Федеральное агентство по образованию
Государственное образовательное учреждение высшего
профессионального образования
Владимирский государственный университет

А.Г. САМОЙЛОВ, С.А. САМОЙЛОВ

ИНФОРМАТИКА В РАДИОСВЯЗИ И ТЕЛЕВИДЕНИИ: ОСНОВЫ ПРОГРАММИРОВАНИЯ НА C++

Учебное пособие
В двух частях
Часть 2

Владимир 2005

УДК 621.32 + 621.37
ББК 32.97 + 32.884.1 + 32.94
С17

Рецензенты:
Кафедра радиовещания и электроакустики Московского технического университета связи и информатики (МТУСИ), зав. кафедрой доктор технических наук, профессор
М.Д. Венедиктов

Кандидат технических наук, доцент
зав. кафедрой информатики и защиты информации Владимирского государственного университета
М.Ю. Монахов

Печатается по решению редакционно-издательского совета Владимирского государственного университета

С 17 **Самойлов, А.Г.** Информатика в радиосвязи и телевидении: основы программирования на С++ : учеб. пособие. В 2-х ч. Ч. 2 / А.Г. Самойлов, С.А. Самойлов; Владим. гос. ун-т. – Владимир: Изд-во Владим. гос. ун-та, 2006. – 118 с. – ISBN 5-89368-639-X

В первой части учебного пособия изложены основные общие теоретические вопросы информатики и наиболее важные вопросы практического применения вычислительной техники.

Во второй части – основы программирования на языке С++. Рассмотрены общие методы написания программ и наиболее важные вопросы практического применения языка программирования для решения прикладных задач. Для выполнения рейтинг-контроля успеваемости студентов все разделы учебного пособия сопровождаются контрольными вопросами.

Подготовлено для студентов первого курса технических специальностей 210405 (201100) – радиосвязь, радиовещание и телевидение, 210302 (200700) - радиотехника, 210303 (201500) - бытовая радиоэлектронная аппаратура, 210301 (071500) - радиофизика и др., но может применяться и при подготовке специалистов не технических направлений, а также лицами, самостоятельно изучающими вычислительную технику и программирование.

Табл. 3. Ил. 5. Библиогр.: 6 назв.

УДК 621.32 + 621.37
ББК 32.97 + 32.884.1 + 32.94

ISBN 5-89368-639-X

Введение

Программирование – это искусство нахождения точных и однозначных указаний, позволяющих механизмам решать конкретные задачи без участия человека. Языков программирования много, но язык C++ разработан для профессиональных программистов, решающих сложнейшие задачи, и в то же время он не нуждается в предварительных глубоких знаниях при освоении.

Язык C++ разработан Бьярном Страуструпом как надмножество языка C в 1980 –1983 гг. Знаки ++, придуманные Риком Маскитти, указывают на это расширение, поэтому программы, написанные на C, могут обрабатываться компиляторами C++. В C++ можно обращаться к библиотечным функциям языка C, но кроме этого C++ вводит понятие классов и поддерживает абстракцию данных, обеспечивая объектно-ориентированное программирование.

Для обучения программированию требуется:

1. Интерес к этой профессии. С помощью компьютера можно решать сверхсложные задачи физики, астрономии и др., но человечество в деле освоения компьютерной техники пока находится на уровне обезьян.
2. Как можно больше практиковаться. Ведь без велосипеда не научишься на нём ездить.
3. Настойчивость в поиске решений. Не подсказки, а самостоятельный поиск дает запоминающиеся ответы.
4. Элементарное умение загружать и открывать файлы, просматривать Web-страницы, пользоваться текстовым редактором и минимумом знаний по любому языку программирования, например по Бейсику.

Язык машин скучен, так как кроме нулей и единиц ничего не содержит. Язык программирования приближен к человеку, но для его использования на ЭВМ необходимы специальные программы перевода языка программирования в понятные машине сочетания единиц и нулей – компиляторы (в переводе с латыни – грабители). Компиляторы – это программы перевода команд с языка программирования на машинный язык.

Создание программы состоит из этапов, которые можно условно определить таким образом:

1. Разработка алгоритма программы.
2. Написание программы на языке C++ с расширением **.cpp**.
3. Трансляция программы компилятором в объектный файл с расширением **.obj**
4. Исправление обнаруженных компилятором ошибок и возврат к п. 3.
5. Компоновка программы с другими программами, требуемыми для решения задачи (например, вывод текста на экран или определение

места щелчка мышки нуждается в определенных программных кодах, которые при самостоятельном написании потребовали бы большого времени и замедлили бы процесс программирования). В результате получается исполняемый файл с расширением **.exe**. Обычно компиляторы делают это автоматически.

6. Запуск и исполнение программы.

Программы, написанные на языке C++, имеют расширение **.cpp.** Компиляторов программ с языка C++ несколько. Компания Borland Internetional разработала программное обеспечение Turbo C++ и более полную версию компилятора Borland C++. Компания Microsoft создала для этих же целей программный продукт Visual C++. Наиболее прост компилятор для среды MS-DOS, которым в основном и будем пользоваться. Особенности выполнения программ зависят от конкретного компилятора языка C++ и операционной среды для него и излагаются в техническом описании этого программного продукта. Независимо от используемого компилятора при работе в MS-DOS исполняемый модуль программы записывается в некоторый каталог (директорию).

Мышление человека инерционно, и поэтому часто при присвоении имен файлам человек повторяется. Файлы нужно объединять в папки и в этом случае можно иметь разные файлы с одинаковыми именами в различных папках. Группы программ составляют проект, другими словами, проект это несколько скомпонованных файлов, являющихся частями одной программы. Использовать готовые части программ разумно, т.к. существенно сокращается время разработки всей программы.

