоки. Для этого можно использовать конструкторы соответствующих классов *файловых потоков* ввода-вывода в самой про-

стой форме — без параметров. Следующий пример демонстрирует выполнение всех четырех основных этапов работы с файлами через потоки.

Пример 1. Копирование одного файла в другой.

Рассмотрим программу копирования файла *in.dat* в файл *out.dat*, в которой явно приведены все четыре основные этапа при работе с файлами. Предполагается, что файлы находятся в корневом каталоге логического диска A. Файл *in.dat* содержит строки символов, разделенные парами символов *'\r'* и *'\n'* (*символом-ограничителем* «конец файловой строки»). Длина файловой строки не превосходит 128 символов.

```
#include <fstream>
#include <stdlib>  // exit()
// вспомогательная функция
void error (const char * message)
  {
    cerr<<'\n'<<message<<'\n';
    exit(1);
  }
int main ()
  {
    char buf[128]; // вспомогательный буфер
    // 1. Создание потоков двух классов
    ifstream fin;
    ofstream fout;
    // 2. Открытие файлов в соответствующих режимах и связывание их с потоками
    fin.open("A:\\in.dat");
    if (!fin.good()) error("Не могу открыть файл для чтения.");
    fout.open("A:\\out.dat");
    if (fout.fail()) error("Не могу открыть файл для записи.");
    // 3. Выполнение операций обмена
    char ch;
    while (fin.getline(buf,sizeof(buf))) fout<<buf<<endl;
    if (!fin.eof() || !fout.good())
      error("Ошибка ввода-вывода при копировании файлов");
    // 4. Закрытие файлов, разрыв связей потоков с файлами
    fin.close();
    fout.close();
    return 0;
  }
```

Число символов, извлекаемых из потока методом *getline()* и помещаемых в буфер, задается вторым параметром. При этом, считывание происходит не более чем до первого встреченного *символа-ограничителя* «конец файловой строки». Метод *getline()* извлекает символ-ограничитель из потока, но не помещает его в буфер. Поэтому после каждого вывода содержимого буфера в поток *fout* вставляется и символ ограничитель. При использовании потока *cin* символ-ограничитель появляется в потоке при нажатии клавиши <Enter>.

Для данного примера причинами выдачи сообщений *«\nНе могу открыть файл...»* могут быть следующие:

• отсутствие файла *in.dat* в корневом каталоге логического диска *A* или его особые атрибуты;

• неготовность устройства (не вставлена дискета, не закрыт карман и др.);

• у существующего файла *out.dat* установлен атрибут «только для чтения» или другие особые атрибуты.

В рассматриваемом примере режим работы с файлом явно не задан, поскольку используется устанавливаемый по умолчанию. Значение режима работы с файлом, устанавливаемое по умолчанию, определяется реализацией языка. Обычно это наиболее «естественный» режим для соответствующего класса файловых потоков. Например, часто по умолчанию бинарный режим не установлен, т. е. предполагается работа с текстовыми файлами.

Подробности работы с режимами следует искать в документации по используемой версии C++.

Возможно совмещение первых двух этапов с помощью второй формы конструкторов потоков:

```
ifstream(const char *fname, int mode = ios::in);
ofstream(const char *fname, int mode = ios::out);
ifstream(const char *fname, int mode);
```

Например:

```
ifstream in( "C:\\autoexec.bat ");
```

Режим работы можно явно указывать и во втором параметре метода *open()*.

Для задания режимов используются символические константы, определенные в классе *ios* (табл. 13.1)

Константы определены в классе *ios*.

Таблица 13.1

Константы режимов работы с файлами

Константа	Режим	Позиция ввода-вывода
in	Открытие файла для чтения	начало файла
out	Открытие файла для записи	начало файла
ate	При открытии файла искать конец файла	конец файла
app	Открытие существующего файла для обновления (записи и чтения)	конец файла
trunc	Усечение размера файла до нулевой длины	начало и конец файла
nocreate	Не создавать новый файл. Если файл не существовал, то при открытии поток переходит в состояние *fail*	
noreplase	Не открывать существующий файл. Если файл существовал, то при открытии без режимов *ate* или *app* поток переходит в состояние fail	
binary	Открыть для символьного (бинарного) обмена	

Режим бинарного обмена не вошел в стандарт C++, но широко используется в имеющихся реализациях.

При связывании файла с потоком соответствующим образом инициализируется и внутренняя переменная потока «позиция ввода-вывода». Значение этой переменной указывает на место записи/чтения очередной порции данных в файле и приведено в табл. 13.1.

Пример 2. Создание нескольких потоков.

Рассмотрим создание нескольких потоков для работы с одним файлом в различных режимах.

```
#include <fstream>
int main()
  {
    const char fn[]="C:\\TEMP\\FILE.DAT";
    char buf[128];
    ofstream fout(fn);          //создали 1 поток для вывода
    if (!fout.good())
      {cerr<<"\nНе могу создать файл с именем '" << fn <<"'."; return 1;}
    fout << "\n1. Проверка файлового ввода. Запись в новый файл.";
    fout.close();               // закрытие для последующей работы с
                                //файлом через другой поток
    fstream fio;                //создали новый поток но другого класса
    fio.open(fn,  ios::out | ios::ate);  //связываем и определяем режим
    fio<<  "\n2. Проверка файлового ввода. Добавление в существующий файл.";
    fio.close();                //закрываем, разрываем связь
    ifstream fin(fn);           //еще один поток но для ввода
    while (!fin.eof())
      {
        fin.getline(buf,128);
        cout<<buf<<endl;
      }
      fin.close();              //разорвали связь
   fio.open(fn,ios::in|ios::noreplace); // установили связь, изменили режим
   fout.open("TEMP.DAT");       //изменили связь потока, режим прежний
   cout<<"\nВывод содержимого файла '"<<fn<< "' через поток fio в файл
'TEMP.DAT'";
  while (!fio.eof())
  {
    fio.getline(buf,128);
    fout<<buf<<endl;
  }
return 0;
}
```

Причиной вывода сообщения *«Не могу создать файл...»* может явиться отсутствие каталога с именем *TEMP*.

Отметим, что режим *(ios::out | ios::ate)* эквивалентен режиму *ios::add* для потоков класса *fstream*.

Из примера понятно, что метод *close()* необходимо явно вызывать при *перенаправлении потока* и изменении режима его работы.

Для ввода и вывода числовых значений можно пользоваться операциями извлечения и вставки. При этом следует помнить, что при вводе пробельные символы извлекаются и игнорируются, а при выводе необходимо между числами вставлять пробельные символы явно.

Пример 3. Перенаправление потоков. В файле *param.dat* записаны значения двух параметров программы в следующем виде:

```
5.6 // r
4.7 // h
```

Программа записывает в файл *result.dat* объем цилиндра, вычисленный на основе значений из файла *param.dat*.

```
#include <fstream>
#include <math.h>              // M_PI
int main()
{
 char rbuf[81];
 double r,h;
 ifstream in( "PARAM.DAT ");
 if (!in.good()) {cerr<< "\nНе могу открыть файл параметров. "; return 1;}
 in >> r; in.getline(rbuf,sizeof(rbuf));
 in >> h; in.getline(rbuf,sizeof(rbuf));
 ofstream out( "RESULT.DAT ");
 out << M_PI*r*r*h <<  "       // v "<<endl;
 return 0;
}
```

В примере 3 приведено *перенаправление потоков fio* и *fout* для работы с различными файлами. Можно переназначать или перенаправлять потоки другим способом. В качестве примера приведем программу, в которой осуществляется *перенаправление стандартных потоков cin* и *cout* для работы с файлами *input.dat* и *output.dat*.

Пример 4. Переназначение стандартных потоков.

```
#include <fstream>
#include <stdlib>
ifstream fin;
ofstream fout;
int main()
  {
     char buf[1];
     fin.open( "PARAM.IN ");
     if(fin.good()) cin = fin; // переназначение cin
     else {cerr<< "Не могу открыть файл 'param.in' "; exit(1);}
     fout.open( "RESULT.OUT ");
   if(fout.good()) cout = fout; // переназначение cout
```

```
  while(!cin.eof())
  {
    cin.read(buf,sizeof(buf));
    cout.write(buf,sizeof(buf));
  }
 return 0;
}
```

Приведенная программа осуществляет посимвольное копирование содержимого входного файла в выходной файл при помощи методов *write()* и *read()* неформатированного ввода-вывода. Правила использования этих методов, по-видимому, понятны из контекста, но рассмотрим неформатированный ввод-вывод подробнее.

13.3. Неформатированный ввод-вывод

Неформатированный ввод-вывод обычно подразумевает работу с *бинарными файлами* или с *потоками символов* (байтов). Среди методов классов *istream*, *ostream*, *iostream* имеются такие, которые ориентированы на работу со строками символов, и соответственно позволяют работать с текстовыми файлами. Поэтому можно выделить два варианта неформатированного ввода-вывода: символьный (бинарный) и строко-ориентированный.

Символьный ввод-вывод

При символьном обмене данными символы (байты) передаются через поток без преобразования. Обычно для этого используются методы *read()* и *write()*, заголовки которых имеют вид:

```
istream& read(char* pstring, int);
ostream& write(const char* pstring, int n);
```

Метод ***read()*** извлекает n символов из потока ввода и помещает их по указателю *pstring*. Метод ***write()*** по указателю *pstring* из области памяти вставляет в поток вывода n символов. Пример использования этих функций приведен выше.

Для ввода одного символа можно использовать метод *get()* параметра. А для посимвольного вывода используется метод *put()* с единственным параметром типа *char*.

Метод *get()* с параметром типа *char* при символьном вводе-выводе работает так же, как и без параметра.

Пример 1. Работа с файлами при символьном вводе-выводе.

Рассмотрим программу, которая добавляет содержимое входного файла в конец выходного файла.

```
#include <fstream>
#include <stdlib> // exit()
int main()
  {
    ifstream fin;
```

```
    fstream fout;
    char ch;
    char fni[80];
    char fno[80];
    cout<<"Введите имя входного файла: ";
    cin.getline(fni, sizeof(fni));
    fin.open(fni,ios::binary);
    if(!fin.good()) { cerr<<"Немогу открыть входной файл."; exit(1);}
    cout<<"Введите имя выходного файла: ";
    cin.getline(fno, sizeof(fno));
    fout.open(fno, ios::app|ios::binary);
    if(!fout.good()){ cerr<<"Немогу открыть выходной файл."; exit(1);};
    while(!fin.eof())
      {
        fin.get(ch);
        fout.put(ch);
      }
    cout<<"Содержимое файла '" << fni << "' добавлено в конец файла '"
<< fno << "'";
    return 0;
  }
```

Если предполагается, что в файле хранятся строки, то имеет смысл работать с такими файлами, не используя символьный ввод-вывод.

Строко-ориентированный ввод-вывод

Классы потоков ввода-вывода имеют средства для чтения и записи строк. При работе с файлами, содержащими строки символов, наряду с рассмотренными выше методами *getline()* и *write()*, следует использовать метод *get()*, который имеет следующий формат заголовка:

```
istream& get(char* pstring, int n, char = '\n');
```

В качестве третьего параметра в *get()* можно задавать другой символ, используемый для разделения строк.

Например, при вызове

```
mstream.get(istr, n);
```

из потока *mstream* будут извлечены *n* символов, если не встретился символ-разделитель *'\n'*, и помещены в массив символов *istr*. В противном случае из потока будут извлечены все символы до разделителя и помещены в массив *istr* с добавленным в конец нуль-символом. Символ-разделитель из потока не извлекается.

Напомним, следующие особенности при работе с файлами:

- при выводе символ *'\n'* заменяется последовательностью символов *'\r'* и *'\n'*;
- при вводе последовательность из этих двух символов рассматривается как один символ *'\n'*.

Пример 2. Работа с текстовым файлом.

Предположим, что в файле *par.dat* хранятся значения радиуса основания и высоты цилиндра в виде следующих двух строк:

```
r  =   3.4
h  =  19.2
```

Программа, выполняющая считывание значения r и v из файла и запись в новый файл вычисленной площади поверхности цилиндра, имеет вид:

```
#include <fstream>
#include <stdlib>                    // exit()
#include <math>                      // M_PI
int main()
  {
    ifstream fin;
    ofstream fout;
    char rstr[80];
    double r,h;
    fin.open("par.dat");
    if(!fin.good())
      {
        cerr<<"Не могу открыть входной файл."; exit(1);
      }
    fin.get(rstr,sizeof(rstr),'=');
    fin.get();                       // считывание разделителя
    fin>>r;
    fin.get(rstr,sizeof(rstr),'=');
    fin.get();
    fin>>h;
    fout.open("dat.out");
    fout<<"s = "<< 2*M_PI*r*(h+r)<<endl;
    return 0;
  }
```

Вместо двух вызовов *get()* для считывания строки символов, находящихся до строки с символьным представлением числа, можно однократно использовать метод *ignore()*. Заголовок этого метода имеет вид

```
istream& ignore(int n = 1, int delimiter = EOF);
```

Здесь n — число символов извлекаемых из потока до символа-разделителя (*delimiter*), заданного вторым параметром; *EOF* — символическая константа, означающая признак конца файла. Из потока извлекается и символ-разделитель.

Например, вызов метода

```
fin.ignore(80,'=');
```

позволяет считать из потока символы до символа, разделяющего строку файла *par.dat* на две части, и сам символ-разделитель.

Из приведенных примеров ясно, что для успешной работы должна быть априорная информация о структуре входных файлов.

Еще одним полезным методом потокового ввода является метод ***gcount()***, который возвращает число успешно считанных символов в предшествующей операции неформатированного ввода. Неформатированное извлечение символов из входного потока осуществляется с помощью одного из методов *get()*, *getline()* и *read()*.

Строко-ориентированный ввод-вывод полезен не только при работе с текстовыми файлами. Можно поток связать со строкой в памяти и производить обмены в памяти потоковыми методами. Рассмотрим этот вопрос подробнее.

13.4. Обмен со строкой в памяти

Библиотека потоков C++ позволяет организовать обмен данными между различными модулями программы через общий массив символов или строку типа *string*. При обменах через массив символов обычно используют С-строки и следующие три класса, объявленные в файле ***sstream.h*** (***strstrea.h***): *istrstream, ostrstream* и *strstream*. Для обменов со строкой типа *string* используются также три класса: *istringstream*, *ostringstream* и *stringstream*. С данными классами работают аналогично как и с соответствующими классами файловых потоков. Небольшие отличия связаны, в первую очередь, с упрощением интерфейса работы с памятью. Классы строковых потоков хорошо обеспечивают форматированный обмен данными с символьным массивами и строками типа *string*.

Мы рассмотрим обмены в памяти с использованием первых трех классов строковых потоков. Организация обмена со строкой типа *string* производится аналогично.

Пример 1. Основные возможности строковых потоков.

Рассмотрим обмен данными в памяти через массив символов (строку). Для простоты обмены проводятся в пределах одного модуля. В программе формируется массив комментариев и значений радиуса основания, высоты и плотности для вычисления массы цилиндра. Затем этот массив считывается. Вычисленная масса с комментарием дописывается в массив. В конце содержимое массива выводится на экран.

```
#include <strstrea>
#include <stdlib>                    // exit()
#include <math>                      // M_PI
char str[180];                       // глобальная переменная
int main()
  {
    char rstr[80];
    double r = 17.,
      h = 5.,
      p = 1.;
    // запись параметров
    ostrstream sout(str,sizeof(str)); // выходного строкового потока
                                      // требуется указывать размер
```

```
  if(!sout.good())
    {
      cerr<<"Немогу записывать в массив ostr";
      exit(1);
    }
  sout << " Для цилиндра с параметрами r = " << r;
  sout << "; h = " << h;
  sout << " и p = " << p;
  // чтение параметров
  istrstream s1in(str);
  s1in >> rstr >> r >> rstr >> h;
  // запись результата
  sout << " получим m = " << M_PI * r * r * h * p;
  // передача из строки в cout
  istrstream s2in(str);
  while (!s2in.eof())
    {
      s2in.get(rstr,sizeof(rstr));
      s2in >> r;
      cout << rstr << r;
    }
  return 0;
}
```

На примере приведенной программы нетрудно заметить схожесть работы с файловыми и строковыми потоками.

В классе *ostrstream* имеется конструктор по умолчанию, который выделяет динамический массив символов. Размер динамической строки автоматически изменяется при выполнении операций обмена. Для работы с такими массивами потребуется метод *str()* без параметров, который возвращает указатель на массив символов, изменяя его размеры при обменах. Для динамической строки метод *str()* «замораживает» массив, и его необходимо явно удалять или возвращать в собственность потока («размораживать») с помощью вызова вида

```
sout.rdbuff->freeze(0);
```

Пример 2. Работа с динамической строкой.

Рассмотрим технику работы со строковыми потоками, один из которых имеет собственный динамический буфер. Для простоты обмен данными организуется в пределах одного модуля. В программе формируется строка, которая затем выводится на экран.

```
#include <strstrea>
#include <stdlib> // exit()
char *str;
int main()
  {
    char rstr[80];
    int p;
    ostrstream sout;        // строковый поток с динамическим буфером
```

```
    if(!sout.good())
      {
        cerr<<"Не могу записывать в поток sout";
        exit(1);
      }
    sout << " В поток sout вставлено не менее ";
    p = sout.pcount()+ 2 + 10;
    sout << p << " символов " << ends;
    str = sout.str();       // "замораживание" динамического буфера
    p=0;
    istrstream slin(str);
    if(slin.fail())
      {
        cerr<<"Не могу считывать из массива str";
        exit(1);
      }
    while(!slin.eof())
      {
        slin.get(rstr,sizeof(rstr));
        p+=slin.gcount();  // подсчет числа успешно считанных символов
        cout << rstr;
      }
    cout << "\nИз потока slin считано не менее " << p <<" символов";
    sout.rdbuf()->freeze(0);
    return 0;
}
```

В примере использован метод выходных строковых потоков *pcount()*, который возвращает число записанных в буфер символов. Применение манипулятора *ends* гарантирует запись признака конца строки при работе с выходным строковым потоком.

Вместо цикла считывания строк из массива символов можно было вывести содержимое массива с помощью оператора

```
cout << str;
```

и второй поток не потребовался бы.

Работать с массивом символов в режиме конкатенации строк можно, указав в третьем параметре конструктора выходного потока режим *ios::ate* или *ios::app*.

13.5. Использование библиотеки *stdio*

Для совместимости с предыдущими версиями язык C++ поддерживает стандартные средства языка ANSI C для работы с потоками в процедурно-ориентированном стиле, которые объявлены в заголовочном файле *stdio.h*. Имеется много общего при работе со средствами работы с файлами через потоки в C и C++. Поэтому сначала кратко рассмотрим работу со стандартными потоками, а затем файловый ввод-вывод.

Вввод-вывод через стандартные потоки

При подключении файла stdio.h автоматически создаются следующие пять потоков:
stdin – для ввода со стандартного устройства (обычно клавиатура);
stdout – для вывода на стандартное устройство (обычно монитор);
stderr – для стандартного вывода сообщений об ошибках (обычно на монитор);
stdaux – для вывода на стандартное дополнительное устройство (обычно на монитор);
stdprn – для вывода на стандартное устройство печати (принтер).

Первые три потока имеют прямые аналоги в C++ – ***cin***, ***cout*** и ***cerr*** соответственно. Их аналогия заключается в устанавливаемой по умолчанию связи с соответствующим файловым устройством и наличии или отсутствии буферизации. Но изначально все эти потоки потоки разные, поскольку за их работу отвечают разные средства и используются различные системные буферы.

Форматированный ввод-вывод

Операции форматированного обмена данными с консолью через стандартные потоки ***stdin*** и ***stdout*** выполняются с помощью двух функций: *printf()* и *scanf()*. Прототипы функций имеют следующий вид:

```
int printf(const char *format[, argument, ...]);
int scanf(const char *format[, address, ...]);
```

Обязательным аргументом функций является строка форматирования. Необязательные аргументы задают имена (в *printf*) или адреса (в *scanf*) переменных, значения которых выводятся или вводятся соответственно.

При успешном выводе метод *printf()* возвращает число переданных байтов. В случае ошибки возвращается значение *EOF*.

При успешном форматированном вводе метод *scanf()* возвращает число запомненных полей. В случае ошибки ввода возвращается 0, если ни произведено ни одного запоминания по указанным адресам. Если при выполнении *scanf()* происходит событие «конец файла» или делается попытка прочитать пустую строку, то возвращается значение *EOF*.

Пример 1. Использование стандартных потоков языка C.

Рассмотрим программу, в которой выполняется форматированный ввода-вывод через стандартные потоки языка C. Она осуществляет считывание с клавиатуры трех чисел и вывод полученной суммы на экран.

```
#include <stdio>
int main(void)
  {
    int i;
    double d1,d2,d3, dsum;
    printf( "\nВведите 3 вещественных числа, разделенных пробелами\n ");
    scanf( "%f%f%f ",&d1,&d2,&d3);
    dsum = d1+d2+d3;
    printf( "%f + %f + %f = %f ",d1,d2,d3,dsum);
    return 0;
  }
```

При выводе с помощью *printf()* строка форматирования выводится посимвольно на экран, пока не встретится символ начала спецификатора формата – %. После этого начинается вывод преобразованного (форматированного) значения первой переменной, следующей в списке аргументов за строкой форматирования. Вывод строки форматирования продолжается до очередного спецификатора и выводится значение второй переменной и т. д.

Рекомендуется, чтобы число аргументов за строкой форматирования совпадало с числом спецификаторов, и тогда соответствие между ними определяется очередностью следования в *printf()*.

Ввод с помощью *scanf()* осуществляется аналогично с одним существенным отличием. В качестве аргументов, следующих за строкой форматирования, необходимо передавать адреса переменных, значения которых вводятся с клавиатуры.

Общая форма *спецификатора формата* имеет следующий вид:

```
%[*][width][.prec][F|N][h|l|L]type_char
```

В квадратных скобках указаны необязательные составляющие спецификатора, а в табл. 13.2 даны необходимые пояснения.

Таблица 13.2

Спецификаторы формата

Компонент	Назначение
*	Здесь указывается один из следующих символов: минус — выровнять по левому краю поля вывода; пробел — если число положительное, то вместо знака выводить пробел; + — значения знаковых типов всегда выводятся со знаком; 0 — заполнить лишнее пространство нулями вместо пробелов; # — выводить 0 перед восьмеричным или 0x перед шестнадцатеричным значением
width	Указывается целое положительное число, задающее максимальную ширину поля вывода в символах
prec	Указывается целое положительное число, задающее максимальное число цифр дробной части, для целых - минимальное число выводимых цифр
F|N	Один из модификаторов размера адреса (указателя) аргумента: F — дальний (длинный), N — ближний (короткий)
h|l|L	Один из модификаторов типа аргумента: h = *short int* l = *long int*, если type_char определяет целочисленное преобразование; l = *double*, если *type_char* определяет вещественное преобразование; L = *long double*
type_char	Указывается один из символов, приведенных в таблице 13.3, который задает соответствующее преобразование типа при вводе-выводе

Таблица 13.3

Символы преобразования при вводе-выводе

Значение type_char	Тип выводимого значения аргумента
	числовые
d	Десятичное целое со знаком типа *int*
D	Десятичное целое со знаком типа *long*
e, E	Вещественное типа *float* в форме *[–]d.dddde[–]ddd* или *[–]d.ddddE[–]ddd*
f	Вещественное типа *float* в форме *dddd.dddd*
g, G	Вещественное типа *float*
o	Восьмеричное целое типа *int*
O	Восьмеричное целое типа *long*
i	Десятичное, восьмеричное или шестнадцатеричное целое типа *int*
I	Десятичное, восьмеричное или шестнадцатеричное целое типа *int*
u	Беззнаковое десятичное целое типа *unsigned int*
U	Беззнаковое десятичное целое типа *unsigned long*
x, X	Шестнадцатеричное типа int
	символьные
s	Строка символов
c	Символ
%	Символ «%»
	адресные
n	Указатель на целое
p	Указатель в шестнадцатеричной форме: *YYYY:ZZZZ* или *ZZZZ*

Пример 2. Некоторые возможности функции *printf()*.

```
#include <stdio>
int main(void)
  {
    printf("%s начала %s!\n","Программа","работу");
```

```
    int i = 98;
    printf("%d = %o <8> = %x <16>.\n",i,i,i);
    float f1=345.54321;
    printf("Различная ширина и точность: \n%f\n%6.2f\n%10.5f",f1,f1,f1);
    printf("\nВыровняли по левому краю f1 = %-15.2f = %.2f;",f1,f1);
    printf("\nЗаполнили нулями f1 = %015.2f = %.2f;",f1,f1);
    float f2 = 3.4554321;
    printf ("\n (%f) / (%f) * 100%% = %2.0f%%",f2,f1,(f2/f1*100.));
    return 0;
}
```

Для функции *scanf()* обычно используют следующие спецификаторы формата:

- одиночный символ – *%c*
- массив символов – *%[size]c*, где *size* – размер массива (константа);
- строка – %s;
- вещественное число – *%e, %E, %f, %g* или *%G*;
- беззнаковые – *%d, %i, %o, %x, %D, %I, %O, %X, %c, %n*;

Использование функций *printf()* и *scanf()* требует большой внимательности. Поэтому проще реализовывать так называемый *строко-ориентированный* ввод-вывод, к рассмотрению которого мы и переходим.

Строко-ориентированный ввод-вывод

Для ввода символов из потока ***stdin*** используется функция, прототип которой имеет вид:

```
int getchar(void);
```

Кроме того, можно использовать функцию с прототипом:

```
int getc(FILE *stream);
```

Функция *getc()*, в отличие от getchar(), позволяет выбирать символ из заданного потока. Поэтому для ввода из потока *stdin* его нужно указать в качестве аргумента. Функции считывают символ из буфера, после того как нажата клавиша <Enter>. В случае ошибки ввода-вывода, функции возвращают значение EOF.

Прототипы аналогичных функций вывода символов имеют вид

```
int putchar(int c);
int putc(int c, FILE *stream);
```

Для ввода символа на экран с помощью *putc()* в качестве второго аргумента нужно указать *stdout* или *stderr*. При возникновении ошибки ввода-вывода функции возвращают значение EOF, в противном случае – значение аргумента (первого – для putc());

Пример 3. Варианты вывода символов.

Рассмотрим программу, в которой четырьмя способами выводится введенный символ.

```
#include <stdio>
int main(void)
```

```
{
  char c1;
  printf("Введите символ: ");
  c1 = getc(stdin);
  printf("\nВведен символ: %c",c1);                    // 1
  putchar('\n');
  putchar(c1);                                        // 2
  putc('\n',stdout);
  putc(c1,stdout);                                    // 3
  if(putc('\n',stderr)==EOF) printf("EOF in stderr!");
  putc(c1,stderr);                                    // 4
  return 0;
}
```

Для вывода строк используются функции, имеющие прототипы:

```
int puts(const char *s);
int fputs(const char *s, FILE *stream);
```

Например, если в программе определена строка вида

```
char str[]= "Проверка вывода строки ";
```

то вызов функции *puts(str)*, как и вызов *fputs(str,stdout)*, приводит к выводу в стандартный поток *stdout* строки символов *str*, оканчивающуюся нулем, заменяя последний нулевой байт символом «новая строка». В случае ошибки вывода возвращается значение EOF, в противном случае возвращается неотрицательное число.

Для того чтобы вывести вещественное число с помощью *puts()*, необходимо воспользоваться функцией с прототипом

```
char *gcvt(double value, int ndec, char *buf);
```

которая обявлена в *stdlib.h*. Функция преобразует значение переменной *value* в ASCIIZ-строку, содержащую символьное представление числа с плавающей запятой, buf — указатель на буфер, в котором хранится полученная строка из ndec значащих цифр.

Пример 4. Использование функции *puts()*.

```
#include <stdio>
#include <stdlib>
int main(void)
 {
  double value = 1.234;
  char buf[19];
  int ndec = 5;
  gcvt(value, ndec, buf);
  puts(buf);
  return 0;
 }
```

Для ввода строк используется функция *gets()*, прототип которой аналогичен *puts()*. Для ввода чисел с помощью *gets()* необходимо воспользоваться функциями *atoi()*, *atol()* и *atof()* для преобразования строки, содержащей символьное представление числа в целое, длинное целое и вещественное число соответственно.

Пример 5. Использование функции *gets()*.

```
#include <stdio>
#include <stdlib> // atoi() ,atol(), atof()
int main(void)
{
   float value;
   char buf[19];
   puts( "Введите вещественное число:  ");
   gets(buf);
   value = atof(buf);
   printf( "Было введено: \n%f\n ", value);
   return 0;
}
```

Файловый ввод-вывод

Для того чтобы начать работу с файлами, расположенными на логических дисках, необходимо «создать» соответствующие потоки. Для этого используются переменные типа указатель на структуру *FILE*, которая объявлена в *stdio.h*.

Пример 6. Задание указателей на структуру *FILE*.

```
#include <stdio.h>
...
FILE *fin, *fout, *fio;
```

Можно считать, что после объявления этих трех переменных в программе созданы три потока для ввода из файла (*fin*), для вывода в файл (*fout*) и для обмена данными с файлом в обоих направлениях (*fio*). Имена этим переменным можно давать любые, но лучше отражать в них функциональное назначение потока.

Для дальнейшей работы после создания потока его нужно связать с необходимым файлом, при этом указав режим работы. Для этого используется функция *fopen()*, которая в случае ошибки ввода-вывода возвращает нулевое значение указателя на *FILE*. Прототип функции *fopen()* имеет вид

```
FILE *fopen(const char *filename, const char *mode);
```

Пример 7. Открытие файла.

```
If  ((fin=fopen( "A:\DAT.IN ",  "r ")) == NULL)
  {
    fprintf(stderr, "\nНе могу открыть файл! ");
  }
```

При связывании файла с потоком соответствующим образом инициализируется и внутренняя переменная потока «позиция ввода-вывода». Значение этой перемен-

ной соответствует номеру позиции в потоке, с которой начинаются записываться или считываться символы. В табл. 13.4 приведены значения, с помощью которых задают режим работы с текстовым файлом и соответствующие значения переменной «позиция ввода-вывода».

Таблица 13.4

Режим работы с текстовым файлом

Значение режима	Описание режима	Позиция ввода-вывода
rt	Открытие файла только для чтения	Начало файла
wt	Создание файла для записи. Если файл с указанным именем существовал, то его содержимое теряется после вызова *fopen(...,"w");*.	Начало файла
at	Открытие файла для добавления в конец файла; если файл с указанным именем не существует, то он создается	Конец файла
rt+	Открытие существующего файла для обновления	Конец файла
wt+	Создание нового файла для обновления. Если файл с указанным именем существовал, то его содержимое теряется	Начало файла
at+	Открытие файла для обновления, начиная с конца; если файл не существовал, то он создается	Конец файла

Для задания режима работы с бинарными файлами вместо символа *t* следует использовать символ *b*. Если явно тип файла не задан, то значение по умолчанию определяется глобальной переменной *_fmode*, объявленной в файле *fcntl.h*.

Перед обменом данными с файлом можно изменять значение внутренней переменной «позиция ввода-вывода» — позиционировать поток, связанный с определенным файлом. Позиционирование осуществляется с помощью функции *fseek()*, имеющей следующий прототип:

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

Аргумент *offset* задает смещение в байтах относительно точки отсчета, положение которой определяется третьим аргументом *whence*. Параметр *whence* может принимать 3 значения, которые приведены в табл. 13.5.

Таблица 13.5

Значение параметра *whence*

Константа	Значение	Положение точки отсчета
SEEK_SET	0	Начало файла
SEEK_CUR	1	Текущее значение переменной «позиция ввода-вывода»
SEEK_END	2	Конец файла

После вызова функции *fseek()* следующая операция обмена с файлом при обновлении его содержимого (режимы *a+* и *w+*) может быть вводом и выводом. Функция возвращает ненулевое значение при ошибке позиционирования. Рассмотрим варианты организации третьего этапа работы с текстовыми файлами.

Форматированный ввод-вывод

Для обмена данными между программой и файлом со строкой форматирования используются две функции *fprintf()* и *fscanf()*, которые аналогичны функциям *printf()* и *scanf()* соответственно. Первое отличие заключается в том, что перед строкой форматирования нужно указывать имя потока.

Например:

```
fprintf(fout,"Первая строка файла\n");
fprintf(fin,"%s",stringbuf);
```

Второе отличие связано с функцией *fscanf()*, которая в указанном выше примере будет считывать не всю строку, а только подстроку (слово) до первого пробельного символа.

Для организации цикла чтения из файла можно воспользоваться функцией *feof()*, которая сообщает о возникновении события «конец файла» в потоке, связанном с определенным файлом. Прототип данной функции имеет вид

```
int feof(FILE *stream);
```

Функция возвращает ненулевое значение, если произошло событие «конец файла» при выполнении предшествующей операции чтения из файла.

При завершении работы с файлом целесообразно разрывать связь потока с файлом с помощью функции *fclose()*, у которой в качестве аргументаа следует использовать имя соответсвующего потока.

Можно использовать функцию *fcloseall()* для закрытия всех открытых в программе файлов. У данной функции пустой список аргументов.

При выполнении закрытия файлов буфер потока освобождается: содержимое выталкивается в файл, в который производилась запись, или теряется, если последней производилась операция чтения.

Если операция чтения выполнилась с ошибкой, то для корректной дальнейшей работы с файлом необходимо прежде всего явно очистить буфер с помощью функции *fflash()*. Для анализа наличия ошибки в потоке используется функция *ferror()*. Данные функции возвращают ненулевое значение, если произошла ошибка при выполнении последней операции записи/чтения или освобождении буфера. Единственный аргумент функций — имя потока.

Например:

```
fprintf(fin,"%s",stringbuf);
if (ferror(fin))
  {
    printf("\nОшибка чтения файла!");
    if (fflash(fin))
      printf("\nОшибка работы с потоком fin!");
  }
```

При таком флэшировании файл остается открытым и информация в нем не искажается. Функция fflash() не влияет на потоки без буферов.

Пример 8. Работа с одним файлом в режимах записи, добавления и чтения через один поток.

Завершить выполнение программы можно принудительно при чтении файла *tstfile.txt* путем нажатия на клавишу <Esc>. Для чтения очередного слова из файла требуется на клавиатуре нажимать на символьную клавишу.

```
#include <stdio.h>
#include <conio.h>                          //kbhit()
int main(void)
  {
   FILE * fsio;                             //FileStreamInputOutput
   char fname[13] = "tstfile.txt";
   char buf[11] = "0123456789";
   char rbuf[81] = "";
   fsio = fopen(fname, "wt");
   fprintf(stdout,"Открыт файл %s в режиме 'wt'\n", fname);
   fprintf(fsio,"Пример работы с файлом;\n");
   for (int i = 2; i<17; i++)
     fprintf(fsio," %i строка.", i);
   fclose(fsio);
   fsio = fopen(fname, "at");
   for ( i = 1; i<17; i++)
     fprintf(fsio," %i добавленная строка.", i);
   fclose(fsio);
   fsio = fopen(fname, "rt");
   printf("\nСодержимое файла %s:\n", fname);
   while(!feof(fsio))
     {                                      //считывается не вся строка,
      fscanf(fsio,"%s",rbuf);               //а только очередное слово
      fprintf(stdout,"очередное слово: %s\n",rbuf);
      while(!kbhit())
        {
         if (getch()!=27) break;
         else
           {
            puts("\nQuit by ESC...");
            fclose(fsio);
            return 0;
           }
        }
     }
   fclose(fsio);
   puts("\nProgram done...");
   return 0;
}
```

В примере используется функция *kbhit()*, определяющая состояние буфера клавиатуры. Функция возвращает нулевое значение, если буфер клавиатуры пуст, и не нулевое — в противном случае.

Строко-ориентированный ввод-вывод

Для ввода-вывода строк через потоки используются функции, прототипы которых имеют вид

```
int fputs(const char *s, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
```

Данные функции по сравнению с аналогичными функциями имееют следующие отличия:

- в качестве второго фактического параметра нужно задавать имя потока;
- при использовании *fputs()* строка выводится без замены терминального символа *0x00* символами *\r\n* (терминальный символ не выводится).
- второй параметр в функции *fgets()* задает размер массива, где будет хранится считанная строка; поэтому, если длина файловой строки меньше величины $n-1$, то она считывается полностью с добавлением нулевого терминального символа, в противном случае считывается только n символов.

Пример 9. Работа с текстовым файлом.

```
#include <stdio>
#include <string>

int main(void)
  {
    FILE *pfs; //PointerFileStream,
    char rbuf[127] = "";
    pfs = fopen("dat.txt", "wt");
    fputs(strcpy(rbuf,"Пример работы с файлом.\n"),pfs);
    fputs(strcpy(rbuf,"Последняя строка файла."),pfs);
    puts("Произведена запись в файл dat.txt 2-х строк.");
    fclose(pfs);
    pfs = fopen("dat.txt", "rt");
    puts("\nСодержимое файла 'dat.txt':");
    while(!feof(pfs));
      {
        fgets(rbuf,sizeof(rbuf)-1,pfs);
        printf("очередная строка: %s",rbuf);
      }
    fclose(pfs);
    puts("\nProgram done...");
    return 0;
  }
```

Контрольные вопросы и задания

1. Какие методы используются для неформатированного ввода-вывода?
2. Как задать бинарный режим работы с файлом?
3. Когда целесообразно использовать средства строко-ориентированного ввода-вывода?
4. Когда целесообразно использовать строковые потоки?
5. Какие классы используются для создания строковых потоков?
6. В чем заключается особенность использования строковых потоков, для которых не указан размер массива символов?
7. Что общего и в чем отличие стандартных потоков библиотек *iostream* и *stdio*?
8. Какие функции библиотеки *stdio* используются для форматированного ввода-вывода?
9. Что такое строка форматирования и спецификатор формата?
10. Напишите программу, в которой стандартный поток *cerr* перенаправлен в файл *C:\error.log* в режиме добавления.
11. Напишите программу с перенаправлением потока *clog* в файл *error.log*.

14. ШАБЛОНЫ

Шаблон представляет собой специальное описание родовой (параметризованной) функции или родового класса, в которых информация об используемых в реализации типах данных преднамеренно остается незаданной. Типы используемых данных передаются через параметры шаблона. Аппарат шаблонов позволяет одну и ту же функцию или класс использовать с различными типами данных без необходимости программировать заново каждую версию функции или класса.

Компилятор автоматически порождает (*instanting*) столько разных версий классов или функций на основе шаблона, сколько требуется для обработки каждого из вновь используемых типов.

Особенно полезен такой подход при конструировании контейнерных классов, а также функций, реализующих обобщенные алгоритмы обработки. ***Контейнером*** называется класс, который используется как структура данных, содержащая набор элементов (некоторого другого типа). Примеры контейнеров: массивы, множества, списки, очереди, хэш-таблицы.

Шаблон создается при помощи ключевого слова *template* и угловых скобок со списком параметров, непосредственно за которым следует описание класса или функции:

```
template <class Фиктивное_Имя1, class Фиктивное_Имя2, ... > определение ;
```

Вместо ключевого слова ***class*** допускается использование ключевого слова ***typename***.

Существует взгляд на шаблоны как на «интеллектуальные» макросы, которые способны сообщить компилятору о каждом случае использования шаблона для аргументов новых типов данных.

14.1. ПАРАМЕТРИЗОВАННЫЕ ФУНКЦИИ

Рассмотрим функцию *max(x, y)*, которая возвращает больший из двух аргументов; *x* и *y* могут принадлежать к любому типу.

Первый способ решения этой проблемы — использовать макрос:

```
#define max(x,y) (((x) > (y))? (x) : (y))
```

Однако использование *#define* обходит механизм контроля типов. Фактически такое использование макроопределений почти вышло из употребления в C++. Естественно требовать, чтобы функция *max(x, y)* сравнивала только совместимые типы. К

сожалению, использование макроса допускает любое сравнение, например, между целочисленным значением и структурой, которые несовместимы.

Второй недостаток использования макро-подстановки заключается в том, что подстановка будет выполнена не там, где необходимо. Используя взамен макроса шаблон, можно определить заготовку для семейства связанных перегруженных функций или классов, при этом тип данных передавать как параметр:

```
template <class T> T max(T x, T y)
  {
    return (x > y)? x : y;
  };
```

Здесь тип данных представлен аргументом шаблона *<class T>*, где *T* — фиктивное имя типа данных, которое программист выбирает по своему усмотрению. При использовании его в приложении компилятор сгенерирует соответствующий код для функции *max* согласно действительному типу данных, использованному при вызове функции.

Пример 1. Параметризованная функция для произвольного типа данных.

```
#include <iostream.h>
class Myclass
  {
    public:
      float r;
      Myclass(float _r){r=_r;};
      // чтобы иметь возможность сравнивать объекты типа
      // MyCalss, перегружаем оператор >
      bool operator>(Myclass& l){return r > l.r; };
  };
template <class T> T max(T x, T y)
  {
    return (x > y)? x : y;
  };
void main(int argc, char **argv)
  {
    int i=25;
    Myclass x(10.95), y(11.01);
    int j = max(i,5);              // аргументы являются целыми
    cout <<" j= " << j << endl;
    Myclass z = max(x,y);          // аргументы имеют тип Myclass
    cout << " z= "<<z.r ;
  }
```

В результате выполнения программа выдает следующее:

```
j= 25
z= 11.01
```

Любой тип данных (не только класс) может использоваться в подстановке *<class T>*. Компилятор сам позаботится о вызове соответствующего оператора

сравнения *operator>()*, так что возможно использовать определенную таким образом функцию с аргументами любого типа, для которого определен оператор сравнения.

При использовании шаблонов следует проявлять осторожность. Дело в том, что логика работы созданной параметризованной функции для определенных типов данных может отличаться от предполагаемой. Приведем пример, иллюстрирующий вариант такого использования шаблона.

Пример 2. Некорректное использование шаблона.

```
#include <string.h>
#include <iostream>
template <class T> T max (T x, T y)
  {
    return (x > y)? x : y;
  };
int main()
  {
    int a = 5, b=30 , c=15, d = 0;
    float pi = 3.141596, e = 2.172;
    d = max(a,max(b,c));
    cout << " Наибольшее из трех целых:" << d << endl;
    cout << " Наибольшее из двух целых:" << max(d, a) << endl;
    cout << " max(pi,e) = " << max(pi,e) << endl;
    char m[] = "Мама", p[] = "Папа", *s;
    s= max(p,m);              // Внимание! Функция max сравнит два
                              // указателя, а не содержимое строк!
    cout << "Кто в доме хозяин? :" << s << endl;
  }
```

В результате выполнения программа выдаст следующее:

```
Наибольшее из трех целых: 30
Наибольшее из двух целых: 30
Кто в доме хозяин? : Мама
```

Предполагалось, что в последнем случае программа будет посимвольно сравнивать две строки, а в кодировке ASCII, справедливо отношение "Папа" > "Мама". В действительности результаты такого сравнения непредсказуемы и зависят от того, в какой последовательности компилятор распределяет в памяти указанные строки символов. Для приведенной программы компилятор автоматически сгенерирует по шаблону следующие три перегруженные функции:

```
int max (int x, int y)
  {return (x > y)? x : y;}
float max (float x, float y)
  {return (x > y)? x : y;}
char* max (char * x, char * y)
  {return (x > y)? x : y;}
```

Экземпляр функции создается каждый раз при вызове функции с аргументами, для которых определения не найдено. Чтобы не проводилось бессмысленное сравнение двух указателей, для строк можно явно перегрузить функцию *max* следующим образом:

```
char *max(char *x, char *y)
  {
    return(strcmp(x,y) > 0)? x : y;
  }
```

В этом случае компилятор не будет производить генерацию функции по шаблону для аргументов типа *char**, а воспользуется уже готовой. Другое решение проблемы — использовать классы, для которых определена операция «>», например, *class String*.