Создадим для обучения на диске C каталог C:\BOOK и будем в нём размещать подкаталоги. Компьютер всегда предполагает, что работа ведется с файлами текущего каталога, поэтому различит одноименные файлы в разных папках, т.к. мы работаем в конкретный момент времени в определенном каталоге. Поиск файла внутри каталога облегчается при использовании символа звездочка. Например, ***.cpp** означает все файлы с расширением .cpp, а ***.*.cpp** означает все возможные файлы с этим расширением.

Глава 1. БАЗОВЫЕ ПОНЯТИЯ С++

1.1. Первые программы на С++

Начнём с традиционной для начинающих программы, выводящей на экран монитора фразу Hello, World! – (Здравствуй, мир!).

Программа имеет вид

```
// First.cpp – имя файла с этой программой
#include <iostream.h>
void main()
{ cout <<"\nHello, world!\n";
getch();
}
```

Результат выполнения программы – появится фраза на экране **Hello, world!**

В первой строке текста программы – однострочный комментарий, начинающийся символом // и заканчивающийся неотображаемым символом - конец строки. Между этими символами может быть помещен любой текст, а в данном случае – имя файла с программой **first.cpp**.

Во второй строке текста команда препроцессору, обеспечивающая подключение стандартных средств ввода и вывода данных. Эти средства находятся в файле с именем **iostream.h**. В языке С++ активно используется английская мнемоника – **i** – input (ввод), **o** – output (вывод), **stream** – поток, **h** – head (заголовок). Стандартные средства ввода-вывода обеспечивают чтение данных от клавиатуры и вывод на экран дисплея.

Третья строка является заголовком функции с именем **main.** В С++ любая программа должна включать в себя только одну функцию с этим именем и именно с нее начинается выполнение программы. Перед именем **main** находится служебное слово **void –** это спецификатор, указывающий, что функция **main** в этой программе не возвращает никакого значения. Круглые скобки после main требуются в соответствии с синтаксисом для любой функции, и в них при необходимости размещается список параметров. В нашей программе списка нет.

Тело любой функции – это последовательность описаний, определений и операторов, которая обязательно заключается в фигурные скобки. Каждое описание, определение или оператор справа должны иметь точку с запятой. В нашем случае есть только один оператор

cout << "\nHello, world!\n";

Имя **cout** в соответствии с файлом **iostream.h** является именем объекта, обеспечивающим стандартный поток вывода информации на экран. Информация для вывода **"\nHello, world!\n"** передается этому объекту

cout с помощью операции **<<** (поместить в). Что необходимо вывести на экран, помещается справа от знака **<<**. В нашем случае это строка текста **Hello, world!.** Строка в С++ определяется как заключенная в кавычки **” ”** последовательность символов, в том числе и не отображаемых на экране. В нашем случае не отображаются управляющие символы перехода к началу следующей строки **\n** .

Функцию **getch ()** нужно ставить перед любым окончанием работы программы – эта команда ждет нажатия на любую клавишу.

Для усвоения материала приведем еще одну программу с текстовым содержанием. Но чтобы вывести её на экран монитора, необходимо заложить в неё команды для предварительного очищения экрана от прежней информации.

```
// Выводит текст стихотворения
#include <stdio.h>
#include <stdio.h>
void main ( )
{       clrscr();//очистка экрана
         printf (“Унылая пора! Очей очарованье!\n”) ;
         printf (“Приятна мне твоя прощальная краса -\ n”) ;
         printf (“Люблю я пышное природы увяданье,\ n”) ;
         printf (“В багрец и золото одетые леса.\ n\ n ”) ;
         printf (“A.C. Пушкин \ n”) ;
         printf (“\ n\ n Для завершения нажмите <Enter>”);
         getch ( ); //  чтобы стихотворение не исчезло с экрана
}
```

Отметим одну особенность языка С++, называемую **перегрузкой**, или расширением действия стандартных операций. Лексема **<<** означает в нашей программе операцию вставки (поместить в) только если слева от неё стоит имя объекта **cout**. Иначе << означает двойную операцию сдвига влево.

До выполнения программы мы сделали несколько операций, и схема подготовки к её выполнению выглядит так, как показано на рис. 1. Препроцессор обрабатывает директиву **#include<iostream.h>** и подключает к тексту программы средства для обмена с дисплеем и, выполнив директиву, формирует полный текст программы. Компилятор создает объектный файл ***.obj** с указанным ранее именем, а компоновщик добавит в программу библиотечные функции, например для работы с объектом **cout,** и создаст программный файл ***.exe**, который и запускается на выполнение.

Рассмотрим пример посложнее, например, расчет окладов по единой тарифной сетке оплаты труда в России. Исходные данные для расчета:

- минимальная ставка 1-го разряда (**smin**);
- массив тарифных коэффициентов **a[ ]**;

- номер разряда **r**.

Минимальную ставку задаём в самой программе, её же можно задавать, изменяя аргумент командной строки при запуске программы, а номер разряда задается с клавиатуры. То есть присутствуют три способа задания исходных данных в этой программе.

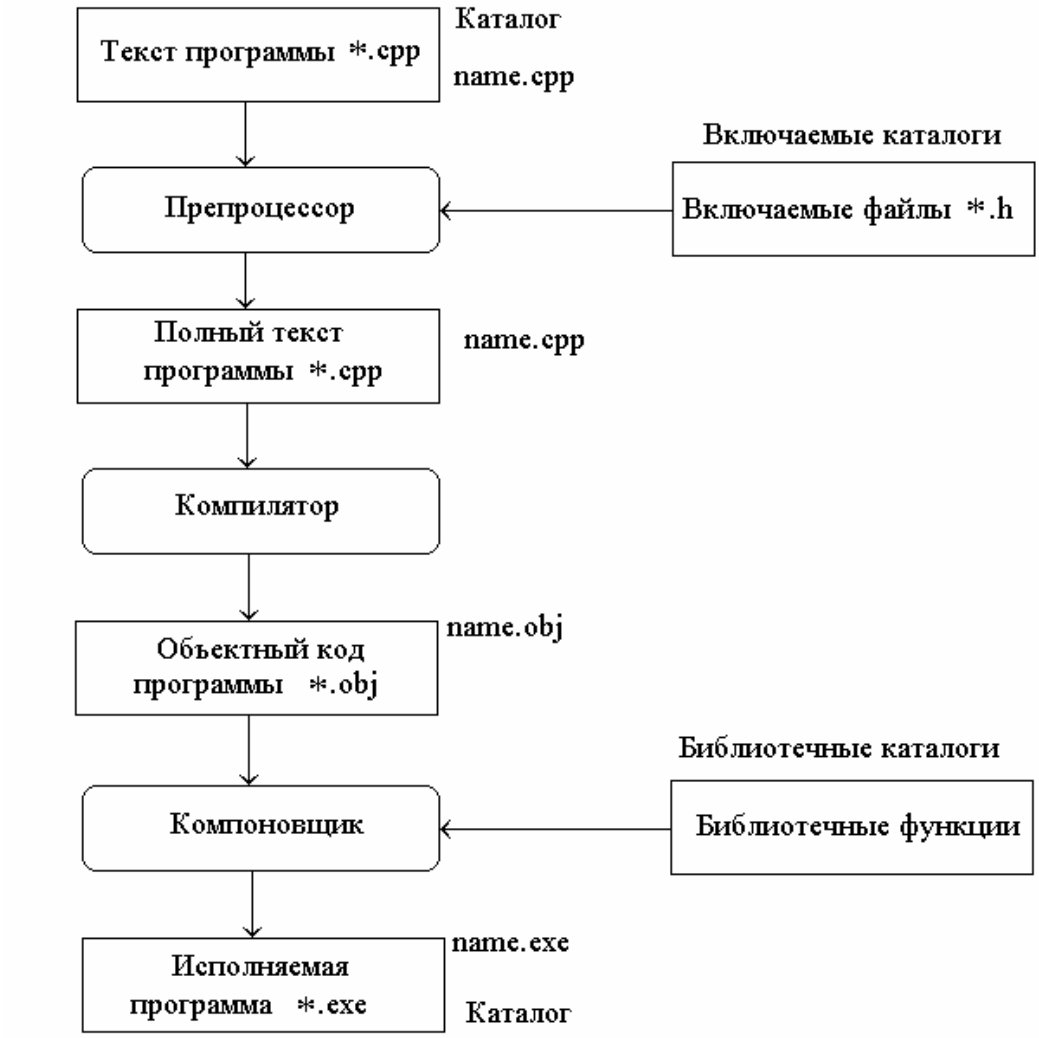

Рис. 1. Подготовка программы к исполнению

Дадим команду предпроцессору по подключению из файла **iostream** средств связи с библиотечными функциями потокового ввода-вывода. Там определены стандартные потоки ввода от клавиатуры **cin** и вывода данных на экран дисплея **cout**, соответствующие чтению из потока >> и операции записи в поток <<. Средства для обмена со строковыми потоками подключаются командой из файла **strstrea**. Строка – **str** (string), поток – **strea** (stream), заголовок – **h** (head) создают безымянный строковый поток, связанный со строкой **arg[1]**.

Функция **main ()** возвращает целочисленное значение **int**, которое после завершения программы передаётся в операционную систему для анализа. При аварийном завершении программы передается не нулевое значение, что позволяет компилятору найти ошибку. При правильном завершении в операционную систему передается нулевое значение.

В списке аргументов функции **main()** аргумент **int narg** передает в программу количество параметров, используемых в командной строке при запуске программы на исполнение. Если narg=1, то параметр в командной строке явно не указан. Спецификатор **float** указывает, что это вещественная арифметическая переменная в формате с плавающей запятой. Лексема >> означает извлечение данных из входного потока, только если слева от неё находится имя потока, например **cin >>**, иначе это просто двойной сдвиг вправо. **cin>>r** означает присвоение целочисленных значений переменной **r** (при этом нельзя набирать символы, отличные от цифр).

```
//Программа вычисления оклада
#include <conio.h>
#include<iostream.h>
#include<strstrea.h>
int main(int narg, char**arg)
{clrscr();
 float smin =  600; //Ставка первого разряда
 // a[] -массив значений тарифных коэффициентов:
 float a[] = { 1.00, 1.11, 1.23, 1.36, 1.51, 1.67, 1.84, 2.02, 2.22, 2.44,
 2.68,       2.89, 3.12, 3.36, 3.62, 3.90, 4.20, 4.50 };
  int r; //r - разряд тарифной сетки
  cout <<"\n";
  cout << "Программа вычисляет оклад в соответствии с единой";
  cout << "\nтарифной сеткой оплаты труда в России.\n";
  if (narg == 1)
{ cout << "\n По умолчанию минимальный оклад ";
cout << smin <<  "руб.";
cout << "\nПри необходимости изменить значение минимального\n";
cout << " оклада его нужно указать в командной строке.\n ";
}
  else
  { // Чтение из безымянного строкового потока:
  istrstream (arg[1]) >> smin;
  cout << "\nОпределен минимальный оклад";
  cout << smin << "руб.\n";
  }
cout << "\nВведите номер разряда тарифной сетки: ";
```

```
cin >> r; //Вводится с клавиатуры номер разряда
if (r < 1 || r > 18)
    { cout << "Ошибка в выборе разряда!";
    getch();
    return 1; //Аварийный выход из программы
    }
  cout << "Введенному разряду соответствует ставка  ";
  cout << (long) (a[r-1]*smin) <<       " руб.";
  getch();
  return 0;  //Безошибочное завершение программы
        }
```

1.2. Лексика C++. Основные сведения

Программа на языке C++ представляет собой текстовый файл с названием и расширением cpp, то есть **<имя>.cpp.** Имя выбирается произвольно, но не должно совпадать со служебными словами языка, которые рассмотрим ниже. Не следует применять длинные имена.