Следует иметь в виду, что для параметризованных функций преобразования аргументов по умолчанию не выполняются. Кроме того, параметризованная функция должна использовать все типы формальных аргументов шаблона, в противном случае компилятор не сможет определить фактические типы для генерации тела.

Принимая решение о перегрузке операндов, компилятор игнорирует функции, которые сгенерированы компилятором неявно.

Пример 3. Преобразование типов аргументов при использовании шаблонов.

```
template <class T> T min(T a, T b)
  {
    return  (a < b) ? a : b;
  };
void    f(int i, char c)
  {
    min(i, i);              // вызов min(int ,int )
    min(c, c);              // вызов min(char,char)
    min(i, c);              // нет совпадений для min(int,char) !
    min(c, i);              // нет совпадений для min(char,int) !
  }
```

Для того чтобы вызов функции *min()* не приводил к сообщениям об ошибках, в последних двух строках данного примера, необходимо явно объявить формат, позволяющий произвести преобразование типов. Так, если добавить в указанную программу только объявление (тело функции будет создано шаблоном):

```
int     min(int,int);
```

то сообщения об ошибках выдаваться не будут.

14.2. Параметризованные классы

Рассмотрим пример класса *Vector* (одномерный массив). Независимо от того это есть вектор целых, вещественных, комплексных чисел или любого другого типа, основные операции, которые он позволяет выполнять — те же (инициализация, получение элемента по индексу и т. п.). Естественное решение для определения такого класса — использовать шаб-

лон. Аналогично тому, как компилятор производил генерацию функций, при подстановке фактического типа в шаблон класса *Vector*, система создает класс автоматически.

Пример. Использование шаблона для определения класса *Vector*.

```
template <class T> class Vector
  {
    T *data;
    int size;
    public:
      Vector(int);
      ~Vector( ) { delete[ ] data; }
      T& operator[ ] (int i) { return data[i]; }
  };
template <class T> Vector<T>::Vector(int n)
  { // конструктор
    data = new T[n];
    size = n;
  };
int main()
  {
    Vector<int> x(10);      // создаем вектор из 10 элементов типа int
    for (int i = 0; i < 10; ++i)
      x[i] = i*i;           // присваиваем значения
    Vector<char> c(5);      // создаем вектор из 5 элементов типа char
    for (char ic = 0; ic < 5; ++ic)
      c[i] = ic + 'a';      // присваиваем значения
    //
    for (int i=2; i<7; i++ )
      cout << x[i] << " ";
    return o;
  }
```

Так же как для параметризованных функций, для определенного типа данных возможно запретить автоматическую генерацию классов по шаблону:

```
class Vector<char *> {...};
```

Вектор строк будет всегда соответствовать объявлению с типом *char** в угловых скобках.

14.3. Стандартная библиотека шаблонов

В последних версиях стандартных библиотек, базирующихся на стандарте для языка C++, содержится большой и исчерпывающий набор структур данных и функций, основанных на шаблонах. Такое расширение стандартных библиотек носит название *стандартной библиотеки шаблонов* (Standart Template Library, *STL*). Эти структуры

данных содержат шаблоны классов для представления множеств, списков, карт (словарей), стеков, очередей, очередей с приоритетами и пр. Такая стандартизация улучшает мобильность и надежность программ, позволяет быстро создавать надежные приложения и поддерживать их с меньшими усилиями.

Все классы библиотеки STL разрабатывались для совместного функционирования с большим набором ***обобщенных алгоритмов*** (шаблонов функций общего вида для манипулирования контейнерами произвольной природы). Такое разделение функций (контейнеры — хранят данные, алгоритмы — ими оперируют) позволило существенно сократить объем библиотеки, причем во многих случаях такая реализация оказывается более эффективной по сравнению с реализациями, основанными на полной инкапсуляции данных и операций.

Основная часть STL не объектно-ориентированная. Для тех, кто привык к преимуществам объектно-ориентированного программирования, их отсутствие может потребовать некоторого пересмотра привычного мышления. Тем не менее, такой подход имеет ряд преимуществ, таких как меньший исходный код, более эффективное кодирование и гибкое использование алгоритмов на основе указателей.

Основной *недостаток* использования стандартной библиотеки шаблонов — более высокий риск ошибки. Например, итераторы не должны быть разыменованными (нулевыми). Шаблоны обеспечивают менее точную диагностику, и код может оказаться неожиданно большим. Опыт работы с библиотекой и компилятором помогут уменьшить эти проблемы. Рассмотрим основные понятия, используемые в STL.

Одно из важнейших понятий библиотеки, наряду с контейнером и обобщенным алгоритмом — итератор. ***Итератор*** представляет собой класс, обеспечивающий доступ к данным другого класса. В общем смысле итератор обобщает понятие указателя: он предоставляет доступ к элементам контейнера без необходимости инкапсулировать контейнер. Аналогично тому, как используются обычные указатели, итераторы могут использоваться для различных целей. Итератор может указывать на конкретный объект, пара итераторов определяет диапазон объектов в последовательности (начало — конец). Причем элементы последовательности следуют друг за другом в логической последовательности (в физической памяти они располагаются не обязательно последовательно). Физическая реализация порядка выборки элементов из последовательности возлагается на контейнеры.

Другое понятие, широко используемое в STL, но несколько необычное, — расширение понятия функции до ***объекта-функции***. В большинстве обобщенных алгоритмов STL используются указатели на функции, которые передаются как параметры. Для того чтобы осуществлять контроль за передаваемыми значениями и, как следствие, повысить надежность разрабатываемых программ, вместо указателей на функции передаются объекты необходимого типа. Для этих объектов должен быть определен оператор «скобки». Кроме того, зачастую такая подстановка приводит к лучшей производительности за счет использования *inline*–подстановки, которая невозможна для механизма, основанного на передаче указателя на функцию.

Пример 1. Функциональный объект, накапливающий значения.

```
class Summator : public unary_function<Arg, Arg>
  {
    int  Summa;
    public:
      Arg operator()(const int & arg)
        {
          Summa +=  arg;
          return Summa;
        }
  };
```

Если описанный функциональный объект поместить в программу, то каждый раз при вызове функции-скобки с аргументом он будет накапливать переданные значения и возвращать результат.

Пример 2. Использование шаблонов объектов-функций.

```
#include<functional>
#include<deque>
#include<vector>
#include<algorithm>
#include <iostream>

using namespace std;

// Создаем новый функциональный объект на основе unary_function
template<class Arg>
class factorial : public unary_function<Arg, Arg>
  {
    public:
      Arg operator()(const Arg& arg)
        {
          Arg a = 1;
          for(Arg i = 2; i <= arg; i++)
            a *= i;
          return a;
        }
  };

int main()
  {
    // Инициализируем двустороннюю очередь массивом целых
    int init[7] = {1,2,3,4,5,6,7};
    deque<int> d(init, init+7);
    //Создаем пустой вектор для хранения значений факториала
    vector<int> v((size_t)7);
```

```
    // Трансформируем числа в двусторонней очереди в их факториалы
    // и сохраняем их в векторе
    transform(d.begin(), d.end(), v.begin(),
      factorial<int>());
    // печатаем результаты
    cout << " Данные числа: " << endl << "      ";
    copy(d.begin(),d.end(),
      ostream_iterator<int,char>(cout," "));
    cout << endl << endl;
    cout << "Имеют факториалы: " << endl << "      ";
    copy(v.begin(),v.end(),
      ostream_iterator<int,char>(cout," "));
    return 0;
  }
```

В результате выполнения программа выдаст следующее:

```
Данные числа:
1 2 3 4 5 6 7
Имеют факториалы:
1 2 6 24 120 720 5040
```

Рассмотрим пример, который иллюстрирует работу некоторых базовых алгоритмов STL. Для двух объектов типа мультимножество найдем общие и различные элементы.

Пример 3. Использование контейнера типа мультимножество

```
#include <set>
#include <algorithm>
#include <iostream>
using namespace std;

typedef multiset<int, less<int>, allocator> set_type;

ostream& operator<<(ostream& out, const set_type& s)
  {
    copy(s.begin(),s.end(),
      ostream_iterator<set_type::value_type,char>(cout," "));
    return out;
  }

  int main(void)
  {
    // создаем мультимножество целых величин
    set_type  si;
    int  i;
    for (int j = 0; j < 2; j++)
```

```
    {
      for(i = 0; i < 10; ++i)
        {
          // инициализируем его значениями от 1 до 10
          si.insert(si.begin(), i);
        }
    }
  // вывод на печать
  cout << si << endl;
  set_type si2, siResult;
  for (i = 0; i < 10; i++)
    si2.insert(i+5);
  cout << si2 << endl;

  // пример работы нескольких алгоритмов
  // получим объединение двух множеств и результаты поместим в siResult
  set_union(si.begin(),si.end(),si2.begin(),si2.end(),
  inserter(siResult,siResult.begin()));

  cout << "Объединение:" << endl << siResult << endl;
  siResult.erase(siResult.begin(),siResult.end());

  // получим пересечение двух множеств и результаты поместим в siResult
  set_intersection(si.begin(),si.end(),si2.begin(),si2.end(),
  inserter(siResult,siResult.begin()));
  cout << "Пересечение:" << endl << siResult << endl;

  return 0;
}
```

Результаты работы программы:

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 6 7 8 9 10 11 12 13 14

Объединение:

0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 11 12 13 14

Пересечение:

5 6 7 8 9
```

Контрольные вопросы и задания

1. Для чего используются шаблоны?
2. В чем основное преимущество использования шаблонов; в чем их недостатки?
3. Каким образом с помощью шаблонов можно создать контейнеры, содержащие элементы различных типов? Какой способ эффективнее?

4. Поясните основное отличие между перегруженными функциями, создаваемыми с помощью шаблонов, и обычной практикой создания таких функций?

5. В чем основное преимущество использования функциональных объектов?

6. Дайте общую характеристику стандартной библиотеки шаблонов.

7. Укажите возможные причины возникновения некорректностей при использовании шаблонов.

8. Составить программу, реализующую шаблон класса список. Список должен инкапсулировать все базовые операции работы со списками, такие как: добавление, поиск, исключение и сортировка элементов.

9. Составить программу, реализующую шаблон для создания двоичных деревьев с возможностью поиска элементов, определения максимальной длины ветвей и преобразования к сбалансированному виду.

10. Составить программу с использованием STL, заменяющую последние пять элементов вектора типа *float*, состоящего из 20 элементов, на квадраты первых пяти элементов списка целочисленных значений. Вывод содержимого вектора осуществить в порядке возрастания элементов.

15. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ C++

15.1. Пространство имен

Пространство имен (*namespace*) представляет собой некоторую объявляемую с помощью специальной конструкции область, служащую для локализации (ограничения области действия) имен переменных и объектов с целью избежания конфликтов при использовании одинаковых имен. Ситуация, когда в программе применяется несколько одинаковых идентификаторов, обычно возникает в случае использования библиотек функций и классов разных производителей. При использовании пространства имен, помимо необязательного глобального пространства имен, выделяются несколько частных пространств. В разных частных пространствах можно использовать одинаковые идентификаторы переменнных, классов и объектов, которые действуют в рамках своего частного пространства имен и имеют в нем свой смысл.

Пространство имен задается следующим образом:

```
namespace имя
  {
    // объявления
  }
```

Здесь:

namespace — ключевое слово, задающее пространство имен;

имя — любой идентификатор.

В фигурных скобках находится само пространство имен, имеющее название с заданным именем. В результате такого определения пространства все имена, входящие в данное пространство имен, будут видимы только внутри введенного пространства.

Пример 1. Определение и использование пространства имен.

```
#include <iostream>
namespace A
  {
    int x,y;
    class M
      {
        public:
          void vvodx(int c) {x=c;}
```

```
        int dostx() {return x;}
      };
    void vyvod(int k) {cout << k;}
  }
void main()
  {
    int z,t,y;
    y=3;
    A::y=7;
    A::M b;
    z=9;
    b.vvodx(z);
    A::x=z;
    cout << "\n A::x=";
    A::vyvod(A::x);
    cout << "\n y=";
    A::vyvod(y);
    cout << "\n A::y=" << A::y;
    t=b.dostx();
    cout << "\n t=" << t;
  }
```

В результате работы программы на экран будут выданы следующие сообщения:

```
A::x=9
y=3
A::y=7
t=9
```

В примере переменные *x*, *y*, класс *M* и функция *vyvod()* находятся в области видимости, определенной пространством имен *A*. Описание функции *main()* находится в глобальном пространстве имен. В глобальном пространстве имен видимы переменные *z*, *t* и *y*. Таким образом, в программе объявлены две переменные с одним именем *y*: первая видима в пространстве имен *A*, а вторая — в глобальном пространстве.

Применение имен идентификаторов внутри пространства имен производится обычным образом. При использовании имен идентификаторов другого пространства имен необходимо применять операцию *::* указания области видимости. Например, в функции *main()* оператор *A :: x=z;* присваивает значение переменной *z* (из глобального пространства имен) переменной *x*, принадлежащей пространству имен *A*. Для объявления объекта *b* класса *M*, описание которого находится в пространстве имен *A*, необходимо также использовать операцию *::* указания области видимости при объявлении объекта: *A::M b;*.

Если необходимо часто применять операцию указания области видимости, то целесообразно использовать оператор ***using***, который позволяет получить доступ к конкретному имени в пространстве имен, либо ко всем именам пространства имен.

Для получения доступа к отдельному имени в пространстве имен используется оператор в следующем формате:

```
using имя_пространства_имен :: имя;
```

Например, для того, чтобы обеспечить доступ к переменной *x* пространства имен *A* из функции *main()* приведенной программы, достаточно выполнить оператор *using A :: x;*.

Для обеспечения доступа ко всем именам некоторого пространства имен используется оператор *using* в следующем формате:

```
using namespace  имя_пространства_имен;
```

После выполнения такого оператора указанное пространство имен добавляется в текущую область видимости.

Для уменьшения вероятности конфликта имен в последних версиях компиляторов библиотека C++ определена в собственном пространстве имен *std*. Использование оператора *using namespace std;* дает возможность добавить в текущую область видимости все имена, содержащиеся внутри библиотеки C++.

Существует возможность введения более одного пространства имен с одним и тем же именем. Это позволяет разделить пространство имен на несколько файлов или на несколько частей внутри одного файла.

Например:

```
namespace L
  {
    int x;
    . . .
  }
. . .
namespace L
  {
    float y;
    . . .
  }
```

Здесь объявлены две части одного пространства имен L.

В языке C++ можно объявить ***безымянное пространство имен***:

```
namespace
  {
    // объявления
  }
```

Безымянные пространства имен дают возможность полностью закрыть доступ извне к именам пространства имен, поскольку получить доступ к именам с помощью оператора *using* или операции :: невозможно, т. к. отсутствует имя у пространства имен.

Пример 2. Применение нескольких пространств имен.

```
#include <iostream>
// Определение первого пространства имен
namespace name_A
  {
    int i;
```

```
    class P
      {
        int x;
        public:
          P(int newy) {x=newy;}
          void izmx(int c) {x=c;}
          int dost() {return x;}
      };
      char str[]="\n Пространство имен";
  }
// Определение первой части второго пространства имен
namespace name_B
  {
    int y;
  }
// Определение второй части второго пространства имен
namespace name_B
  {
    int z;
  }
int main()
  {
    // Создание объекта b класса P
    name_A :: P b(2);
    cout << "\nЗначение элемента данных x объекта b: " << b.dost();
    cout << endl;
    b.izmx(15);
    cout << "Новое значение элемента данных x объекта b: " << b.dost();
    // Задание видимости строки str в текущем пространстве имен
    using name_A :: str;
    cout << str;
    cout << endl;
    // Задание видимости пространства имен name_A в текущей
    // области видимости
    using namespace name_A;
    for (i=1;i<11;i++)
      cout << i << " ";
    // Использование обеих частей второго пространства имен
    name_B :: y=100;
    cout << "\n name_B :: y=" << name_B :: y;
    name_B :: z=200;
    cout << "\n name_B :: z=" << name_B :: z;
    // Присоединение пространства имен name_B к текущему пространству имен
    using namespace name_B;
    P by(y), bz(z);
```

```
    cout << "\n Значение элемента данных x объекта by:" << by.dost();
    cout << "\n Значение элемента данных x объекта bz:" << bz.dost();
    cout << endl;
    return 0;
}
```

В результате работы программы на экран будет выведено следующее:

```
Значение элемента данных x объекта b:2
Новое значение элемента данных x объекта b:15
Пространство имен
1 2 3 4 5 6 7 8 9 10
name_B :: y=100
name_B :: z=200
Значение элемента данных x объекта by:100
Значение элемента данных x объекта bz:200
```

В приведенной программе определены два пространства имен: *name_A* и *name_B*. Кроме того, существует глобальное пространство имен программы. В пространстве имен *name_A* видимы класс *P*, переменная *i* и строка *str*. Пространство *name_B* состоит из двух частей. В первой части пространства *name_B* видима переменная *y*, а во второй — переменная *z*. Описание функции *main()* находится в глобальном пространстве имен.

При расширении области видимости с помощью оператора *using* необходимо следить за тем, чтобы в полученном совместном пространстве имен отсутствовали одинаковые имена. В случае наличия таковых действующим является имя, которое объявлено в текущем пространстве имен.

15.2 Обработка исключений

Исключительные ситуации (*exception handling*) представляют собой события, возникающие в программе и приводящие к ненормальной ее работе или ошибкам времени исполнения.

Исключения встроены в язык C++ для обработки ошибочных ситуаций, таких как ошибки конструктора или перегруженной операции, которые иным образом обработаны быть не могут. Обработка исключений позволяет автоматизировать большую часть кода для обработки ошибок, для чего раньше требовалось ручное кодирование. Обслуживаются только так называемые ***синхронные исключения***, которые возникают *внутри* программы. Внешние события, такие как нажатие клавиш <*Ctrl*>+<*C*>, исключениями не являются.

Когда программа встречает ненормальную ситуацию, на которую она не рассчитана, можно передать управление другой части программы, способной справиться с этой проблемой, и либо продолжить выполнение программы, либо завершить работу.

Выброс исключения (*exception throwing*) представляет собой событие, приводящее к передаче управления в секцию кода, содержащую функцию-обработчик данного исключения. Выброс исключения позволяет собрать в точке выброса информацию, которая может оказаться полезной для диагностики причин, приведших к нарушению нормального процесса функционирования программы.

Как правило, исключение должно выбрасываться в следующих случаях:

- нет другого способа сообщить об ошибке (например, в конструкторах, перегруженных операциях и т. д.);
- ошибка неисправима (например, недостаток памяти);
- ошибка непонятна или неожиданна, в связи с чем ее тестирование затруднено.

В языке C++ предусмотрено три ключевых слова для обработки исключительных ситуаций: ***try*** (контролировать), ***catch*** (ловить) и ***throw*** (генерировать, порождать, бросать, посылать, формировать). Код, способный сгенерировать исключение, должен исполнятся внутри фигурных скобок, следующих за ключевым словом *try*. Если блок *try* обнаруживает исключение внутри этого блока кода, происходит программное прерывание и выполняется указанная ниже последовательность действий:

1. Программа ищет подходящий обработчик исключения.
2. Если обработчик найден, стек очищается и управление передается обработчику исключения.
3. Если обработчик не найден, вызывается функция *terminate()* для завершения программы.

Возникшая исключительная ситуация обрабатывается внутри блока *catch*, который должен следовать непосредственно за блоком *try*. Для каждого исключения необходимо предусмотреть свой обработчик, иначе может возникнуть ненормальное завершение программы. Обработчики исключений просматриваются по порядку и управление передается тому обработчику, тип аргумента которого совпадает с типом вызвавшего этот обработчик исключения. Если в теле обработчика исключения не содержится операторов перехода, то выполнение программы продолжается с точки, следующей непосредственно за последним блоком *catch*.

Блоки *try* и *catch* имеют следующие форматы:

```
try
  {
   // Код, генерирующий исключение
  }
catch ( Type   X )
  {
   // Обработчик исключения X типа Type, которое могло быть
   // ранее сгенерировано внутри предыдущего блока try
  }
// Обработчики других исключений предыдущего блока try
catch(...)
  {
   // Обработчик любого исключения предыдущего блока try
  }
```

Размеры блока *try* могут колебаться в больших пределах: от нескольких операторов до всей функции *main()*. В последнем случае вся программа будет охвачена обработкой исключений.

Оператор ***throw*** имеет следующую форму записи:

```
throw исключение;
```

В данной инструкции *исключение* означает сгенерированное значение. Выполнение этой инструкции должно происходить либо внутри блока *try*, либо в функции, вызванной из блока *try*. В случае генерации исключения, которому не соответствует ни одна инструкция *catch*, может произойти аварийное завершение программы. При возникновении такого необработанного исключения вызывается функция *terminate()*, которая по умолчанию вызывает функцию *abort()*, приводящую к завершению выполнения программы.

Пример 1. Обработка исключений.