Для удобства нахождения и распознавания программ хорошим тоном считается в начале текста помещать строку комментария с именем файла программы. Комментарий можно задавать двумя способами – традиционным для многих языков программирования, ограничивая его слева двумя символами /* и справа двумя символами */, либо ограничивая слева двумя символами // и справа переходом на новую строку \n. В комментарии можно помещать любой текст, в том числе и русский.

Текстовый файл обрабатывает препроцессор, распознающий команды, которые начинаются с символа # . Так команда #include<имя включаемого файла> вставляет в текст ранее приготовленные файлы. При этом текст программы изменяется и поступает на компиляцию. Компилятор выполняет две функции – во-первых, выделяет из текста лексические элементы **(лексемы)**, а во-вторых, распознает конструкции языка из этих лексем. В итоге получается объектный модуль программы. Для разделения лексем в C++ используются символы пробелов – пробела, табуляции, перехода на новую строку.

Алфавит C++ содержит:

- прописные и строчные буквы латинского алфавита;
- цифры 0. 1. 2. 3. 4. 5. 6. 7. 8. 9;
- знаки * . ? # { } / \ & _ ! : ” , ’ [ ] ( ) + - ; < > = ^ %

Из символов алфавита строятся лексемы C++, к которым относятся: идентификаторы, служебные слова, разделители, константы, знаки операций.

1.3. Идентификаторы

Это последовательность латинских букв и знаков, начинающаяся не с цифры. Прописные и строчные буквы различаются, поэтому имена BOY и boy различаются. Длина идентификатора для разных компиляторов различна, например, для компиляторов Borland не более 32 символов, а для некоторых не более 8 символов.

Идентификаторы, используемые в С++ для служебных целей, называют ключевыми словами. Их следует знать, чтобы не присваивать такие имена. В языке С++ ключевых слов 50:

asm	double	new	smitch
auto	else	operator	template
break	enum	overload	this
case	extern	private	throw
catch	float	protected	try
char	for	public	typedaf
class	friend	register	typeid
const	goto	return	union
continue	if	short	unsigned
default	inline	signed	virtual
delete	int	sizeof	void
do	long	static	volatile
		struct	while

Компиляторы фирмы Borland имеют дополнительные ключевые слова

cdel	_export	_loadds	_saveregs
_cs	far	near	_seg
_ds	huge	pascal	_ss
_es	interrupt	_regparam	

Не следует начинать имена с двух подчеркиваний, так как этот знак __ резервируется для библиотек С++, да и с одного подчеркивания _ не следует, так как этот символ используется как идентификатор в языке Си.

1.4. Константы

Константа – это лексема, которая изображает числовое, литерное (символьное) или строковое значение. Константы делят на пять групп: целые, вещественные с плавающей точкой, перечислимые, строковые, символьные. В программировании С++ дробную часть числа отделяют от целой части точкой, так как знак запятой является определенной операцией.

Целочисленные константы разделяют на десятичные, восьмеричные и шестнадцатеричные. Отрицательные константы организуют применением операции изменения знака. Абсолютные значения констант С++ зависят от типа данных, в зависимости от которого компилятор различным образом представляет данные. Типов данных для целых констант четыре:

- **int** и **unsigned** для чисел до 32767;
- **long** для чисел от 32768 до 2147483647;
- **unsigned long** для чисел от 2147483648 до 4294967295.

Тип данных автоматически присваивается компилятором, но если он не устраивает программиста, то можно прибавлением суффиксов **L** (long) и **u** (unsigned) или их вместе **Lu**, а можно и **uL,** изменять тип данных. Восьмеричные константы начинаются всегда с нуля, а шестнадцатеричные с символов 0x. Отрицательные константы не должны превышать значения 2147483648, то есть двух с небольшим миллиардов.

Вещественные константы с плавающей точкой даже не отличаясь от целых констант по значению по другому представляются в ЭВМ. Константа с плавающей точкой может включать шесть частей: целую десятичную часть, точку, дробную часть, признак экспоненты **e** или **E**, показатель десятичной степени (десятичная целая константа, возможно со знаком), суффикс **F (f)** или **L (*l*)**. В записях вещественных констант могут опускаться (не одновременно) либо целая, либо дробная часть или суффикс. Например: 345. . .007 4.156F 33.42e-3 2E+7L 5.67

При отсутствии суффиксов в С++ тип данных для вещественных констант **double**, имеющий размер 64 бит. Добавив суффикс **F** или **f,** получим тип данных **float** с диапазоном значений от 3.4E-38 до 3.4E+38 и размером 32 бит. Добавив суффикс **L** или **l,** получим тип данных **long double** с диапазоном от 3.4E-4932 до 1.1.E+4932 и размером 80 бит.

Напишем программу по определению размера памяти для констант

```
// Размер памяти для констант разного типа
#include <iostream.h>
#include<conio.h>
void main( )
{ clrscr( );
  cout << ”\n sizeof 485 = ” << sizeof 485;
  cout << ”\n sizeof 48500u = ” << sizeof 48500u;
  cout << ”\n sizeof 485L = ” << sizeof 485L;
  cout << ”\n\t sizeof 4.85 = ” << sizeof 4.85;
  cout << ”\n\t sizeof 4.85f = ” << sizeof 4.85f;
  cout << ”\n\t sizeof 4.85L = ” << sizeoL 4.85f;
  getch( );
}
```

По результатам выполнения получим на зкране монитора надпись – размер в байтах, занимаемый соответствующими данными:

sizeof 485 = 2
sizeof 48500u = 2
sizeof 485L = 4
sizeof 4.85 = 8
sizeof 4.85f = 4
sizeof 4.85L = 10

В этой программе использована операция языка C++ **sizeof**, позволяющая определять размер в байтах, выделенный в памяти для стоящего справа операнда. Пара символов **\t** в тексте программы означают табуляцию при выводе текста. Как видно, при использовании для целых чисел, превышающих значение 32767 суффикса **u,** и при использовании для вещественных чисел с плавающей точкой суффикса **f** получаем выигрыш в размере занимаемой в ЭВМ памяти.