```
#include <iostream>
int main(void)
  {
    cout << "Начало программы\n";
    try                          // начало блока try
      {
        cout << "Находимся внутри блока try\n";
        throw 13;                // генерация ошибки
        cout << "Этой строке управление передано не будет ";
      }
    catch (int exception)
      {
        cout << "Перехвачена ошибка со значением ";
        cout << exception << "\n";
      }
    cout << "Окончание программы";
    return 0 ;
  }
```

В результате работы программы на экране можно наблюдать следующие результаты:

```
Начало программы
Находимся внутри блока try
Перехвачена ошибка со значением 13
Окончание программы
```

Как отмечалось, тип исключения должен соответствовать типу, указанному в инструкции ***catch***. Если, например, исправить вторую строку исходного кода в операторе ***try*** на

```
throw 13.0 ;
```

то программа выдаст на экран следующее:

```
Начало программы
Находимся внутри блока try
Abnormal program termination
```

Исключение может быть сгенерировано из функции, вызванной из блока *try*:

Пример 2. Генерирование исключения из функции.

```
#include <iostream>
void test_exception(int t)
  {
    cout << "Находимся внутри test_exception";
    if (!t) throw t;
  }

int main(void)
  {
    cout << "Начало программы\n";
    // начало блока try
    try
      {
        cout << "Находимся внутри try-блока\n";
        test_exception(2);
        test_exception(1);
        test_exception(0);
      }
    catch(int i)
      {
        cout << "Ошибка: i=" << i;
      }
    cout << "Окончание программы";
    return 0;
  }
```

В результате получим:

```
Начало программы
Находимся внутри try-блока
Находимся внутри try-блока
Находимся внутри try-блока
Ошибка:i=0
Окончание программы
```

В языке C++ имеется ряд ***дополнительных возможностей*** для управления обработкой исключительных ситуаций. Они позволяют сделать ее более гибкой и эффективной.

В некоторых случаях возникает необходимость перехватывать ***все исключения***. Для реализации этой возможности используется оператор вида:

```
catch(...)
  {
    // обработка исключения любого типа
  }
```

Данный оператор должен описываться в программе последним из операторов *catch*, следующих за блоком *try*, иначе не будут выполняться обработчики исключений, находящиеся за этим оператором.

Пример 3. Использование инструкции *catch(...)*.

```
#include <iostream>
void exception_handler(int  t)
  {
    try
      {   // начало try-блока
        if (!t) throw t;                  // генгерация int
        if (t==1) throw 1.11;             // генгерация double
        if (t==2) throw "Ошибка!";        // генгерация char *
      }
    catch(char *s)                        // обработчик char *
      {
        cout << s << "(t=2)\n";
      }
    catch(...)                            // обработчик любого исключения
      {
        cout << "Ошибка в программе!\n";
      }
  }   // exception_handler
int main()
  {
    cout << "Начало программы\n";
    exception_handler(0);
    exception_handler(1);
    exception_handler(2);
    cout << "Окончание программы";
    return 0;
  }
```

Данная программа выдает на экран следующее:

```
Начало программы
Ошибка в программе!
Ошибка в программе!
Ошибка!(t=2)
Окончание программы
```

Как видим, обработчик *catch(...)* вызывается случаях, когда не найдется другого обработчика соответствующего типа исключения.

Если возникает необходимость ограничить типы исключений для функции, которые она может генерировать, или же вообще запретить ей генерировать исключения, то для задания этих ограничений необходимо добавить к определению функции ключевое слово ***throw*** следующим образом:

```
тип имя_функции (список аргументов) throw (список типов)
  {
    // тело функции
  }
```

Определенная таким образом функция может генерировать только те исключения, типы которых указаны в *списке типов* в *throw*. Они должны следовать в списке, разделенные запятыми. При попытке сгенерировать исключение, тип которого не указан в *throw*, произойдет аварийное завершение программы. Для запрета генерации функцией исключений необходимо объявить ее, как и в предыдущем случае, но список типов в *throw* оставить пустым.

Пример 4. Задание ограничений на генерацию исключений функцией.

```
#include <iostream.h>
void exception_handler(int t) throw (char)
  {
    cout << "Находимся внутри exception_handler\n";
    if (!t) throw 'A';          // генерация char
  }
int main()
  {
    cout << "Начало программы\n";
    try                         // начало блока try
      {
        exception_handler(1);
        exception_handler(0);
      }
    catch(char ch)
      {
        cout << "Сгенерировано исключение типа char\n";
      }
    cout << "Окончание программы";
    return 0;
  }
```

Результаты работы программы представлены ниже:

```
Начало программы
Находимся внутри exception_handler
Находимся внутри exception_handler
Сгенерировано исключение типа char
Окончание программы
```

Если в операторе *throw* оставить скобки пустыми, то произойдет аварийное завершение программы, так как в этом случае функции *exception_handler* запрещено генерировать любые исключения.

Отметим, что функция, которая ограничена на выбрасывание исключений некоторых типов, может иметь в своем теле *try*-блок и генерировать любые ис-

ключения до тех пор, пока они перехватываются инструкциями *catch* внутри самой функции.

Может возникнуть необходимость генерации исключения из блока, который обрабатывает исключение. Это достигается путем указания оператора *throw* без параметров. В этом случае текущее исключение будет передано для обработки во внешнюю последовательность *try/catch*.

Пример 5. Повторная генерация исключений.

```
#include <iostream>
void exception_handler(int x)
  {
    try
      {
        if(!x) throw x;          // генерация int
      }
    catch(int)                   // перехват int
      {
        cout << "Перехват исключения типа int внутри
exception_handler\n";
        throw;                   // повторная генерация исключения
      }
  }

int main()
  {
    cout << "Начало программы\n";
    try                          // начало try-блока
      {
        exception_handler(1);
      }
    catch(int)
      {
        cout << "Перехват исключения типа int внутри main\n";
      }
    cout << "Окончание программы";
    return 0;
  }
```

Программа выдает на экран следующее:

```
Начало программы
Перехват исключения типа int внутри exception_handler
Перехват исключения типа int внутри main
Окончание программы
```

Рассмотрим комплексный пример использования обработки исключительных ситуаций. Приведенная ниже программа выполняет считывание из файла двух чисел и деление первого числа на второе. Ошибки в программе могут возникнуть в следую-

щих случаях: ошибка открытия файла, ошибка при делении на ноль и ошибка при выводе результата деления на экран.

Пример 6. Обработка исключений.

```
#include <stdio.h>
#include <fstream.h>
int main()
  {
    try
      {
        float  a,b,c;
        ifstream in("input.txt");              //открытие файла
        if(!in)
          throw "Ошибка открытия файла!";
        in >> a >> b;                          // чтение чисел
        in.close();                            // закрытие файла
        if(!b)
          throw "Деление на ноль невозможно!";
        c = a/b;
        if(printf("Результат деления:%f",c)  <= 0)
          throw "Ошибка при выводе результата!";
      }
    catch(char * s)
      {
        puts(s);
      }
    return 0;
  }
```

Обработка исключительных ситуаций в языке C++ представляет собой хорошо структурированное средство для обнаружения ошибок времени исполнения. Используя этот подход, можно обезопасить себя от всякого рода неожиданностей и повысить надежность программ. Следует иметь в виду, что интенсивное применение обработки исключений приводит к большим непроизводительным затратам по размеру кода и по времени выполнения. Это справедливо даже для операционных систем типа Microsoft NT, которая поддерживает обработку исключений на уровне операционной системы. Лучше всего использовать исключения тогда, когда непроизводительные затраты не оказывают отрицательного влияния.

15.3. Динамическая идентификация типов

Динамическая идентификация типов (*Run-Time Type Identification*, RTTI) является относительно новым средством языка C++, которое обеспечивает возможность идентификации типов в процессе выполнения программ. Динамическая идентификация типов требуется при выполнении программ, в которых используется *динамический полиморфизм*.

Напомним, что динамический полиморфизм в C++ реализуется при работе с виртуальными функциями с опорой на иерархию классов и указатели базовых классов. Для поддержки динамического полиморфизма виртуальные функции вызывают через *указатель базового класса*, используемый в качестве ссылки на объект производного класса. Указатель базового класса может использоваться для указания либо на объект базового класса, либо на объект производного класса. Поэтому информация о типе объекта, адресуемого с помощью указателя базового класса, заранее не известна и может быть уточнена только во время выполнения программы.

Механизм динамической идентификации типов служит для этих целей и реализуется он с помощью оператора ***typeid***. Оператор *typeid* имеет следующие формы записи:

1. *typeid(выражение)*
2. *typeid(имя_типа)*

При использовании оператора *typeid* в программу следует включить заголовок ***<typeinfo>*** с помощью директивы препроцессора *#include*. Для каждого класса имеется связанный с ним объект типа *type_info*, который содержит различную информацию о классе. Описание *класса* ***type_info*** при удалении из него зависимых от реализации языка элементов будет иметь следующий вид:

```
class type_info
  {
    public:
      virtual    ~typeinfo();                    // конструктор
      bool operator==(const type_info&) const;   // операция сравнения
      bool operator!=(const type_info&) const;   // операция сравнения
      bool before(const type_info&) const;       // упорядочение
      const char* name() const;                  // имя типа
}
```

В случае первой наиболее часто используемой формы записи оператор *typeid* для заданного аргумента *выражение* возвращает ссылку на объект типа *type_info*, который представляет *тип* обозначенного с помощью *выражения* объекта.

В случае использования второй формы записи оператор *typeid* по заданному операнду *имя_типа* возвращает ссылку на объект типа *type_info*, который представляет *имя типа*.

Пример 1. Определение типа объектов.

```
#include <iostream>
#include <typeinfo>
void main()
  {
    char C;
    double D;
    cout << "type of C is " << typeid (C).name() << endl;
    cout << "type of C+D is " << typeid(C+D).name() << endl;
  }
```

При выполнении программы на экран будут выданы следующие строки:

```
type of C is char
type of C+D is double
```

Отметим, что при втором использовании оператора *typeid* в качестве аргумента задано выражение *C+D*, тип которого определяется после автоматического преобразования типов.

Приведенная в описании класса *type_info* функция *before()* возвращает истину, если вызывающий объект в порядке сортировки расположен раньше объекта, указанного в качестве аргумента. Перегруженные операторы == и != обеспечивают возможность сравнения типов.

Пример 2. Сравнение и анализ порядка следования типов.

```
#include <iostream>
#include <typeinfo>
class A1 { };
class A2 : A1 { };
void main()
  {
    char C;
    double D;
    if (typeid( C ) == typeid( D ))
      cout << "C and D are the same type." << endl;
    else cout << "C and D are not the same type" << endl;
    if  (typeid(char).before(typeid(double)))
      cout << typeid(char).name() << " before " <<
typeid(double).name();
    else cout << typeid(double).name() << " before " <<
typeid(char).name();
    cout << endl;
    if  (typeid(A1).before(typeid(A2)))
      cout << typeid(A1).name() << " before " << typeid(A2).name();
    else cout << typeid(A2).name() << " before " << typeid(A1).name();
  }
```

В результате выполнения программы на экран будут выданы следующие строки:

```
C and D are not the same type
char before double
A1 before A2
```

Приведенные примеры иллюстрируют возможности оператора *typeid* на примере объектов, тип которых известен заранее. В действительности он наиболее полезен в случае задания в качестве аргумента указателя полиморфного базового класса. При этом оператор *typeid* возвращает тип объекта (базового или производного класса), на который указывает указатель.

В качестве рекомендаций по использованию динамической идентификации типов заметим: не следует использовать динамическую идентификацию типов случаях, когда можно применить виртуальные функции.

Пример 3. Использование динамической идентификации типов.

Рассмотрим пример динамической идентификации типов, когда тип заранее не известен и определяется при выполнении программы при обращении к случайной функции *rand()*. При обращении к функции *rand()%2* получается последовательность псевдослучайных целых, итзменяющихся в диапазоне от 0 до 1.

```
#include <iostream>
#include <stdlib>
#include <typeinfo>
class parent
  {
                                    //объявление базового класса
    public:
      int k;
      parent(int a) {k = a;}        // конструктор класса
      virtual void fv()             // виртуальная функция
        {
          cout << "call fv() of base class: ";
          cout << k << endl;
        }
  };
class child1: public parent         // производный класс
  {
    public:
      child1(int x): parent(x) {}
      void fv()
        {
          cout << "call fv() of derived class child1: ";
          cout << (k + 1) << endl;
        }
  };
parent *Generat()
  {
    if (rand()%2==0)
    return new parent(5);
    else return new child1(8);
  }
int main ()
  {
    int i;
    parent *p;                      //объявление указателя на базовый класс
    randomize();                    //инициализация генератора случайных чисел
    for (i=0;i<5;i++)
      {
        p= Generat();               //создание объекта
        cout << typeid(*p).name()<<endl;
```

```
        p->fv();                    // вызов виртуальной функции
      }
  }
```

В результате выполнения программы на печать могут быть получены следующие строки:

```
parent
call fv() of base class: 5
child1
call fv() of derived class child1: 9
parent
call fv() of base class: 5
parent
call fv() of base class: 5
child1
call fv() of derived class child1: 9
```

Реальная последовательность строк зависит от последовательности генерируемых псевдослучайных чисел.

Заметим, что здесь мы *динамически* идентифицируем имя типа объекта по значению указателя на базовый тип с помощью функции *typeid()*. Фактически мы вполне можем обойтись без использования динамической идентификации типа, поскольку все то же самое нам обеспечивает обращение к виртуальной функции *fv()*.

В качестве варианта *обоснованного использования динамической идентификации* типов можно назвать расширение функциональных возможностей базового библиотечного класса и создания производного класса, когда невозможно или нецелесообразно добавлять виртуальные функции к базовым классам иерархии. Например, возможное схематичное построение кода имеет вид:

```
#include <typeinfo>
class TmyLib :  TbaseLib {};
class TmyClass : public TmyLib
  {
    public:
    virtual void MyMethod1();
  }
void Actions(TbaseLib *p)
  {
    if(typeid(p) == typeid(TmyClass))
    ((TmyClass*)p)-> MyMethod1();
    // ...
  }
```

Для повышения надежности программирования и эффективности создаваемого кода аппарат динамической идентификации типов следует использовать в случаях, когда тип объекта невозможно определить при компиляции или с помощью механизма виртуальных функций.

15.4. Приведение типов

В языке C++ кроме доставшихся по наследству от языка C операторов, имеется ряд новых операторов приведения типов. К ним относятся операторы: *dynamic_cast, const_cast, reiterpret_cast* и *static_cast*.

Оператор ***dynamic_cast*** предназначен для осуществления динамического приведения типов в процессе выполнения программы. В этом смысле указанный оператор очень близок к динамической идентификации типов. Оператор *dynamic_cast* имеет следующую форму записи:

```
dynamic_cast<целевой_тип> (выражение)
```

Здесь *целевой_тип* представляет собой тип, которым должен стать тип параметра *выражение* после выполнения оператора приведения типов. Целевой тип должен иметь тип указателя или ссылки и в результате выполнения оператора параметр *выражение* должен получить тип указателя или ссылки.

Оператор *dynamic_cast* в основном служит для выполнения приведения полиморфных типов. Например, если имеются два полиморфных класса: родительский класс *Parent* и производный от него класс *Child*. В этом случае с помощью оператора *dynamic_cast* можно привести тип указателя *Child** производного класса к типу указателя *Parent** родительского класса. Это обусловлено тем, что указатель родительского класса может указывать на объект производного класса.

С помощью оператора *dynamic_cast* можно привести также тип указателя *Parent** к типу указателя *Child** при условии, что указатель *Parent** указывает на объект типа *Child*. В общем случае оператор *dynamic_cast* выполняется *успешно*, когда указатель или ссылка (параметр выражения) после приведения типов становится указателем или ссылкой на объект целевого типа или на производный от целевого типа объект. При удачном приведении типов оператор *dynamic_cast* с использование указателей возвращает единицу, в противном случае ноль. Если в операторе использовались ссылки, то при неудачном приведении типов возникает исключительная ситуация *bad_cast*.

Пример 1. Динамическое приведение типов.

```
#include <iostream>
class Parent
  {
    public:
      virtual void fv(){…};
  };
class Child : public Parent
  {
    public:
      void fv(){…};
  };
int main(void)
  {
    Parent *pP, oP;
```

```
    Child *pC, oC;
    bool R;
    pP=&oC; // указывает на объект производного класса
    R = dynamic_cast<Child*>(pP);
    if (R) cout << "Successful type promotion"<< endl;
    else cout << "Not successful type promotion"<< endl;
    pP=&oP; // указывает на объект родительского класса
    R = dynamic_cast<Child*>(pP);
    if (!R) cout << "Not successful type promotion"<< endl;
    else cout << "Not successful type promotion"<< endl;
    return 0;
}
```

В результате выполнения приведенной программы на экран будут выданы следующие строки:

```
Successful type promotion
Not successful type promotion
```

Первая строка указывает на успешное преобразоваие типа при первом выполнении оператора *dynamic_cast*, вторая строка — на неуспешное приведение типов при следующем выполнении оператора.

Заметим, что попытка записи следующего оператора присваивания

```
pC=&oP;
```

приводит к ошибке при компиляции ввиду недопустимости адресации указателем производного класса объекта родительского класса.

Оператор ***const_cast*** служит для выполнения операции приведения типов с целью явной подмены описателей *const* и/или *volatile* (переменный). Указанный оператор имеет следующий формат записи:

```
const_cast<целевой_тип> (выражение)
```

При этом целевой тип должен совпадать с типом выражения, за исключением описателей *const* или *volatile*. Чаще всего оператор *const_cast* используют для того, чтобы значение лишить описателя *const*.

Оператор ***reinterpret_cast*** служит для преобразования указателя одного типа в указатель другого типа и позволяет приводит указатель к типу целого и целое к типу указателя. Оператор *reinterpret_cast* имеет следующую форму записи:

```
reinterpret_cast<целевой_тип> (выражение)
```

Здесь *целевой_тип* может быть типом указателя, ссылки, целым (*char, short, int, long, enum*) и вещественным (*float, double, long double*) типом. *Выражение* может принадлежат типу указателя, ссылки, целой или вещественной переменной, если *целевой_тип* является типом указателя или ссылки. Кроме того, *выражение* может быть указателем или ссылкой, если *целевой_тип* является целым или вещественным типом. Оператор *reinterpret_cast* целесообразно применять для приведения внутренне несовместимых типов указателей.

Пример 2. Использование оператора *reinterpret_cast*.

```
#include <iostream.h>
int main(void)
  {
    int* k;
    int p=123;
    k=reinterpret_cast<int*> (p);      // k присваивается значение адрес
    cout<<reinterpret_cast<int*> (p)<<endl;
    cout<<k;
    return 0;
  }
```

После выполнения программы возможна выдача следующих двук одинаковых строк:

```
0x0000007b
0x0000007b
```

Выдаваемые строки (адреса) зависят от местоположения значения переменной p.

Оператор ***static_cast*** предназначен для приведения типов объектов неполиморфных классов. Оператор *static_cast* имеет следующую форму записи:

```
static_cast<целевой_тип> (выражение)
```

Пример 3. Использование оператора *static_cast*.