Константы перечисления вводятся с помощью служебного слова **enum.** Это обычные целочисленные константы с удобными обозначениями, присваиваемыми им с помощью определения

enum { one = 12, two = 275, three = 3);

где **enum –** служебное слово, определяющее перечисление, а one, two, three – условные имена для обозначения констант. Далее в программе вместо 275 можно использовать её обозначение two. Если в определении констант не указывать знак = и числовых значений констант, то они будут приписываться именам по умолчанию. Самый левый в фигурных скобках получит значение 0, а каждый следующий будет увеличиваться на единицу. Например, для нашего случая **enum** {one, two, three}; получим one= =0, two= =1, three= =2. Правило действует и когда нескольким слева присвоены конкретные значения. Например, команда **enum {one =12, two=275, three};** введёт такие константы как one = =12, two = = 275, three = =276.

Одно и то же значение могут иметь несколько перечислимых констант. Например, **enum** {one=12, two, three = 12} даст one = = 12, two = = 13, three = =12. Значения перечислимых констант можно задавать выражениями, например, так **enum** {two = 2, four = two*2};.

Для перечислимых констант можно после служебного слова enum вводить имя типа для приведенного списка. Например,

enum name {oly, vasy, petr, gena};.

Символьные (литерные) константы – это один или два символа, заключенные в апострофы. Односимвольные константы имеют тип **char,** занимают в памяти один байт и для их представления нужно вводить переменную символьного типа, т.е. типа **char.** Пример односимвольных кон-

стант: `*`, `\0`, `\025`, двухсимвольных: `bb`, `\x09`, `\n\t`. Символьная константа не может быть пустой и запись двух апострофои ` ` недопустима.

Отметим важную роль символа обратная косая черта. С её помощью выводятся символы апострофа, обратной косой черты, знака вопроса, кавычек и символьные константы при задании их кодов в восьмеричной или шестнадцатеричной форме. Литеры, начинающиеся со знака \ , называют эскейп-последовательностями. Часто используется последовательность **`\0`,** обозначающая пустую **(null)** литеру. В табл. 1 приведены часто используемые эскейп-последовательности для языка С++.

Таблица 1. Эскейп-последовательности языка программирования С++

Изображение	Код ASCII	Символ	Действие
\a	0x07	be1(audio bell)	Звуковой сигнал
\b	0x08	bs (backspace)	Возврат на шаг (забой)
\f	0x0C	ff (form feed)	Перевод страницы
\n	0x0A	lf (line feed)	Перевод строки
\r	0x0D	cr (carriage return)	Возврат каретки
\t	0x09	ht (hjrizontal tab)	Табуляция горизонтальная
	0x0B	vt (vertical tab)	Вертикальная табуляция
\\	0ч5C	\ (backslash)	Обратная косая черта
\`	0x27	` (singe quote)	Апостроф
\”	0x22	”(double quote)	Двойная кавычка
\?	0x3F	? (question mark)	Знак вопроса
\000	000	Любой	Восьмеричный код символа
\xhh	0xhh	Любой	Шестнадцатеричный код символа

Можно писать программу символами кода ASCII, например:

```
// Использование кодов ASCII
#include <iostream.h>
#include<conio.h>
void main()
{ clrscr();
cout << `\x0A` << `\x48` << `\x65` << `x6C` << `x6C` << `x6F` << `\41`;
getch();
}
```

На экран будет выведено Hello!, так как`\x0A` с новой строки, `\x48` - H, `\x65` - e, `x6C` - l, `x6C` - l, `x6F` - o, `\41`- !

В примере использованы шестнадцатеричные коды символов, но можно использовать и восьмеричные. В качестве примера определения кода символа можно предложить следующую программу:

```
// Выводит код символа
# include <stdio.h >
```

```
# include <conio.h >
void main ( )
{ unsigned char ch;
// Если ch объявить как  char, то буквам русского
// алфавита будут соответствовать отрицательные числа
printf (“\nВводите символы. \n ”);
printf (“Для завершения введите точку. \n ”);
do  { ch = getch( );
        printf (“Символ: %c  Код:  %i\n”, ch, ch);
    }
while ( ch ! =  ’ . ‘ );
printf (“Для завершения работы нажмите <Enter>”);
getch ( ) ;
}
```

Строковая константа, или литерная строка определяется как последовательность символов, заключенная в кавычки. В состав строки могут входить символы, задаваемые кодами. Строки, записанные через пробельные разделители при компиляции склеиваются (конкатенируются). Разделять их можно по-разному – вводя перенос \n или просто символ пробела \ , но продолжение перенесенной с помощью пробела строки будет через количество символов на следующей строке, включая пробелы.

Например, программа

```
// Использование кодов ASCII
#include <iostream.h>
#include<conio.h>
void main()
{clrscr();
  cout <<  ”\n 20”
 ”января” ; // При выводе пробелов не будет
 cout <<  ”\n был \вторник.”; //  Пробелы в строке сохранятся
 getch ();
}
```

Программа даст результат в виде:

```
20 января
был                          вторник.
```

Размещая строку в памяти, компилятор автоматически добавляет в её конец символ ` \0 `, то есть нулевой байт, и количество символов в представлении строки всегда на один больше, чем в её записи. Кавычки без косой черты не входят в строку, а служат её ограничителями. Если в начале или в конце строки необходимы кавычки, то требуется запись с символом кавычки, например, ” …строка текста \”a\”.” Тогда будет выведено

…строка текста ”a”. Строка может быть пустой ” ”;, а может иметь один символ, например ”M”.

Не следует путать строку с символьной константой, ограничиваемой не кавычками, а апострофами, например `A `. Можно строку помещать в символьный массив типа **char** с выбранным именем и в дальнейшем обращаться к ней по имени, например:

```
// Приветствие
# include <stdio.h >
# include <conio.h >
void main ( )
{      char name [15 ];  // имя
        char fam  [20];    // фамилия
        printf (“Как вас зовут?\n ”);
        printf (“Введите свое имя и фамилию,”);
        printf (“затем нажмите <Enter>”);
        printf (“->  ”);
        scanf (“%s”,  &name);
        scanf (“%s”,  &fam); // функция scanf читает из буфера клавиатуры символы до разделителя – пробела
        printf (“Здравствуйте,  %s  %s! \n ”,  name,  fam);
        printf (“\ nДля завершения нажмите <Enter>”);
        getch ( ) ;
}
```

1.5. Знаки операций

Кроме операций **[ ] ( )** и **?:** все они распознаются компилятором как отдельные лексемы, но в зависимости от контекста программы одна и та же лексема может обозначать разные операции. В этом и состоит главное отличие языка C++ от языка C. В C++ есть возможность расширения действия стандартных операций (перегрузка – overload) и распространение их действия на нестандартные операции.

В C++ определены следующие знаки операций:

[ ]	**( )**	**.**	**->**	**++**	**--**	**: :**	**.***
&	*****	**+**	**-**	**~**	**!**	**->***	**new**
sizeof	**/**	**%**	**<<**	**>>**	**<**	**>**	**<=**
>=	**= =**	**!=**	**|**	**&&**	**||**	**?:**	**=**
***=**	**/=**	**%=**	**^**	**+=**	**-=**	**<<=**	**>>=**
&=	**^=**	**| =**	**,**	**#**	**##**	**delete**	**typeid**

Унарные операции

& операция получения адреса;
* операция обращения по адресу;
- изменение знака;
+ знак плюс;
~ побитовое отрицание (поразрядное инвертирование внутреннего двоичного кода целочисленного аргумента);
! логическое НЕ, применяемое к скалярным величинам. В С++ применяют в качестве логических значений нуль (0) – ложь и не нуль (! 0) – истина. Отрицанием любого ненулевого числа будет ноль, а отрицанием нуля будет единица – 1. То есть: !0 равно 1, !(-9) равно 0, !1 равно 0, !2 равно 0 и т.д.;
++ увеличение на единицу значения операнда до или после использования операнда. При этом операнд не может быть константой, то есть записи 6++ или ++18 неверны. Операндами должны быть переменные;
-- уменьшение на единицу, аналогичное увеличению ++;
sizeof - операция вычисления в байтах для объекта.

Бинарные операции

Этих операций 11 групп. Рассмотрим их по порядку.

Аддитивные операции

Бинарный плюс и бинарный минус – сложение или вычитание арифметических операндов.

Мультипликативные операции

* умножение операндов арифметического типа;

/ деление операндов арифметического типа. При целочисленных операндах абсолютное значение результата округляется до целого, например, 20/3 равно 6, -20/6 равно –3;

% получение остатка от деления целочисленных операндов (деление по модулю). Всегда выполняется соотношение (a/b)*b+a%b равно a.

Операции сдвига

<< и >> сдвиг влево или вправо битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого операнда.

Поразрядные операции

& поразрядное И,

| поразрядное ИЛИ,

^ поразрядное исключающее ИЛИ.

Приведем пример программы на операции сдвига и поразрядные операции.

// Сдвиг и поразрядные операции

```
#include <iostream.h>
#include<conio.h>
void main()
{ clrscr();
 cout << ”\n4<<2 равняется ” << (4<<2);
cout << ”\t5>>1 равняется ” << (5>>1);
cout << ”\n6&5 равняется ” << (6&5);
cout << ”\t6|5 равняется ” << (6|5);
cout << ”\t6^5 равняется ” << (6^5);
getch();
}
```

В результате выполнения программы получим:

4<<2 равняется 16 5>>1 равняется 2

6&5 равняется 4 6|5 равняется 7 6^5 равняется 3.

Двоичный код 4 будет 100 и при сдвиге влево на 2 позиции станет 10000, код 5 это 101, 6 – 110. Сдвиг влево на n позиций эквивалентен умножению на 2^n, а вправо – уменьшает значение в 2^n раз с отбрасыванием дробной части результата, и поэтому 5>>1 равно 2.

Операции отношения

< меньше чем;

> больше чем;

<= меньше или равно;

>= больше или равно;

= = равно;

!= не равно.

Операции сравнения на равенство имеют низкий приоритет и, например, в выражении d<c = = b<d сначала вычисляются неравенства и только потом применяется операция сравнения на равенство.

```
// Пример программы сравнения двух целых чисел
# include <stdio.h >
# include <conio.h >
void main ( )
{int a,b;       // сравниваемые числа
 printf (“\ nВведите в одной строке два целых ”);
 printf (“числа и  нажмите <Enter>”);
 printf (“->”);
 scanf (“%i%i ”,  &a,  &b);
        if  (a = = b)
        printf (“Числа равны ”);
 else if (a < b)
        printf (“%i меньше % i \n”,  a, b);
```

```
            else   printf ("%i больше % i \n",  a, b);
        printf ("\ nДля завершения нажмите <Enter>");
        getch ( ) ;
        }
```

Логические бинарные операции

&& – операция И арифметических операндов – целочисленный результат 0 (ложь) или 1 (истина);

|| – операция ИЛИ арифметических операндов или отношений, целочисленный результат 0 (ложь) или 1 (истина).

Рассмотрим на примере программы.

```
#include <iostream.h>
#include<conio.h>
void main()
{ clrscr();
 cout << "\n4>2 равняется " << (4>2);
cout << "\t5>8 равняется " << (5>8);
cout << "\n6= =7 равняется " << (6= =7);
cout << "\t6!=5 равняется " << (6!5);
        cout << "\n3+4>5 && 3+5>4  && 4+5>3 равняется " << (3+4>5 &&
        3+5>4 && 4+5>3);
getch();
}
```

В результате выполнения получим:

4>2 равняется 1;
5>8 равняется 0;
6= =7 равняется 0;
6!=5 равняется 1;
3+4>5 && 3+5>5 &&4+5>3 равняется 1.