```
#include <iostream.h>
int main(void)
  {
    int k;
    float p=123.5;
    k=static_cast<int> (p);      // k присваивается значение 123
    cout<<static_cast<int> (p)<<endl;
    cout<<k;
    return 0;
  }
```

В результате выполнения программы на экран буду выданы строки:

```
123
123
```

С помощью оператора *static_cast* можно выполнить, к примеру, приведение типа указателя базового класса к типу указателя производного класса. Кроме того, этот оператор применим для выполнения любой стандартной операции преобразования в процессе компиляции программы, а не при ее выполнении.

Контрольные вопросы и задания

1. В чем различие между двумя формами использования оператора *using*?
2. Поясните назначение безымянного пространства имен.

3. Каково назначение пространства имен?

4. Чем может быть обусловлено создание нескольких пространств имен с одинаковыми именами?

5. Составте программу, в которой отсутствует глобальное пространство имен.

6. Приведите пример программы, в которой определено безымянное пространство имен.

7. Что называется исключительной ситуацией?

8. Поясните смысл выражения «выброс исключения»?

9. В каких случаях генерируется исключение?

10. Какие ключевые слова предусмотрены в языке C++ для обработки исключительных ситуаций?

11. Укажите назначение операторов *try*, *catch* и *throw*.

12. Приведите пример альтернативы операторам *try/catch*.

13. Поясните, почему нецелесообразно использовать выброс исключения из конструктора класса.

14. Когда целесообразно применять обработку исключений?

15. Составьте программу вычисления $y = \sqrt{\sqrt{x_1} + \sqrt{x_2}}$, где исходные величины вводятся из файла. Используйте обработку исключений при извлечении квадратного корня.

16. Составьте программу по условию предыдущего задания так, чтобы обработка исключений выполнялась при открытии файла с исходными данными.

17. Составьте программу вычисления частного от деления значений двух переменных, вводимых с клавиатуры. Предусмотрите обработку исключения в случае деления на ноль.

18. В чем суть динамической идентификации типов?

19. Укажите рекомендации по целесообразному применению динамической идентификации типов.

20. Укажите назначение объекта типа *type_info*.

21. Приведите основные формы записи оператора *typeid* и дайте им краткую характеристику.

22. Приведите примеры операторов сравнения и анализа порядка следования типов.

23. Укажите назначение операторов приведения типов.

24. Приведите форму записи операторов приведения типов.

25. Поясните разницу между статическим и динамическим приведением типов.

26. Составьте программу с использованием динамического приведения типов.

27. Составьте программу, в которой выполняется динамическая идентификация типа объектов.

16. РАЗРАБОТКА ПРИЛОЖЕНИЙ В BORLAND C++ BUILDER

В этом разделе рассматривается технология разработки приложений (прикладных программ) в системе программирования Borland C++ Builder. Такой выбор обусловлен тем, что это — одна из современных систем, обладающих развитыми возможностями и поддерживающих концепции объектно-ориентированного и визуального программирования.

16.1. Общая характеристика системы

Развитие вычислительной мощности современных компьютеров позволяет решать на ЭВМ задачи большой сложности. Кроме широкого спектра прикладных задач появилась возможность создания удобного графического интерфейса современных операционных систем для общения пользователя с ЭВМ. Одним из обязательных требований к современным программным продуктам является обеспечение такого интерфейса. Современные операционные системы (ОС) работают в графическом режиме. Текстовый режим используется в основном в программах, разработанных ранее.

Разработка программ, поддерживающих текстовый режим отображения информации, большого труда не представляет. Во всех приведенных примерах использовался именно такой режим. Он удобен при работе программ под управлением операционной системы MS DOS. В современных операционных системах, например, MS Windows, обладающих развитым графическим интерфейсом, текстовый режим отображения информации не дает пользователю полного удовлетворения от работы программы.

Разработка современной программы для работы под управлением MS Windows на языке C++ без использования дополнительных библиотек классов является сложной задачей даже для опытных программистов. Разработчик должен учитывать множество особенностей работы самой ОС при разработке программы. Сложными являются этапы тестирования и отладки. При этом существенно повышается вероятность возникновения ошибок, а на разработку программы требуется много времени.

Такое положение дел не могло удовлетворять программистов и фирмы, производящие системы программирования. Вначале для облегчения труда программистов в состав систем программирования стали включаться *библиотеки классов*, позволяющие программисту довольно быстро создавать такие объекты как окно, кнопки,

меню и т. д. При этом технология программирования не менялась, программист, как и раньше, составлял текст программы, который получался существенно проще и не требовалось программирования по созданию многих системозависимых компонентов.

Со временем включаемые в состав систем разработки программ библиотеки классов становились мощнее. Рост сложности решаемых задач требовал изменения самой технологии программирования. Появляется концепция визуального программирования, которая впервые была реализована в системе Visual Basic фирмы Microsoft. Фирма Borland воплотила технологию визуального программирования в системах Delphi, Java Builder, C++ Builder. Система программирования Borland C++ Builder впервые поступила на рынок в 1997 году, в ней реализована концепция визуального программирования на языке C++.

Технология *визуального программирования* позволяет пользователю визуально наблюдать в процессе разработки основные компоненты программы. С помощью мыши можно создавать и перемещать отдельные компоненты, изменять их размер, а также изменять свойства объекта. При этом программирования не требуется, система сама формирует исходный текст программы и автоматически вносит в него исправления. При такой технологии задача программиста сводится к программированию определенных функций для объектов и внесению изменений в автоматически созданный текст программы. Тем самым существенно сокращается время разработки программ.

Получая существенный выигрыш во времени разработки и надежности созданной программы, мы *проигрываем в эффективности* сгенерированного машинного кода. Созданные таким образом программы обычно медленнее работают, требуют для своей работы большее количество памяти. Однако быстрый рост вычислительных мощностей ЭВМ делает эти недостатки менее значимыми.

После инсталляции на компьютер пакета Borland C++ Builder можно создавать свои приложения (программы, работающие под управлением ОС MS Windows), используя технологию визуального программирования. После запуска C++ Builder на экране появляется окно (рис. 16.1), за исключением окна редактора кода.

То, что мы видим на экране, называется *интегрированной средой разработки* (IDE — *Integrated Development Enviroment*). Пользователю сразу доступны следующие компоненты среды.

1. ***Команды главного меню***. Позволяют осуществить доступ к полному перечню команд, управляющих процессом разработки программ.

2. ***Панель инструментов*** (с быстрыми кнопками). Содержит наиболее часто используемые команды интегрированной среды, которые выведены для удобства работы на отдельную панель.

3. ***Палитра компонентов***. Содержит основные визуальные и невизуальные компоненты, которые программист может включать в приложение на этапе разработки. Эти операции проводятся с помощью мыши без программирования. Требуемый код программы генерируется автоматически.

4. ***Инспектор объектов*** — средство, позволяющее программисту быстро менять свойства объектов приложения, не прибегая к программированию. Изменения в программу вносятся автоматически.

5. ***Редактор форм***. Под формой здесь понимается окно, используемое в приложении, со всеми содержащимися в нем компонентами. Редактор форм позволяет

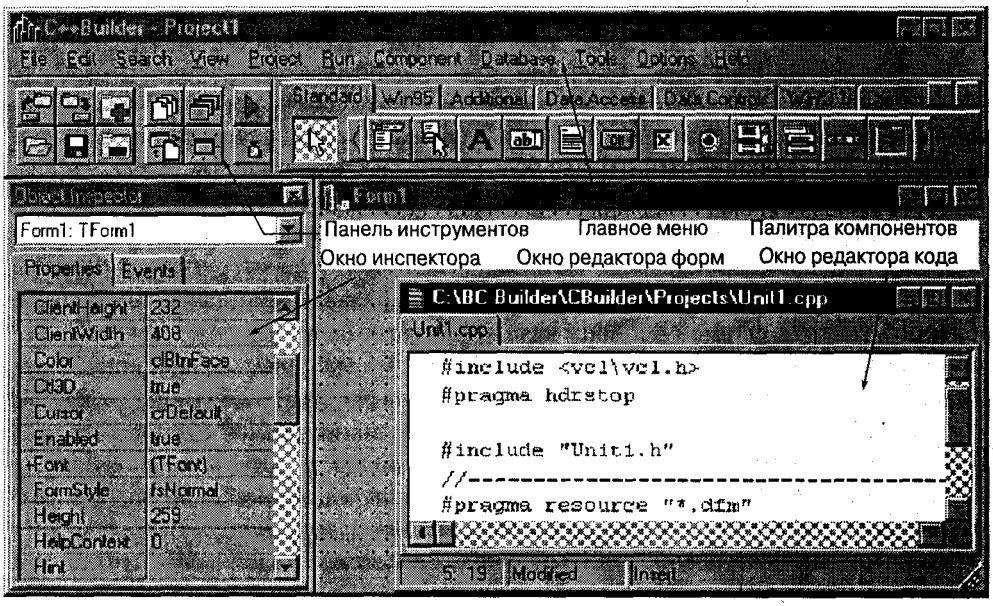


Рис. 16.1. Интегрированная среда разработки

создавать программисту окно требуемого вида и визуально размещать на нем требуемые компоненты.

6. ***Редактор кода*** предназначен для внесения изменений в исходном тексте автоматически сгенерированной программы. Изменения чаще всего требуются для расширения стандартных возможностей созданных объектов.

Кроме того, C++ Builder предоставляет *сервисные средства* по созданию приложения требуемого вида, *стандартные средства* по работе с файлами, управлению печатью и редактированию, *средства поиска информации* и просмотра различных компонентов программы и IDE, *средства по работе с проектом*, инструменты по *запуску и отладке* программы, средства по *созданию пользователем новых компонент*, инструменты для *управления работой баз данных*, всевозможные полезные *утилиты*, средства для *разработки программ в составе группы* программистов, справочную систему.

В целом C++ Builder — это мощный инструмент разработки Windows-приложений. Полное описание всех возможностей данной системы потребовало бы несколько томов. Цель данного раздела состоит в том, чтобы ознакомить читателя с технологией разработки программ в Borland C++ Builder.

16.2. Библиотека классов системы

В любой системе программирования библиотеки классов предназначены для облегчения труда программистов. В системе C++ Builder такой библиотекой является библиотка визуальных компонентов (VCL), ее мы сейчас и рассмотрим.

Причины появления VCL

Вначале первые программы, написанные для выполнения под управлением Windows, использовали API (*Aplication Programming Interface* — интерфейс прикладного программирования) для вызова различных системных функций операционной системы. Общее количество API-функций насчитывало более сотни. Их использование требовало знания многих системозависимых вещей. Интерфейс прикладного программирования предоставляет пользователю самый низкий уровень общения программы с операционной системой. Труд программиста при этом представляет утомительный рутинный процесс.

Чтобы освободить программиста от учета целого ряда системозависимых частей и облегчить его труд, фирмой Borland в начале 90-х годов была предложена первая библиотека классов OWL 1.0 (*Object Windows Library*). Со временем предлагались более мощные версии этой библиотеки. Библиотеки классов содержат в своем составе классы для работы с окнами, компонентами окон (кнопки, линии прокрутки, элементы редактирования и т. д.), графическими изображениями и т. д. Классы инкапсулируют наиболее часто используемые при программировании операции и освобождают разработчика от учета многих системных особенностей.

Широкое распространение также получила библиотека классов MFC (Microsoft Foundation Class) фирмы Microsoft. Между OWL и MFC имела место непрерывная шла конкуренция. Преимущества OWL состояли в том, что она была хорошо продумана с самого начала разработки и предоставляла программисту достаточно высокий уровень абстракции от API-интерфейса, концепция OWL четко соответствовала подходу ООП. Самый существенный недостаток при этом состоял в сложности ее изучения и понимания для новичков программирования. Главное преимущество MFC было в ее большей доступности для изучения и понимания, чем OWL. Не последнюю роль в ее распространении сыграло имя фирмы Microsoft.

Библиотека MFC была менее абстрактна и лежала ближе к API-интерфейсу. К сожелению, она не вписывалась в концепцию ООП и представляла собой, скорее хорошо подобранный набор классов, плохо стыкующихся друг с другом. Среди этих двух библиотек классов трудно выделить победителя. OWL и MFC получили широкое распространение и существенно облегчили труд программистов, разрабатывающих программы в традиционном стиле путем написания кода на языке программирования.

Переход к технологии визуального программирования потребовал создания совершенно новой библиотеки классов, поскольку предыдущие библиотеки не вписывались в предложенную концепцию. Появилась библиотека VCL (Visual Component Library) — библиотека визуальных компонентов, разработанная фирмой Borland сначала для системы Delphi, а затем и для C++ Builder.

Специфика функционирования VCL

Главная особенность новой библиотеки состоит в том, что в ее основе лежит концепция свойств, методов и событий. Используемыми в приложении компонентами можно управлять с помощью методов, свойств и событий.

Обратимся к рис. 16.1. Инспектор объектов отображает доступные для управления свойства того или иного объекта. Сам объект, для которого требуется изменить

свойства, можно выбрать, щелкнув на нем мышью или выбрав из списка в верхней части *Инспектора объектов*. ***Свойства*** объекта определяют его характеристики и поведение. Например, для объекта *форма* (*окно*) свойство *Caption* определяет название в виде текста, которое будет отображаться в верхней части окна, *Color* задает цвет окна, *Width* — его ширину и т. д. Таким образом, через *Инспектор объектов* программист может настроить объект требуемым образом, не прибегая к программированию.

Свойства объекта можно менять и в самой программе. Имя объекта, которое используется в программе, задается с помощью свойства *Name*. Имена свойств указаны в *Инспекторе объектов* и в документации. Сами имена изменить нельзя. Значения свойств в программе можно изменить, например, таким образом:

```
Form1->Caption="Суперпрограмма!";
```

После использования данного оператора в программе текст в верхней части окна будет заменен на заданный.

Другим важным элементом по управлению объектами являются ***методы***. В отличие от свойств, которые представляют собой переменные или объекты, методы представляют собой функции-члены класса (объекта), которые можно вызывать извне для управления объектом. Например, для того чтобы окно стало невидимым в процессе работы программы, можно вызвать следующий метод:

```
Form1->Hide();
```

Чтобы сделать окно вновь видимым, надо вызвать другой метод:

```
Form1->Show();
```

Отметим, что видимостью окна можно управлять с помощью изменения значения свойства *Visible* объекта:

```
Form1->Visible=false;
Form1->Visible=true;
```

Первый оператор делает окно невидимым, второй заставляет окно вновь появиться. Какой из двух перечисленных способов управления объектом выбрать — это дело вкуса программиста.

Полный перечень свойств и методов объекта из библиотеки VCL приводится в документации и во встроенной справочной системе Borland C++ Builder.

Третий элемент управления объектом — это ***события***. Сама операционная система MS Windows — это система, управляемая событиями. При работе операционной системы каждую секунду возникает множество различных событий. Например, это может быть перерисовка окна, активизация окна, нажатие кнопки и т. д. При возникновении событий программа или операционная система уведомляют друг друга о возникновении того или иного события посредством посылки сообщений. При получении сообщения оно обрабатывается требуемым образом.

Для каждого объекта из VCL существует свой перечень событий, для которых можно изменить стандартную реакцию программы. Изменить реакцию на обработку события можно используя вкладку *Events* инспектора объектов. Для этого необходимо в инспекторе объектов выбрать событие, реакцией на которое мы хотим управлять, и дважды на нем щелкнуть мышью. После этого открывается окно редактора кода, в нем находится авто-

матически созданная функция, которая будет вызываться при возникновении данного события. Программисту остается написать код тела этой функции. Визуально пример изменения стандартной обработки события активизации окна показан на рис. 16.2.

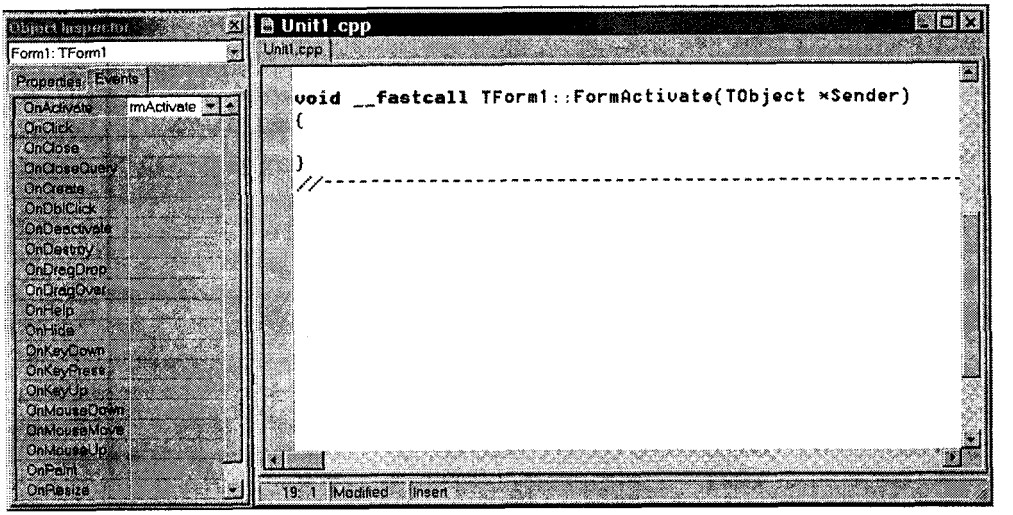


Рис. 16.2. Изменение стандартной обработки события

Отметим некоторые особенности использования VCL в C++ Builder. Сама библиотека VCL написана на Объектном Паскале. Это накладывает ряд ограничений на ее использование, поскольку Объектный Паскаль уступает по мощности реализации принципов ООП языку C++. VCL не допускает использования перегруженных функций, используемые функции-члены класса не имеют аргументов по умолчанию, классы из VCL не поддерживают множественное наследование. Указанные особенности, естественно, ведут к снижению гибкости использования VCL по сравнению с OWL и MFC. Однако преимущества концепции визуального программирования делают эти недостатки библиотеки VCL малозначимыми.

Структура библиотеки VCL

По своей структуре библиотека VCL хорошо продумана с самого начала. При этом максимально использован такой принцип ООП, как наследование. Для начала программирования достаточно иметь общее представление о структуре библиотеки. Набор классов представляет собой строго иерархическую систему. Упрощенно иерархия классов VCL изображена на рис. 16.3.

Класс ***TObject*** является базовым классом для всех классов библиотеки VCL. Он содержит *общие* для всех объектов свойства и методы, позволяющие создавать и удалять объект, обрабатывать сообщения и др. Большинство методов предназначено для внутреннего использования средой C++ Builder и не требует их прямого вызова из программы.

Класс ***TPersistent*** содержит *дополнительные* методы, учитывающие некоторые низкоуровневые детали, о которых разработчику знать необязательно. Кроме того, он содержит ряд методов, которые могут быть перегружены производными компонентами.

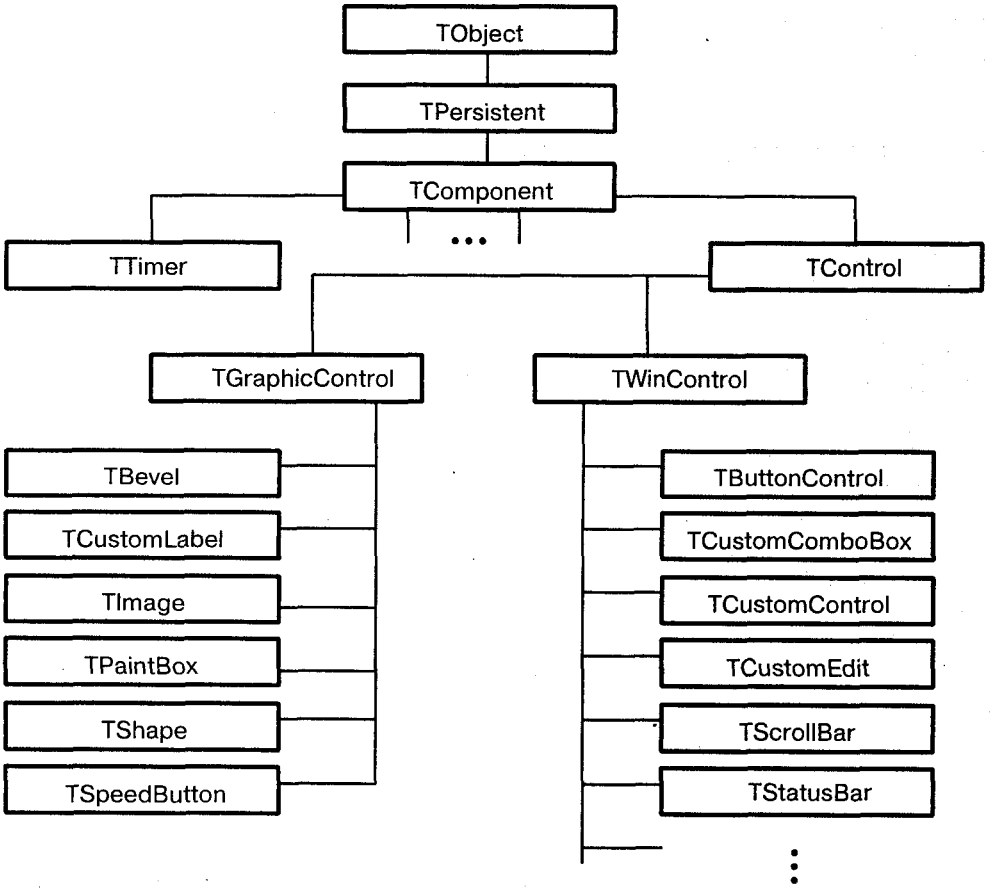


Рис. 16.3. Иерархия классо в VCL

Класс ***TComponent*** является общим предком для всех компонентных объектов. Содержит методы и свойства, обеспечивающие общее поведение для всех компонентов (создание, размещение на форме и др.). Невизуальные компоненты (*TTimer* и др.) создаются на основе данного класса.

Класс ***TControl*** является основой для создания всех визуальных компонентов и содержит в себе дополнительно свойства и методы для управления всеми визуальными компонентами (положение на экране, размер, вид объекта и др.).

Классы ***TWinControl*** и ***TGraphicControl*** разделяют визуальные компоненты на две основные группы. Все компоненты, произведенные от *TWinControl*, способны принимать фокус для ввода, содержат дескриптор и могут выступать в качестве базовых классов. Компоненты, производные от *TGraphicControl*, такими способностями не обладают.

Далее в иерархии следуют классы, производные от указанных. В их состав добавлены методы и свойства, учитывающие специфику функционирования отдельных объектов.

Весь перечень объектов, которые программист может включить в свою программу, можно просмотреть на палитре компонентов (рис. 16.1). Выбрав нужный компонент, разработчик, используя справочную систему или документацию, может получить

информацию о месте данного объекта в иерархии библиотеки и о доступных к использованию методах, событиях и свойствах.

16.3. Интегрированная среда разработки

Интегрированная среда разработки объединяет в своем составе все инструментальные средства, позволяющие пользователю создавать законченные программы. Напомним, что, согласно рис. 16.1, в состав среды входят:

- главное меню;
- панель инструментов;
- палитра компонентов;
- инспектор объектов;
- редактор форм;
- редактор кода.

Меню ***File*** (*Файл*) дает доступ к командам, показанным на рис. 16.4.

Многие из этих команд знакомы программистам, которые ранее работали с другими системами разработки программ. Отметим, что с помощью команды ***New...*** (*Новый...*) мож-

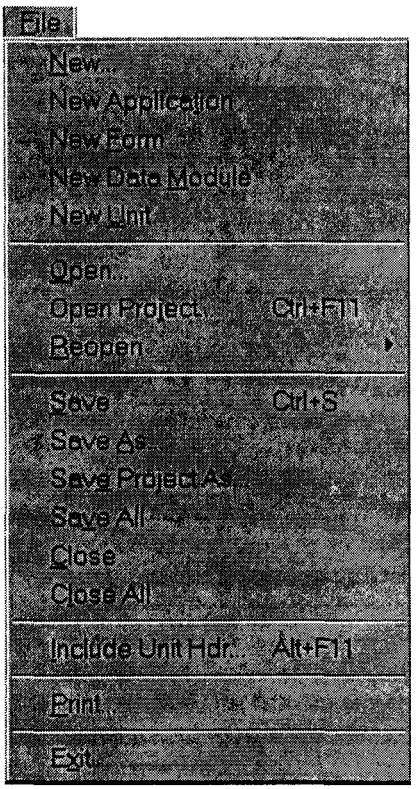


Рис. 16.4. Меню *File*

но создавать различные виды приложений, видов компонент, отдельных компонент, форм и т. д. При этом запускается процедура, называемая *Мастером*, которая в диалоговом режиме позволяет программисту в считанные секунды сгенерировать требуемый элемент.

Меню ***Edit*** (*Правка*) и ***Search*** (*Поиск*) должны быть знакомы пользователям, работавшим с современными редакторами и системами программирования. В них содержатся стандартные команды редактирования и поиска, используемые во многих современных программных системах.

Важным понятием при разработке программ на C++ Builder является понятие проекта. ***Проектом*** называется файл, содержащий информацию о файлах, из которых затем строится готовое приложение, и о том, каким образом это приложение строить. В проект может входить достаточно много файлов. При этом сами файлы могут иметь различные форматы, отличные от текстового. В начале работы в интегрированной среде в распоряжение программиста по умолчанию предоставлен стандартный проект. Если этот проект подходит, то на его основе строится приложение. Иначе программист сам может создать новый вид проекта в зависимости от решаемых задач. В процессе разработки приложения в проект добавляются новые файлы, пока поставленная задача не будет решена.

Просматривать проект, добавлять и удалять из него элементы можно используя команды меню ***View*** (*Вид*) и ***Project*** (*Проект*) (рис. 16.5 и рис. 16.6).

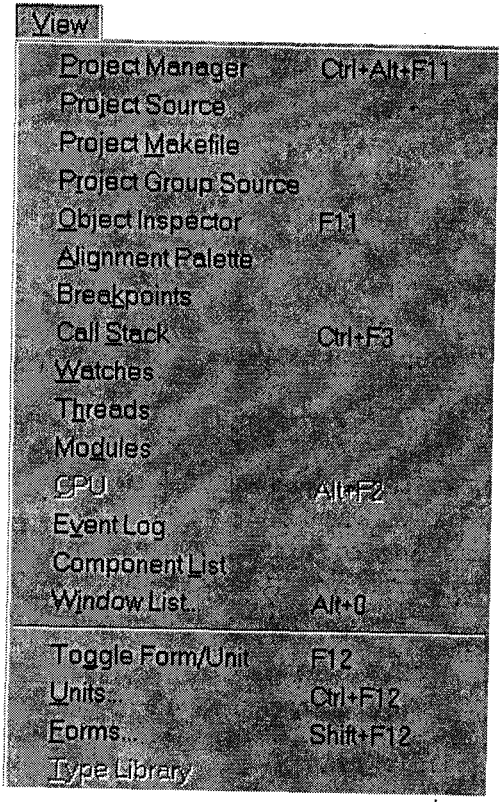


Рис. 16.5. Меню *View*

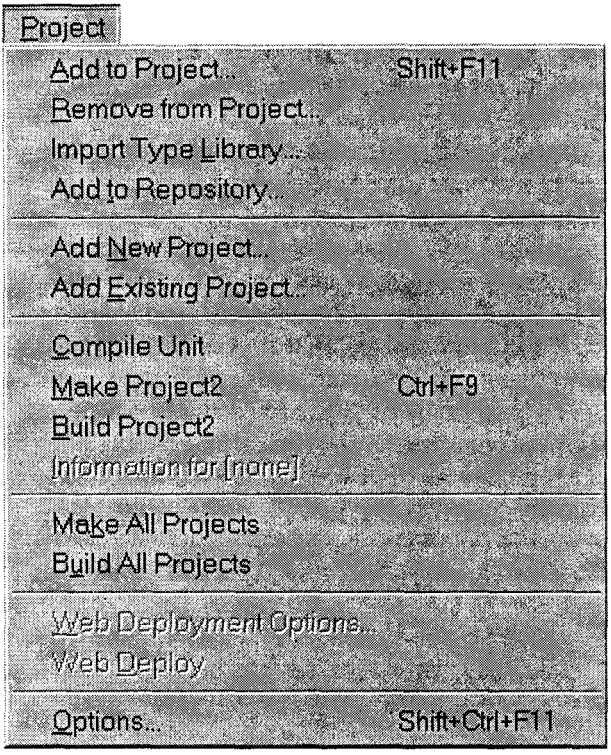


Рис. 16.6. Меню *Project*

По команде ***Project/Options...*** (*Проект/Параметры...*) можно требуемым образом настроить параметры проекта, то есть выбрать нужный вариант построения приложения.

Меню ***Run*** (*Выполнить*) содержит команды запуска и отладки (пошаговое выполнение, установка контрольных точек, инспектирование объектов, просмотр переменных и т. д.) сгенерированного приложения.

Меню ***Component*** (*Компонент*) содержит команды, позволяющие программисту быстро создавать свои собственные компоненты и добавлять их на палитру компонентов для дальнейшего использования.

Меню ***Database*** (*База данных*) содержит команды по управлению разработкой приложений баз данных. В меню ***Tools*** (*Сервис*) входят команды настройки подсистем интегрированной среды разработки IDE, запуска графического редактора, просмотра и редактирования баз данных и др.

Меню ***Workgroups*** (*Рабочие группы*) предоставляет доступ к командам управления разработкой приложения в составе групп программистов. С помощью меню *Help* можно получить доступ к командам справочной системы по C++ Builder.

Панель инструментов позволяет осуществлять быстрый вызов наиболее часто используемых команд IDE (рис. 16.7).

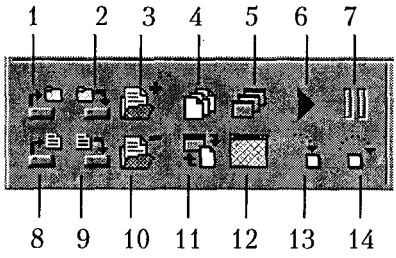


Рис. 16.7. Панель инструментов

Цифрами на рисунке обозначены следующие кнопки:
1. Открыть проект;
2. Сохранить все;
3. Добавить файл в проект;
4. Выбрать модуль с текстом программы из списка;
5. Выбрать форму из списка;
6. Запустить программу на решение;
7. Остановить выполнение программы;
8. Открыть;
9. Сохранить файл;
10. Удалить файл из проекта;
11. Переключиться на редактор форм иди редактор кода;
12. Создать новую форму;
13. Трассировка программы;
14. Пошаговое выполнение программы.

Палитра компонентов позволяет пользователю визуально выбрать любой объект из библиотеки классов VCL и разместить его в своем приложении. Коротко рассмотрим некоторые из них. На рис. 16.8 приведены объекты из вкладки ***Standard*** (Стандартная).

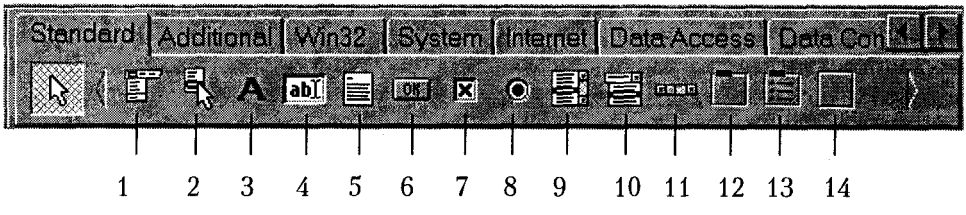


Рис. 16.8. Компоненты вкладки Standard

На вкладке ***Standard*** (Стандартная) содержатся следующие компоненты:
1. ***TMainMenu*** создает для окна панель с командами главного меню.
2. ***TPopUpMenu*** создает всплывающее меню для компоненты приложения.
3. ***TLabel*** помещает на объект текст, который нельзя редактировать.
4. ***TEdit*** помещает на объект область редактируемого ввода в виде одиночной строки.
5. ***TMemo*** размещает область редактируемого ввода из множества строк.

6. ***TButton*** размещает кнопку с названием.

7. ***TCheckBox*** создает управляющий элемент с двумя возможными состояниями, причем состояние одного элемента не влияет на состояние других подобных элементов объекта.

8. ***TRadioButton*** создает управляющий элемент с двумя возможными состояниями. Из группы подобных элементов объекта только один может находиться в установленном состоянии. Установка другого элемента из группы автоматически приводит к выключению предыдущего элемента.

9. ***TListBox*** создает область в виде списка текстовых строк для ввода.

10. ***TComboBox*** создает комбинацию из строки редактируемого ввода и ниспадающего списка текстовых строк для ввода.

11. ***TScrollBar*** создает линейку прокрутки для окна, списка и т. п.

12. ***TGroupBox*** создает контейнер, объединяющий логически связанные объекты.

13. ***TRadioGroup*** создает контейнер для объединения управляющих элементов ***TRadioButton***.

14. ***TPanel*** создает панель инструментов.

Как видно из рис. 16.8, групп компонентов достаточно много, а самих компонентов более сотни. Для получения навыков работы в C++ Builder знаний о стандартных компонентах будет достаточно. Более подробную информацию об остальных компонентах можно получить из справочной системы или из документации.

Работа с ***Редактором форм*** труда не представляет и заключается в визуальном размещении на форме с помощью мыши требуемых объектов. Настройка свойств и реакции на события осуществляется через Инспектор объектов (рис. 16.1).

Редактор кода представляет собой текстовый редактор, позволяющий программисту создавать фрагменты кода программы, чтобы сделать созданные объекты приложения удовлетворяющими требованиям решаемой задачи.

16.4. Создание приложений

Общую схему разработки приложений (прикладных программ) мы уже рассматривали в разделе 1. Здесь мы опишем особенности ее реализации в системе C++ Builder. Рассмотрим технику создания консольных приложений, выполняемых под управлением MS DOS, и Windows-приложений.

Общая схема создания приложений

Рассмотрим процесс создания простейших приложений с помощью C++ Builder. Информацию о том, каким образом строить приложение, система программирования C++ Builder берет из файла проекта. Процедура построения конечного продукта для работы под Windows значительно сложнее и отличается от привычной простейшей схемы, когда один исходный файл с текстом программы обрабатывается компилятором, затем, если не было ошибок, результат в виде объектного модуля поступает компоновщику, после чего получается готовый к выполнению файл с расширением *exe*.

Схема получения исполняемого файла для работы под Windows изображена на рис. 16.9.

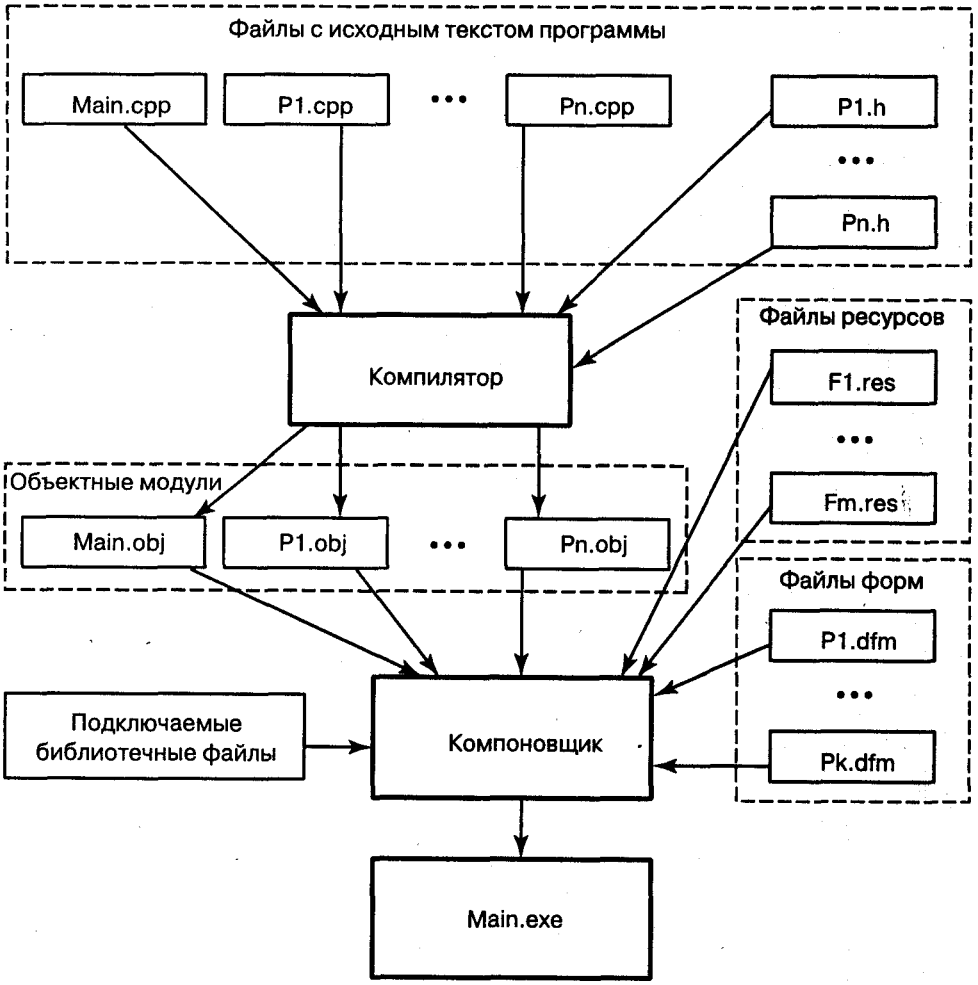


Рис. 16.9. Схема генерации исполняемого файла программы

Как видно из рисунка, кроме файлов с исходным текстом программы, в процессе разработки создаются файлы форм, файлы ресурсов и некоторые другие файлы. В файлах ресурсов обычно хранятся различного типа графические изображения, используемые в программе. Файлы форм в двоичном виде содержат описание формы и всех ее компонентов.

Создание консольных приложений

Приведенные в книге примеры программ не являются Windows-приложениями и для их работы требуется текстовый режим для ввода-вывода информации. C++ Builder предоставляет возможность создания таких программ, которые получили название ***консольные приложения win32***. По результатам работы они не отличаются от DOS-программ. Однако такие программы могут выполняться под управлением операционных систем Windows 95/98/NT.

Приведенные в книге примеры во избежание ошибок следует отлаживать как консольные приложения. Создание консольных программ выполняется в описанной ниже последовательности.

1. После запуска C++ Builder необходимо закрыть автоматически созданный проект, выдав команду меню ***File/Close All*** (*Файл/Закрыть все*).

2. Затем создается проект для консольного приложения, для этого задается команда меню ***File/New...*** (*Файл/Новый...*). На экране появляется окно для выбора нового создаваемого элемента (рис. 16.10).

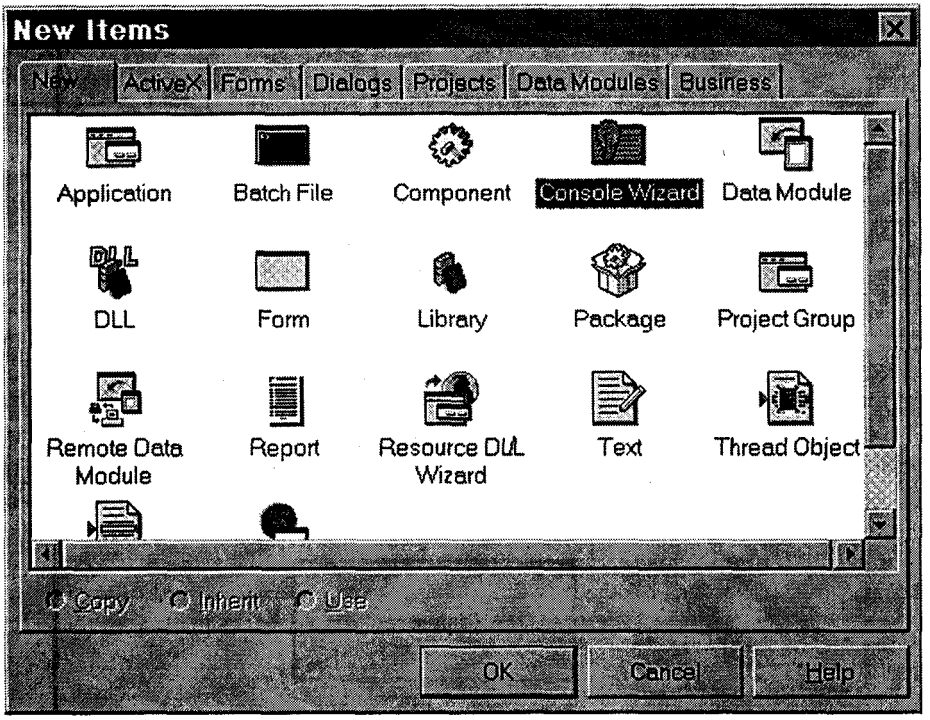


Рис. 16.10. Окно выбора типа элемента

В этом окне выбираем элемент с надписью ***Console Wizard*** (*Мастер консоли*) и нажимаем кнопку OK.

3. Далее запускается *Мастер* по генерации проекта для *консольных приложений*, после чего появляется окно, показанное на рис. 16.11.

4. Не внося изменений, следует нажать кнопку ***Finish*** (*Конец*). После чего появляется окно редактора кода, в котором находится шаблон с текстом программы. Далее можно набирать программу, используя этот шаблон, либо удалить текст шаблона и набирать программу заново (рис. 16.12). При этом для проекта создаются минимум два файла — *исходный файл проекта* с расширением *cpp* (его содержимое отображается в редакторе кода) и *информационный файл проекта*, содержащий установленные программистом параметры компилятора, имена исходных и библиотечных файлов проекта.

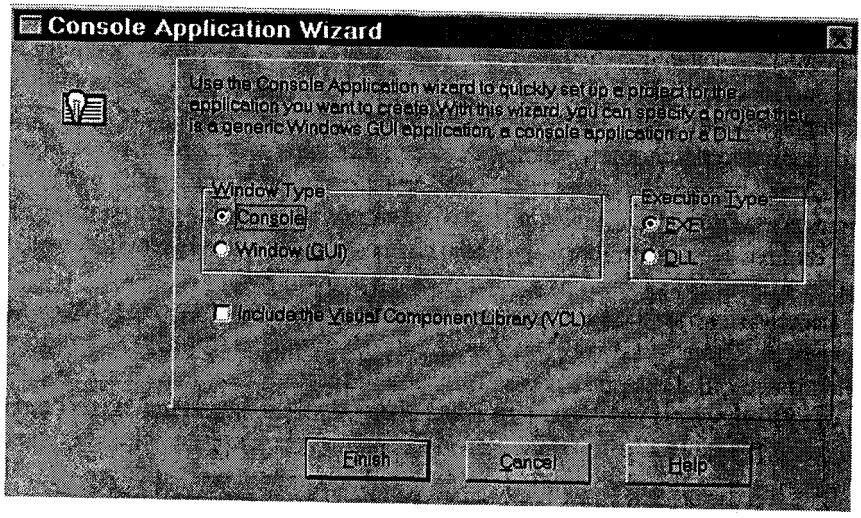


Рис. 16.11. Окно *Мастера консольных приложений*

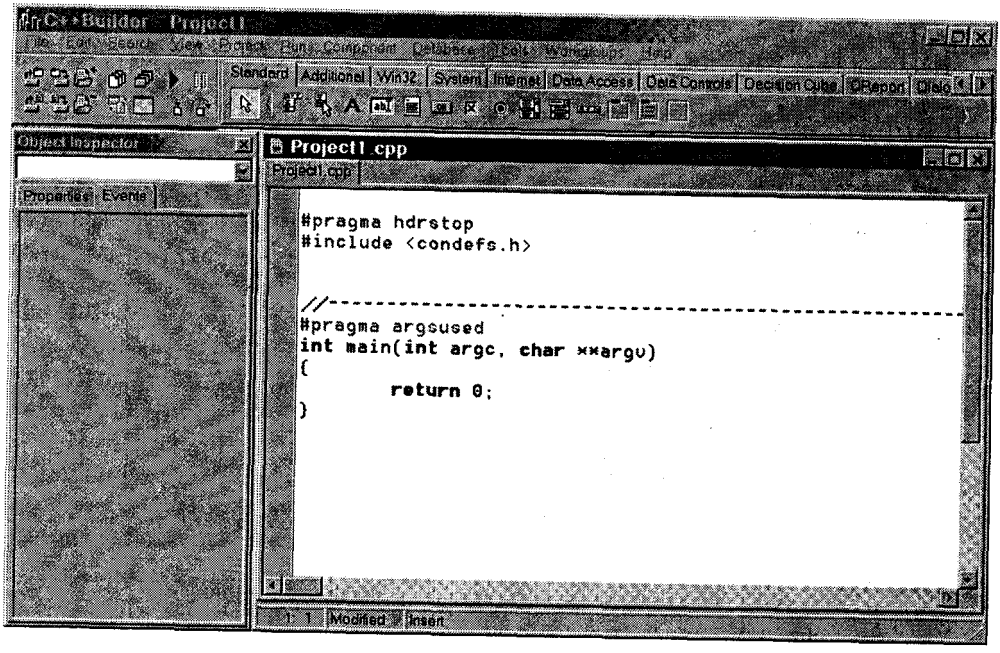


Рис. 16.12. Окно *Редактора кода* с шаблоном для консольного приложения

При наборе программ, являющихся консольными приложениями, следует учитывать, что консольные приложения и Windows-приложения используют разную ***кодировку расширенного набора*** символов (русские буквы и др.). Редактор кода, входящий в C++ Builder, использует кодировку Windows. Поэтому набранные в нем сооб-

щения на русском языке для вывода на экран будут при работе консольной программы отображаться в виде непонятного набора символов. Данную проблему можно решить двумя способами. Первый — это вместо русскоязычных сообщений использовать англоязычные. Но это не всегда приемлемо.

Второй способ состоит в использовании для набора исходного текста программы любого редактора, являющегося DOS-программой (встроенный редактор Norton Commander, DOS и т. п.). После этого с текстом программы можно работать в среде C++ Builder. При этом, чтобы русские символы в редакторе кода отображались правильно, нужно для редактора кода поменять тип шрифта.

Для этого следует задать команду меню ***Tools/Environment Options*** (*Сервис/Параметры окружения*); в результате открывается окно ***Environment Options*** (*Параметры окружения*), позволяющее изменять параметры среды программирования. В нем нужно выбрать вкладку ***Display*** и сменить параметр ***Editor Font*** (*Шрифт редактора*) на тип ***Terminal*** (*Терминальный*) (рис. 16.13). Тем самым проблема несовместимости кодировок будет решена.

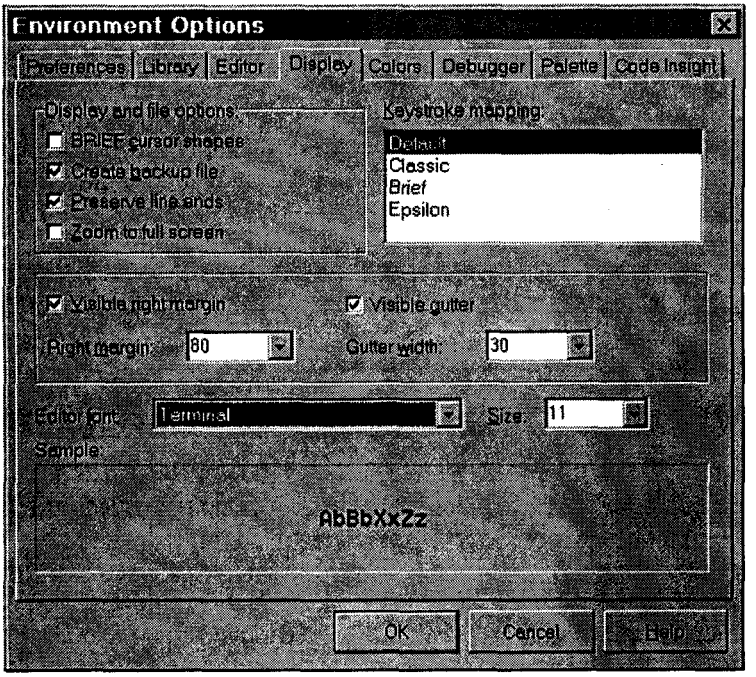


Рис. 16.13. Окно настройки параметров среды программирования

5. Чтобы выполнить программу, необходимо нажать кнопку «запуск программы на решение» на панели инструментов (рис. 16.7). Если ошибок нет, будет сгенерирован *exe*-файл и автоматически запущен на выполнение.

После этого готовое приложение можно запускать в системе Windows. Оно будет запускаться в окне MS DOS и по своей работе выглядеть как DOS-приложение. По приведенному алгоритму в C++ Builder следует запускать на решение программы, приведенные в этой книге.

Создание Windows-приложений

C++ Builder создает проект для Windows-приложения по умолчанию при входе в систему. При этом создаются минимум шесть следующих файлов.

1. ***Исходный файл проекта*** с расширением *cpp*. В нем содержится функция для запуска Windows-приложения *WinMain()* и другие команды, использующиеся при старте.

2. ***Исходный файл главной формы*** с расширением *cpp*. Содержит объявление объекта главной формы и определение методов (функций-членов класса), участвующих в управлении формой и ее компонентами.

3. ***Заголовочный файл главной формы*** с расширением *h*. Содержит определение класса главной формы.

4. ***Файл ресурсов приложения***. Представляет собой двоичный файл, который описывает графический значок приложения.

5. ***Файл ресурсов главной формы***. Также двоичный файл, который содержит описание графических объектов главной формы.

6. ***Информационный файл проекта***. Представляет собой текстовый файл, содержащий данные об именах исходных файлов и форм проекта, параметрах компилятора, именах подключаемых библиотечных файлов.

Пример 1. Создание Windows-приложения.

Пусть необходимо создать окно, в котором при запуске отображается сообщение-приветствие «Здравствуйте! Я удачно запущена!». При этом в нижней части окна будут размещены три кнопки. При нажатии первой кнопки сообщение-приветствие должно заменяться сообщением «Привет от первой кнопки». При нажатии второй кнопки текущее сообщение должно быть заменено на «Привет от второй кнопки!» и третья кнопка, если она присутствует в окне, должна исчезнуть, а если ее нет, то она должна появиться. При нажатии на третью кнопку программа должна завершить свою работу. Надписи на кнопках следующие: «Кнопка 1», «Кнопка 2», «Выход».

Последовательность действий по *созданию приложения* для данного примера такова:

1. После входа в систему C++ Builder будет по умолчанию создан файл проекта под Windows-приложение. В редакторе форм можно визуально просмотреть внешний вид окна нашего приложения. Сейчас в нем еще ничего нет.

2. Поместим на форму три кнопки. Для этого выберем вкладку ***Standard*** на палитре компонентов. Щелкнем мышью на компоненте ***TButton*** (кнопка), а затем на форме в месте размещения кнопки. Заметим, что сами кнопки легко перетаскиваются по форме с помощью мыши, при этом мышью можно менять их размер.

3. Выберем аналогичным образом на палитре компонентов вкладки ***Standard*** объект ***TLabel*** и разместим его в требуемом месте на форме. Результаты наших действий приведены на рис. 16.14.

4. Заменим названия кнопок на требуемые, а также заменим надпись *Form1* в верхней части окна на «*Суперпрограмма*». Для этого будем использовать свойства *Caption*, общее для всех компонентов. Щелкнем мышью на форме, в верхней части инспектора объектов будет отображено имя объекта, для которого можно устанавливать свойства. Для формы это — *Form1:Tform1*. Затем щелкнем мышью на свойстве *Caption* и заменим его значение «***Form1***» на «*Суперпрограм-*

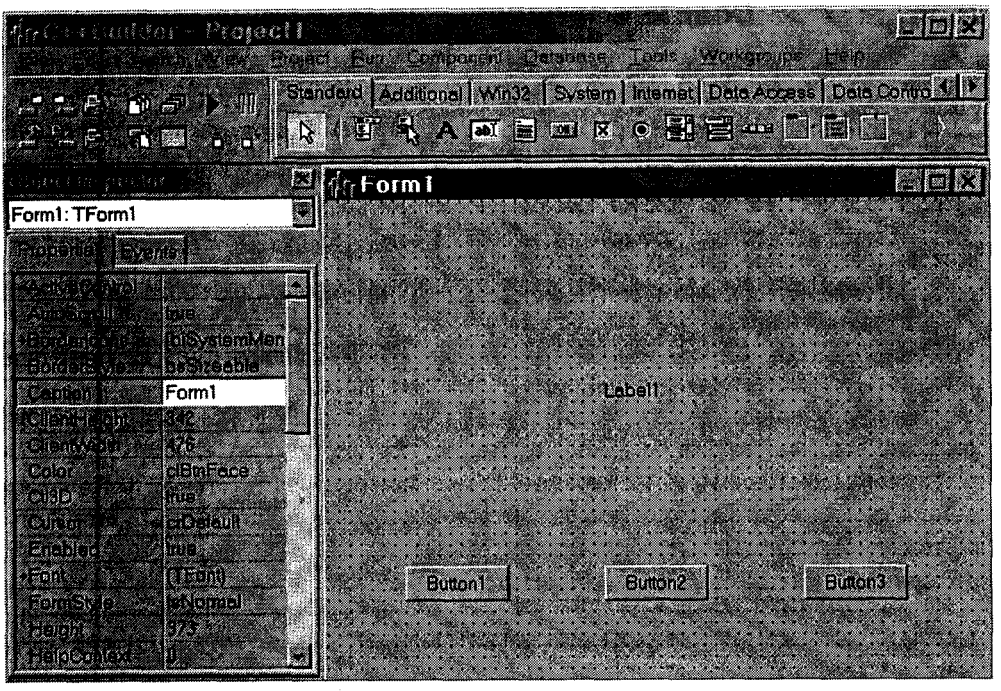


Рис. 16.14. Вид компонентов на форме

ма». Надпись в заголовке окна будет изменена. Аналогично изменим свойство *Caption* для кнопок и строки для сообщения на заданные в условии задачи. Результат показан на рис. 16.15.

5. Чтобы запрограммировать действия, выполняемые в программе при нажатии на кнопку, дважды щелкнем в редакторе форм на кнопке, для которой определяем действия. После этого откроется окно редактора кода с текстом исходного файла формы, в котором находится автоматически созданное определение функции, выполняемой при нажатии кнопки. Введем текст программы, заменяющий сообщение на текст «Привет от первой кнопки!». Он будет следующим:

```
Label1->Caption="Привет от первой кнопки!";
```

В этом случае для управления объектом ***Label1*** мы использовали его свойство *Caption* (рис. 16.16).

6. Нажав кнопку *«Переключиться на редактор форм или редактор кода»* панели инструментов, вернемся в редактор форм. Для второй кнопки аналогичным образом введем следующий программный код:

```
Label1->SetTextBuf("Привет от второй кнопки");
  if (Button3->Visible==true) Button3->Hide();
  else Button3->Show();
```

Заметим, что в этом случае для управления объектом ***TLabel*** использован метод ***SetTextBuf()***, позволяющий заменять текст сообщения. Результат достигает-

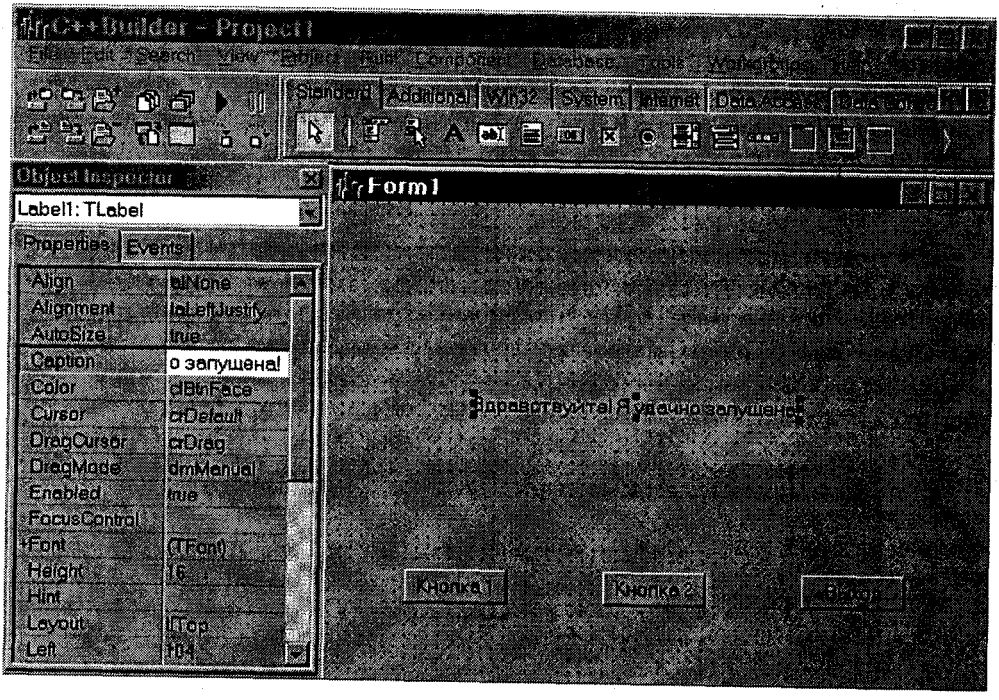


Рис. 16.15. Изменение свойств компонентов через Инспектор объектов

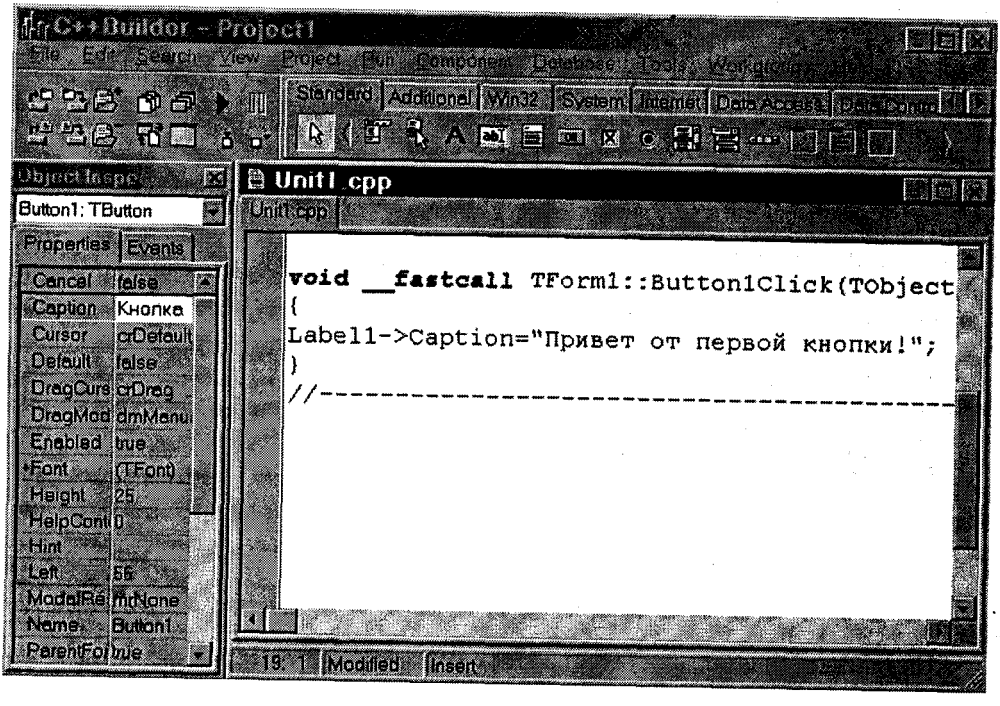


Рис. 16.16. Программное изменение свойства объекта

ся аналогичный, как и для первой кнопки, разница состоит в подходах к управлению объектом. Чтобы управлять видимостью третьей кнопки, анализируется ее свойство ***Visible***, и в зависимости от его значения вызывается метод ***Hide()*** (*Скрыть объект*) или ***Show()*** (*Показать объект*). Для третьей кнопки аналогичным образом введем код:

```
Close();
```

7. На этом составление программы закончено. Осталось убедиться в ее работоспособности. На панели инструментов нажмем кнопку «*Запустить программу на решение*». После этого программа будет откомпилирована и, если нет ошибок, сгенерированный *exe*-файл будет автоматически запущен на выполнение. На экране появятся результаты работы программы (рис. 16.17). Далее, нажимая кнопки формы, остается убедиться в правильности работы программы.

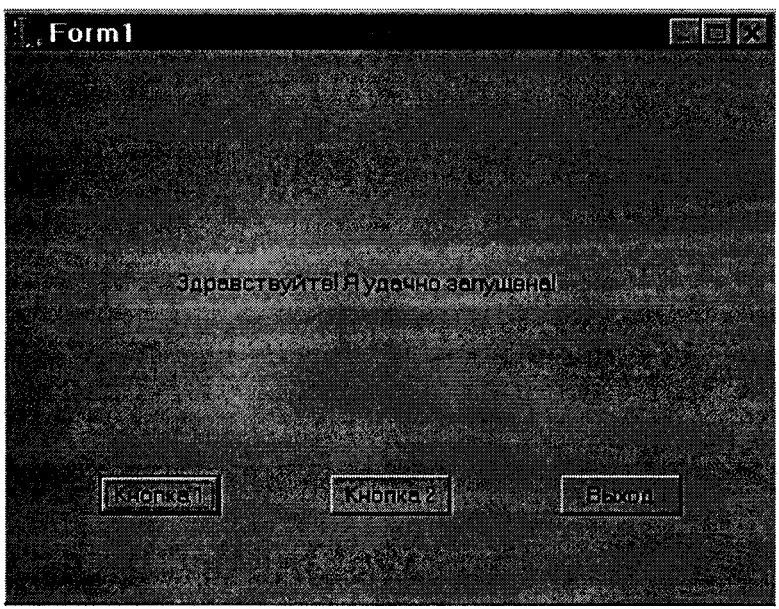


Рис. 16.17. Вид формы при выполнении программы

Таков алгоритм разработки программ в системе C++ Builder. Отметим, что нами описана малая часть возможностей, которыми обладает эта система программирования. Кроме разработки обычных Windows-приложений, в C++ Builder можно разрабатывать приложения по работе с базами данных, по работе в сети Internet и другие. Для изучения данной системы требуется времени не меньше, чем для изучения самого языка C++.

Контрольные вопросы и задания

1.Назовите отличительные особенности системы C++ Builder.

2.Охарактеризуйте интегрированную среду разработки (IDE) и назовите ее основные компоненты.

3.Каково назначение библиотеки классов в системах разработки Windows-приложений?

4.Дайте краткую характеристику и укажите особенности наиболее распространенных библиотек классов.

5.Что понимается под термином «технология визуального программирования»?

6.Поясните понятия «свойства», «методы», «события» применительно к программированию в среде C++ Builder.

7.Каким образом связаны между собой классы библиотеки VCL?

8.Дайте краткую характеристику и назначение основных базовых классов библиотеки VCL.

9.Какие действия может выполнять программист, используя команды главного меню?

10. Поясните смысл понятия «проект» в системе программирования C++ Builder.

11. Каково предназначение проекта?

12. Для чего предназначена палитра компонентов?

13. Каково предназначение Инспектора объектов?

14. Для чего используется Редактор форм в системе программирования C++ Builder?

15. Какова схема генерации исполняемого кода Windows-приложения в C++ Builder?

16. Какой тип программ понимается под термином «консольное приложение»?

17. В чем отличительные особенности шаблона проекта для разработки Windows-программ?

18. Создайте консольное приложение в системе программирования C++ Builder. Составьте и отладьте программу нахождения максимума из трех чисел.

19. Создайте консольное приложение в системе программирования C++ Builder, состоящее из одного окна. Установите ширину окна в 600 точек, а высоту — в 480. Измените надпись в заголовке окна на *«Программа 1»*. Измените имя объекта окна *Form1* на *Wind1*.

20. Доработайте приложение, созданное в предыдущем задании следующим образом. Запрограммируйте реакцию на событие активизации окна. При каждой активизации окна его размеры по ширине и высоте должны уменьшаться в два раза.

21. Доработайте созданное в предыдущем задании приложение, разместив в окне две кнопки. При нажатии на первую должно появляться сообщение приветствие «Привет от Программы 1», при нажатии на вторую кнопку программа должна завершать свою работу. При щелчке «мышью» на сообщении оно должно исчезать.

ПРИЛОЖЕНИЕ

Ключевые слова

В стандарте C++ определено 65 ключевых слов, которые приведены в табл. П.1. Все ключевые слова записываются строчными буквами; ниже приведена краткая справочная информация по их назначению и использованию.

Таблица П.1

Ключевые слова C++

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

and

Слово ***and*** эквиваленно двум символам && и используется для обозначения бинарной операции «логическое И». В табл. П.2 и П.3 приведено определение всех логических операций языка C++.

and_eq

Слово ***and_eq*** эквиваленно двум символам &= и используется для обозначения бинарной операции «присваивание с поразрядным логическим И».

asm

Заголовок ***asm*** используется для встраивания команд на языке ассемблера в программу на C++.

auto

Спецификатор ***auto*** используется для объявления *локальных переменных*. По умолчанию переменные являются локальными, поэтому *auto* используется редко.

bool

Спецификатор ***bool*** указывает на булевский (логический) тип. Переменные типа *bool* могут принимать два значения: *true* (истина) и *false* (ложь). В выражениях значение *false* соответствует 0, а *true* соответствует 1.

break

Оператор ***break*** используется для немедленного выхода из операторов циклов *do*, *for*, *while* и оператора выбора *switch*. При вложении циклов оператор ***break*** позволяет выйти из тела одного оператора цикла.

case

Заголовок ***case*** используется в операторе выбора *switch* для определения одного из выбираемых вариантов действий.

catch

Оператор ***catch*** является частью механизма работы с *исключениями*. Оператор *catch* предназначен для обработки исключений, которые выбрасывает оператор *throw*.

char

Спецификатор ***char*** указывает на *символьный* (8-битный) тип, *sizeof (char) == 1*.

class

Заголовок ***class*** начинает описание (опередение) производного структурного типа данных «класс объектов», который может наследовать элементы-данные и методы родительских классов и иметь собственные элементы данные и методы. Синтаксис описания типа *class* в общей форме имеет вид:

```
class class_name : class_parent_name_list {
  // private  member: variables and functions
  ...
```

```
    protected:              // protected  member: variables and functions
    ...
    public:                 // public member: variables and functions
    ...
} object_list;
```

В объявлении класса список переменных *object_list* типа *class_name* является необязательным.

const

Модификатор ***const*** сообщает компилятору, что значение переменной, которая за ним следует, не должно изменяться. Такие переменные также называют *константными переменными* или *символическими константами.*

const_cast

Бинарная операция ***const_cast*** используется для удаления или добавления модификаторов *const* и *volatile* во время исполнения программы. Синтаксис операции имеет вид

```
const_cast <T> (obj)
```

где: *T* — тип, к которому приводится тип переменной *obj.*

Типы операндов должны отличатся только модификаторами *const* и *volatile.* Результат операции имеет тип *T.*

continue

Оператор ***continue*** используется для обхода оставшихся операторов в теле цикла и перехода к очередной итерации (к проверке условия выполнения тела цикла).

default

Заголовок ***default*** используется в операторе *switch* для указания операторов, выполняющихся по умолчанию.

delete

Унарная операция ***delete*** освобождает память, выделенную с помощью операции *new.* При этом для динамических массивов нужно указывать пару квадратных скобок:

```
delete [ ] array_name,
```

где *array_name* — имя массива или указатель.

do

Заголовок ***do*** является началом одного из трех операторов цикла, имеющихся в языке. Синтаксис *оператора цикла с постусловием do_while* выглядит следующим образом:

```
do {
   statements;
```

```
}while (condition);
```

где: *statements* — операторы, выполняемые в теле цикла;
condition — выражение, определяющее условие повторения тела цикла.

Если в теле цикла имеется один оператор, то фигурные скобки можно опустить. Цикл выполняется до тех пор, пока значение *condition* есть *true*.

Оператор *do — while* обеспечивает выполнение не менее одной итерации.

double

Спецификатор ***double*** задает тип вещественных чисел двойной точности.

dynamic_cast

Бинарная операция ***dynamic_cast*** выполняет преобразование типов полиморфных объектов во время выполнения. Синтаксис операции имеет вид

```
dynamic_cast <T> (pObj),
```

где: *T* — тип ссылки или тип указателя, в том числе и *void* *;
pObj — ссылка или указатель на объект соответственно.

Результат операции *dynamic_cast* имеет тип *T*, если преобразование успешно. В противном случае результат операции равен нулю (для указателей) или выбрасывается исключение *bad_cast* (для ссылок). В контекстах, где не во всех случаях возможно корректное использование полиморфных объектов, операция *dynamic_cast* позволяет ответить на вопрос: «Объект, на который указывает (ссылается) *pObj*, имеет тип *T*?» Для неполиморфных типов следует использовать операцию *static_cast*.

else

Заголовок ***else*** обозначает начало второй необязательной части *оператора ветвления if*.

enum

Спецификатор типа *enum* используется для описания *перечислимых типов* данных, которые являются производными от целочисленных типов. Тип *enum* широко используется для описания символических целых констант.

explicit

Модификатор ***explicit*** используется для описания конструкторов, предусматривающих явное создание объекта с инициализацией вида

```
obj_class obj1(3);
```

и запрещают следующее конструирование объекта *obj1* типа *obj_class*.