Операции присваивания

= присвоить значение выражения операнда из правой части операнду левой части P= 15.7-6*y;

= присвоить операнду левой части произведение значений обоих операндов, то есть P=7 эквивалентно P=P*7;

/= присвоить операнду левой части частное от деления значения левого операнда на значение правого P/= 3.3+t эквивалентно P=P/(3.3+t);

%= присвоить операнду левой части остаток от деления целочисленного значения левого операнда на целочисленное значение правого операнда P %= 6 эквивалентно P=P%6;

+= присвоить операнду левой части сумму значений обоих операндов P+=B эквивалентно P=P+B;

-= присвоить операнду левой части разность значений левого и правого операндов X-=6.7+y эквивалентно X=X-(6.7+y);

<<= присвоить целочисленному операнду левой части значение, полученное сдвигом вправо его битового представления на количество разрядов, равное значению правого целочисленного операнда v << 6 эквивалентно V=V<<6;

>>= присвоить целочисленному операнду левой части значение, полученное сдвигом вправо его битового представления на количество разрядов, равное значению правого целочисленного операнда a..=6 эквивалентно a = a>>6;

&= присвоить целочисленному операнду левой части значение, полученное поразрядной конъюнкцией (И) его битового представления с битовым представлением целочисленного операнда правой части x & 76 эквивалентно x = x & 76;

| = присвоить целочисленному операнду левой части значение, полученное поразрядной дизъюнкцией (ИЛИ) его битового представления с битовым представлением целочисленного операнда правой части b | = m эквивалентно b = b | m;

^= присвоить целочисленному операнду левой части значение, полученное применением поразрядной операции исключающего ИЛИ к битовым представлениям значений обоих операндов X ^= N +Y эквивалентно X = X ^(N+Y).

Операции выбора компонентов структурированного объекта

. точка – прямой выбор компонента структурированного объекта. Формат операции *Имя структурированного объекта . имя компонента*;

-> косвенный выбор (выделение) компонента структурированного объекта, адресуемого указателем. Формат операции *Указатель на структурированный объект -> имя компонента.*

Операции с компонентами классов

.* прямое обращение к компоненту класса по имени объекта и указателю на компонент;

->* косвенное обращение к компоненту класса через указатель на объект и указатель на компонент;

:: операция указания области видимости, имеющая две формы – бинарную для доступа к компоненту класса и унарную для доступа к внешней для некоторой функции именованной области памяти.

Запятая в качестве операции

Несколько выражений, разделенных запятыми, вычисляют последовательно слева направо. Тип и значение результата определяются самым правым из разделенных запятыми операндов.

Скобки в качестве операций

Круглые ( ) и квадратные [ ] скобки играют роль бинарных операций при вызове функций и индексировании элементов массивов. Круглые скобки обязательны в обращениях к функциям *имя функции (список аргументов)*. Здесь операнды *имя функции* и *список аргументов.*

В языках С и С++ принято, что индексы массивов начинаются с нуля, то есть массив **int Z[3]** из трех элементов состоит из элементов Z[0], Z[1], Z[2]. Нужно учитывать, что индекс определяет не номер элемента, а его смещение относительно начала массива. То есть Z[0] – обращение к первому элементу, Z[1] – обращение ко второму элементу, Z[2] – к третьему.

Условная операция

Это операции в отличие от унарных и бинарных с тремя операндами. В изображении условной операции два не подряд идущих символа ? и : и три операнда

Выражение1 ? Выражение 2 : Выражение 3

Первым вычисляется выражение 1. Если оно не рано нулю, то вычисляется выражение 2, которое и становится результатом. Если выражение 1 равно нулю, то результатом становится значение выражения3.

Операция явного преобразования (приведения) типа

В языке С++ эта операция имеет две формы – каноническую и функциональную.

Каноническая форма имеет формат *(имя типа) операнд*. В качестве операнда может быть заключенное в круглые скобки выражение, переменная, константа, например (long double)1.

Функциональная форма используется тогда, когда тип имеет простое (несоставное) наименование, например long (2) – внутреннее представление имеет длину 4 байта.

Операции new и delete для динамического распределения памяти

Эти две унарные операции связаны с распределением памяти. Операция **new имя типа** или **new имя типа инициализатор** позволяет выделить и сделать доступным свободный участок в основной памяти, размеры которого соответствуют типу данных, определяемому именем типа. В выделенный участок заносится значение, определяемое инициализатором, который не является обязательным элементом. В случае успешного выполнения операция **new** возвращает адрес начала выделенного участка памяти. Если памяти нужного размера нет, то операция **new** возвращает нулевое (NULL) значение адреса.

Операция **new float** выделяет участок памяти в 4 байта. Операция **new int(15)** выделяет участок памяти в 2 байта и инициализирует его целым значением 15. Синтаксис операций **new** и **delete** предполагает применение указателей, которые должны быть предварительно определены. Определение указателя имеет вид

тип * имя указателя

Имя указателя – это идентификатор, а в качестве типа можно использовать стандартные типы **int, long, float,double,char.** Например, **int*h;** определение указателя, который может быть связан с участком памяти, выделенным для величины целого типа. Вводя с помощью определения указатель, можно присвоить ему возвращаемое операцией **new** значение **h = new int (15);** В дальнейшем доступ к выделенному участку памяти обеспечивает выражение ***h.**

При отсутствии инициализатора значение, которое заносится в выделенный участок памяти не определено. Если в качестве имени типа используется массив, то для массива должны быть определены все размерности. Продолжительность существования выделенного с помощью операции **new** участка памяти от момента создания до конца программы или до его явного освобождения.