```
obj_class obj1 = 3;
```

Для этого в описании конструктора *obj_class* следует указать

```
explicit obj_class(int state) { ... }
```

export

Модификатор ***export*** сообщает компилятору, что описание функции или шаблона предполагается использовать в других единицах трансляции (исходных файлах).

extern

Модификатор ***extern*** сообщает компилятору о том, что переменная объявляется в другом файле или другом месте программы. Модификатор обычно используется при раздельной компиляции исходных файлов, в дальнейшем объединяемых при редактировании связей. Как правило, модификатором *extern* описываются глобальные переменные в заголовочных файлах.

false

Логическое значение ***false*** (ложь) представляет собой одно из двух значений, которые могут принимать переменные типа *bool*.

float

Спецификатор ***float*** задает тип *вещественных чисел одинарной точности*, представленных в форме с плавающей точкой.

for

Оператор *цикла с параметром* ***for*** позволяет в рамках одного оператора осуществить не только проверку условия выполнения тела цикла, но и инициализацию переменных, и изменение переменных. Общий синтаксис цикла *for* выглядит следующим образом:

```
for (initialization; condition; modification)
  {
   statements;
  }
```

где: *initialization* — список инициализирующих выражений или объявлений переменных с инициализацией;
condition — выражение, определяющее условие повторного выполнения тела цикла;
modification — список выражений, изменяющих значение переменных;
statements — операторы, выполняемые в теле цикла.

Если в теле цикла имеется один оператор, то фигурные скобки не обязательны. Цикл будет выполнятся до тех пор, пока значение *condition* есть *true*. По умолчанию (выражение *condition* опущено) условие выполнения тела цикла равно *true*.

friend

Спецификатор доступа ***friend*** используется для объявления функций или классов, *дружественных* описываемому классу. Дружественные функции или классы не являются членами класса, но имеют доступ к *закрытым* и *защищенным* элементам

класса. Дружественные функции не могут быть вызваны по указателю *this* и не применяются к объектам класса, в котором они объявлены со спецификатором *friend*, с помощью операций «точка» (.) и «стрелка» (–>).

goto

Оператор безусловного перехода ***goto*** вызывает переход в процессе выполнения программы к метке, имя которой указано после ключевого слова *goto*. Использование этого оператора часто приводит к получению программ с плохо понимаемой структурой.

if

Заголовок ***if*** обозначает начало *условного оператора*, синтаксис которого имеет вид:

```
if (condition) {
   bloc_1_statements;
}
else {
   bloc_2_statements;
}
```

где: *condition* — выражение, определяющее какой из двух блоков будет выполнен;
bloc_1_condition, bloc_2_condition — операторы первого и второго блока соответственно.

При использовании одиночных операторов в блоках фигурные скобки можно опустить. Вторая часть условного оператора, начинающаяся со слова *else*, также не обязательна. Выражение *condition* может быть любого типа, кроме *void*, и его значение приводится к типу *bool*. Если значение *condition* есть *true*, то выполняются операторы первого блока, в противном случае — выполняются операторы второго блока (если он имеется).

inline

Модификатор ***inline*** используется для описания так называемых *встраиваемых* функций. Это уменьшает число операций процессора, выполняемых при вызове встраиваемых функций, и, следовательно, позволяет получать более быстродействующие программы. Но при использовании встраиваемых функций увеличивается размер программы. Поэтому они используются, в первую очередь, для описания коротких последовательностей операторов.

int

Спецификатор ***int*** определяет тип *целых чисел*.

long

Спецификатор ***long*** определяет тип *длинных целых*, если он используется со спецификатором *int* или без него. Если *long* используется совместно с *double*, то он определяет тип вещественных чисел двойной точности.

mutable

Модификатор ***mutable*** используется для обеспечения возможности изменения переменной при любом распределении памяти для нее. Другими словами, *mutable* означает: «ни при каких обстоятельствах не является *const*».

namespace

Спецификатор ***namespace*** определяет *пространство имен* функций, классов и переменных, находящихся в отдельной *области видимости*. Синтаксис описания пространства имен имеет вид:

```
namespace name {
// объявления и определения имен
...
}
```

где *name* — идентификатор пространства имен.

Поскольку пространство имен определяет область видимости, то для доступа к членам этого пространства необходимо использовать операцию разрешения видимости.

new

Операция ***new*** осуществляет выделение памяти. Часто используется следующий оператор:

```
ptype =  new type;
```

Здесь *type* обозначает имя типа данных, для которого производится выделение памяти, а *ptype* должен быть указателем на переменную, тип которой совместим с типом *type*. Операция *new* выделяет непрерывный участок памяти, требуемый для хранения объекта указанного типа, и возвращает адрес (указатель) начала этого участка. Если память для хранения объекта выделить невозможно (например, из-за нехватки), то будет возвращено нулевое значение.

Для выделения памяти под динамический массив используется следующий оператор:

```
ptype = new type[size];
```

где ***size*** определяет количество элементов типа ***type***, имеющихся в массиве.

operator

Спецификатор ***operator*** используется для описания *функций-операций*, переопределяющих встроенные операции языка.

private

Модификатор доступа *private* используется для объявления ***собственных (закрытых)*** членов класса. Доступ к таким членам класса могут получить только другие члены этого класса. По умолчанию члены класса являются ***собственными.***

protected

Модификатор доступа ***protected*** используется для объявления *защищенных* членов класса. Доступ к таким членам класса могут получить другие члены этого класса или члены производного класса.

public

Модификатор доступа ***public*** используется для объявления общедоступных членов класса. Доступ к таким членам класса можно получить из любой другой части программы с помощью операций «точка»(.) и «стрелка» (->).

register

Модификатор ***register*** предназначен для объявления локальных переменных, которые должны размещаться в наиболее быстродействующей памяти. Для переменных *скалярных типов* это обычно означает использование одного из регистров процессора. Для других типов объектов это может означать кэширование памяти. Если быстрой памяти недостаточно, то для хранения значений переменной будет выделяться обычная память.

reinterpret_cast

Бинарная операция ***reinterpret_cast*** изменяет при выполнении программы тип второго операнда на тип, задаваемый первым операндом, без проверки их совместимости. Синтаксис операции имеет вид

```
reinterpret_cast <T> (v)
```

где T — тип, к которому приводится выражение v.

Например, *reinterpret_cast* можно использовать для преобразования указателя в целое. Таким образом, *reinterpret_cast* может использоваться для преобразования несовместимых типов.

return

Оператор ***return*** выполняет возврат из тела функции и обычно используется для передачи скалярных значений вызывающей функции. При обработке оператора *return* выполнение функции завершается, а все ее последующие операторы не выполняются.

short

Спецификатор ***short*** определяет тип *коротких целых*.

signed

Модификатор типа ***signed*** наиболее широко используется для объявления переменных *знакового символьного* типа данных. Его можно использовать для любого целого типа, но такое объявление будет избыточным.

sizeof

Унарная операция ***sizeof*** выполняется при компиляции и возвращает выраженную в байтах длину переменной, которой она предшествует. Если операнд есть имя типа, то его заключают в круглые скобки. Операция ***sizeof*** позволяет написать переносимый код C++ в случаях, когда переносимость кода зависит от размерности встроенных типов данных C++.

static

Модификатор типа ***static*** сообщает компилятору о необходимости статического выделения памяти для переменной. Объявленная таким образом переменная имеет *глобальное время жизни*. Значение переменной типа ***static*** не изменяется между вызовами функции, в которой она определена.

Модификатор *static* может применяться и к членам класса. При созданнии объектов класса, в котором имееся переменная типа *static*, будет создана только одна такая переменная. И эта переменная будет общей (разделяемой) переменной для всех объектов данного класса.

static_cast

Операция ***static_cast*** выполняет преобразование *неполиморфных типов* при выполнении программы. Ее можно использовать для любых стандартных преобразований.

struct

Заголовок ***struct*** начинает описание производного структурного типа, который называется *«структура»* и может содержать элементы разного типа. Ниже приведена простейшая форма описания структуры:

```
struct struct_name
  {
    type1  member1;
    type2  member2;
    ...
    typeN  memberN;
  }
object_list;
```

где: *struct_name* — имя типа;
typeN memberN; — объявление *N*-го элемента *N*-го типа;
obj_list — необязательный список переменных типа *struct_name*.

Доступ к отдельным членам структуры осуществляется с помощью операций «.» и «->».

По умолчанию, все члены структуры имеют уровень доступа *public*. Однако, для ограничения доступа при описании структуры можно использовать модификаторы *private* и *protected*. Тип *struct* является частным случаем типа *class*.

switch

Заголовок ***switch*** обозначает начало *оператора выбора* (условного перехода с множественным ветвлением). Общая форма оператора *switch* имеет вид:

```
switch(expression) {
  case constant1:  statement_1list;
    break;
  ...
  case constantN:  statement_Nlist;
    break;
 default:
  statement_default_list;
}
```

где: *expression* — выражение, приводящееся к целому;

constantN — целое значение, которое задает выполнение последовательности операторов *statement_Nlist.*

Оператор *switch* работает по принципу сравнения значения *expression* с константами *constant1, ..., constantN.* Когда значение выражения равно константе, то следующая за константой последовательность операторов выполняется до тех пор, пока не встретится оператор *break*, по которому завершается выполнение оператора *switch.* Если оператор *break* пропущен, то выполняется следующий оператор *case.* Выполнение продолжается до тех пор, пока встретится утверждение *break* или до окончания последнего оператора *case.*

template

Заголовок ***template*** используется для описания *шаблонов,* или параметризованных функций и классов. При создании шаблона функции или класса типы данных, над которыми они оперируют, задаются одним или несколькими параметрами, которые заменяются компилятором фактическими типами данных при создании конкретного экземпляра шаблона. Общая форма *шаблона функции* имеет вид:

```
template <class T> return_type func_name (parameter_list)
  {
    statements;
  }
```

Здесь *T* представляет собой имя формального параметра для типа данных, используемых функцией *func_name.* Имя *T* может использоваться в пределах определения функции для объявления переменных и в других целях. Однако оно является параметром, и компилятор автоматически заменит его нужным типом данных при создании конкретного экземпляра функции. В описании шаблона можно определить несколько параметризованных типов данных, указывая их в списке параметров через запятую.

Общая форма *шаблона класса* имеет вид:

```
template <class T> class class_name
  {
```

```
    // описания и объявления элементов класса
    ...
}
```

Здесь *T* — имя формального параметра-типа, которое компилятор при построении конкретного экземпляра класса заменит указанным типом данных. Можно определить несколько параметризованных типов данных, используя список с запятой в качестве разделителя. Функции-члены параметризованного класса также автоматически становятся параметризованными.

this

Ключевое слово ***this*** является указателем на объект, вызвавший функцию-член. Поэтому выражение ****this*** в теле функции означает объект, для которого вызвана эта функция-член.

throw

Оператор ***throw*** является частью механизма работы с *исключениями* и используется для выбрасывания исключений. Обработка исключений ведется с помощью трех операторов: *try*, *throw* и *catch*. Операторы программы, при выполнении которых могут возникать исключительные ситуации и выбрасываться исключения, помещаются в блок оператора *try*. Если возникает исключительная ситуация, то с помощью *throw* выбрасывается исключение и управление передается первому оператору *catch*, следующему за блоком *try* и способному обрабатывать тип выброшенного исключения. Операторы *trow* должны находится в блоке *try* или в функциях, вызываемых в блоке *try*.

Синтаксис операторов *try*, *trow* и *catch* имеет вид

```
try{
  statements;
  trow exception_expr;
  statements;
  ...
}
catch (type1 exception) {
  statements;
}
...
catch (typeN exception) {
  statements;
}
catch (...) {
  statements;
}
```

Блок *try* может содержать небольшие последовательности операторов и большие части программы, включая функцию *main()*. Исключение, выбрасываемое операто-

ром *trow*, задается выражением *exception_expr*. При выбрасывании исключения оно перехватывается по типу *exception_expr* соответствующим оператором *catch*, который и обрабатывает его. После оператора *try* может находится несколько операторов *catch*, каждый из которых обрабатывает исключения определенного типа. При перехвате исключения параметр *exception* соответствующего оператора *catch* примет значение *exception_expr*. Оператор *catch(...)* перехватывает исключения всех типов. После обработки исключения одним из операторов *catch* управление передается за последний оператор *catch*. Если в блоке *try* не выброшено ни одного исключения, то не выполнится ни один из операторов *catch*. Оператор *throw* должен находится в блоке *try* или в любой функции, вызываемой из этого блока. Если исключение выброшено, а для его типа отсутствует оператор *catch*, то выполнение программы прекращается.

true

Логическое значение ***true*** (истина) представляет собой одно из двух значений, которые могут принимать переменные типа *bool*.

try

Оператор ***try*** является частью механизма работы с *исключениями*. Подробнее см. пояснения к ключевому слову *throw*.

typedef

Оператор ***typedef*** позволяет описывать новые имена для существующих типов данных.

typeid

Унарная операция ***typeid*** возвращает информацию о типе объекта при выполнении программы. Синтаксис операции имеет вид:

```
typeid(object)
```

Результатом операции *typeid* является ссылка на объект типа *type_info*, описывающий тип объекта, заданного параметром *object*. Для использования *typeid* необходимо включить в программу стандартный заголовочный файл *typeinfo.h*. Если в качестве операнда указана ссылка или указатель на *полиморфный тип*, то результатом является информация о фактическом типе объекта. Если операндом является выражение или неполиморфный объект, то результатом является ссылка на объект типа *type_info*, содержащий статический тип выражения.

Класс *type_info* определяет следующие общедоступные члены:

```
bool operator==(const type_info &obj) const;
bool operator!=(const type_info &obj) const;
bool before(const type_info &obj) const;
const char *name() const;
```

Перегрузка операций == и != обеспечивает все необходимое для сравнения типов. Функция *before()* возвращает значение *true*, если вызывающий объект расположен

перед объектом, используемым в качестве параметра. Функция *name()* возвращает указатель на имя типа.

typename

Спецификатор ***typename*** указывает компилятору, что следующий за ним идентификатор в описании *шаблона* является именем типа. Спецификатор *typename* используется в случаях, когда возможна неоднозначная интерпретация идентификатора.

union

Заголовок ***union*** используется для описания производного структурного типа данных *объединение*, в котором все элементы-данные имеют единую область хранения и размер этой области памяти равен размеру самого большого элемента. Объект типа *union* может содержать один из элементов-данных. Тип *union* используется в случаях, когда имеются взаимоисключающие альтернативные данные разных типов об одном классе объектов. Синтаксис описания объединения имеет вид:

```
union union_name {
  type1  member1;
  ...
  typeN  memberN;
} object_list;
```

где: *union_name* — имя типа;
typeN memberN; — объявление *N*-го элемента *N*-го типа;
obj_list — список переменных типа *union_name*.

Тип *union* является частным случаем типа *struct*. Доступ к элементам объединения осуществляется с помощью операций «.» и «–>». По умолчанию, все элементы объединения имеют уровень доступа *public*.

unsigned

Модификатор типа ***unsigned*** используется для объявления переменных *беззнакового символьного* или *целого* типа данных.

using

С помощью оператора ***using*** можно упростить доступа к членам *пространства имен*. Оператор *using* имеет две формы:

```
using namespace name;
using name :: member;
```

В первом случае параметр *name* указывает идентификатор пространства имен, к которому требуется получить доступ. После этого оператора все члены пространства *name*, можно использовать без квалификатора *name* и операции «::». После оператора *using* во второй форме видимым делается только конкретный член *member* пространства *name*.

virtual

Спецификатор ***virtual*** используется для объявления *виртуальных функций*, реализующих *свойство полиморфизма* родственных классов. Виртуальные функции представляют собой функции-члены, которые объявляются в базовом классе, а затем переопределяются в производных классах.

void

Спецификатор базового типа ***void*** используется в основном для того, чтобы объявлять функции, которые не возвращают значения в место вызова, и *универсальные* указатели *void**.

volatile

Модификатор ***volatile*** сообщает компилятору о том, что значение объявляемой переменной может изменяться не описанным в программе образом.

wchar_t

Спецификатор ***wchar_t*** определяет *широкий* (*wide*) *символьный тип*, предназначенный для работы с символами из больших наборов, таких как UNICODE. Размер *wchar_t* в большинстве реализаций C++ составляет 2 байта.

while

Оператор цикла с предусловием ***while*** имеет следующий синтаксис:

```
while (condition)
  {
    statements;
  }
```

где: *condition* — выражение, определяющее условие выполнения тела цикла;
statements — операторы, выполняемые в теле цикла.

Таблица П.2

Бинарные логические операции

Значения операндов		Результат операции		
X	Y	ИЛИ (||, |)	И (&&, &)	ИСКЛЮЧАЮЩЕЕ ИЛИ (^)
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

false = 0; *true* = 1.

Если в теле цикла имеется один оператор, то фигурные скобки можно опустить. Цикл выполняется до тех пор, пока значение *condition* есть *true*. Если значение этого выражения изначально есть *false*, то операторы тела цикла не выполнятся ни разу.

Таблица П.3

Логическая операция «отрицание»

X	НЕ (!, ~)
0	1
1	0

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Л

М

Н

О

П

Р

С

Т

Литература

1. *Бабэ Б.* Просто и ясно о Borland C++: пер. с англ. — СПб.: Питер, 1997. — 464 с.
2. *Бад Т.* Объектно-ориентированное программирование в действии: пер. с англ. СПб.: Питер, 1997. — 464 с.
3. *Ирэ П.* Объектно-ориентированное программирование с использование C++: пер. с англ. К.: НИПФ ДиаСофтЛтд, 1995. — 480 с.
4. *Подбельский В.В.* Язык C++. — М.: Финансы и статистика, 1966. — 558 с.
5. Программирование. Учебник под ред. Свердлика А.Н., МО СССР, 1992. — 608 с.
6. *Сван Т.* Программирование для Windows в Borland C++: пер. с англ. — М.: БИНОМ. — 480 с.
7. *Шамис В.А.* Borland C++ Builder. Программирование на C++ без проблем. — М.: Нолидж, 1997. — 266 с.
8. *Шилдт Г.* Теория и практика C++: пер. с англ. — СПб.: BHV — Санкт-Петербург, 1996. — 416 с.
9. *Шилдт Г.* Самоучитель C++, 3-е издание: пер. с англ. — СПб.: BHV — Санкт-Петербург, 1998. — 688 с.

Содержание

Участие авторов в подготовке материалов: Аверкин В.П., к.т.н, — разделы 3, 16, подразделы 4.1.-4.3, 4.5-4.8; Бобровский А.И., к.т.н., с.н.с., — разделы 1, 2, 8, 12, 13, приложение; Веснич В.В., к.т.н., доцент, — разделы 1, 2, 5, 6, 9, 10, 15; Радушинский В.Ф. — раздел 14, пример 2 подраздела 11; Хомоненко А.Д., д.т.н., профессор, — предисловие, разделы 1-2, 5-13, 15-16, приложение.