Для явного освобождения используется оператор **delete указатель**. **Указатель** адресует освобождаемый участок памяти, ранее выделенный с помощью операции **new**. Например, **delete h;** освободит участок памяти, связанный с указателем **h.** Повторное применение операции **delete** к тому же указателю дает неопределенный результат. Также непредсказуем результат применения этой операции к указателю, получившему значение без операции **new.**

1.6. Ранги операций

В С++ 16 категорий приоритетов операций (табл. 2). С ростом номера категории приоритет ослабевает. Если операции одной категории имеют одинаковый приоритет, то выполняются в соответствии с правилом ассоциативности по стрелке. Если один и тот же знак в таблице приоритетов встречается дважды, то старший по приоритету соответствует унарной операции, и лишь затем бинарной.

Таблица 2. Приоритеты операций

Категория	Операция	Ассоциативность
1	( ) [ ] -> : : .	→
2	! ~ + - ++ -- & * (тип) sizeof new delete тип ( ) (функциональное преобразование типа)	←
3	.* ->*	→
4	* / & (мультипликативные бинарные операции)	→
5	+ - (аддитивные бинарные операции)	→
6	<< >>	→
7	< <= >= >	→
8	= = !=	→
9	&	→
10	^	→
11	|	→
12	&&	→
13	||	→
14	? : (условная операция)	←
15	= *= /= %= += -= &= ^= |= <<= >>=	←
16	, (операция запятая)	→

1.7. Разделители

Разделители или знаки пунктуации используются следующие:

[ ] ( ) { } , ; : … * = # &

Квадратные скобки [ ] ограничивают массивы

// Одномерный массив из 6 элементов:

int X[ ] = {4, 3, 3, 7, 1, 8};

// y- двумерный массив – матрица размерности 3x2:

int x, y [3] [2];.

Круглые скобки ( ):

- выделяют условные выражения (в операторе “если”):

if ([,0) x = - x; // Модуль значения арифметической переменной;

- обязательные элементы любой функции – выделяют список параметров; D(X,N) ; // Вызов функции;

- обязательны в определении указателя на функцию:

int (*func) (void); // Определение указателя на функцию;

- группируют выражения, изменяя последовательность операций:

X = G (b+d); // Изменение приоритета операций;

- входят как обязательные элементы в операторы циклов:

for (i = 0, j = 1; i<k; i+ =2, j++) тело цикла;
while (i<j) тело цикла;
do тело цикла while (m>0);

- необходимы при преобразовании типа, например:

```
long k = 21N; int j; // Определение переменных;
j = int (k);              // Функциональная запись преобразования;
float G;                  // Определение переменной;
G = (float) j             // Явное приведение типа.
```

В результате j получит значение 21N, преобразованное к типу **int,** затем G получит значение 21N, преобразованное к типу **float;**

- рекомендуется в макроопределениях, обрабатываемых препроцессором:

```
# define G(h, k)    sqrt ((h)*(h)+(k)*(k))
```

, что позволяет использовать выражения любой сложности без нарушений приоритетов.

Фигурные скобки { } обозначают:

- начало и конец составного оператора или блока, например:

```
if (h>d)  {h++; d--;}
```

или блок, являющийся телом функции:

```
float exponent (float x, int n)
{
float d = 1.0; int i = 0;
if (x= = 0) return 0.0;
for (; i<abs(n); i++, d*=x);
return n> 0 ? d : 1.0/d;
}
```

После закрывающей составной оператор фигурной скобки точка с запятой не нужна;

- выделение списка компонентов в определениях типов структур, объединений, классов:

```
struct cell { char *b;       // Определение типа структуры
              int ee;
              double U [ 6 ];
            };
union smes  { unsigned  int ii;  // Определение типа объединения
              char   cc [2];
             };
class sir { int B; //  Определение класса
                    public;
                    int X, D;
                    sir (int) ;};
```

Здесь точки с запятой в конце нужны;

- инициализация массивов и структур при их определении // Инициализация массива:
int month [] = {1,2,3,4,5,6,7,8,9,10,11,12}; // Инициализация структуры
struct mixture { int ii; double dd; char cc;}
stock = { 432, 4,85, '\t'};

Здесь **mixture** – имя типа структуры с тремя разновидными компонентами, **stock** – имя конкретной структуры типа mixture. Компоненты ii, dd, cc структуры stock получают значения при инициализации.

Запятая ' , ' разделяет элементы списков и используется как разделитель в заголовках операторов, описаниях и определениях объектов одного типа.

Следует обратить внимание на необходимость с помощью круглых скобок отделять запятую – знак операции от запятой – разделителя. Например (k=7,k*k) – здесь запятая как операция. Операция присваивания имеет более высокий приоритет, чем запятая, поэтому сначала k получает значение 7, затем вычисляется произведение k*k, и этот результат служит значением в скобках, а значением переменной k остается 7.

Точка с запятой *завершает каждый оператор, каждое определение (кроме определения функции) и каждое описание. Символ точка с запятой считается пустым оператором и его можно использовать как тело цикла.*

Двоеточие служит для отделения (соединения) метки и помечаемого ей оператора, например bb: = (c+h)*(k-g);. Также используется для описания производного класса, отделяемого двоеточием от базовых.

Многоточие … обозначает переменное число параметров у функции при ее определении и описании. Указывает возможность использовать разное количество параметров.

Звездочка (*) указывает на операции умножения и разименования (получения значения через указатель). В описаниях и определениях звездочка означает, что описывается указатель на значение использованного объявления типа:

int* point; // Указатель на величину типа int.

Знак = обозначает операцию присваивания, а в описаниях отделяет описание объекта от списка его инициализации:

struct {char y, int z} B = { 'x', 23};

int g = 989.

Знак равно в списке формальных параметров функции указывает на значение аргумента, выбираемое по умолчанию

char G (int V=15, char H = '\0') {…} – по умолчанию параметр V=15, параметр H=0/.

Символ # (диеза в музыке) применяют для обозначения директив препроцессору.

Символ & используется как разделитель при определении переменных типа ссылки:

int A; // Описание переменной

int &A = G; // A – ссылка на G.

1.8. Стадии препроцессорной обработки

Препроцессор делает следующее:

- все системно-зависимые обозначения перекодирует в стандартные коды;
- каждая пара символов \ и ”конец строки” убираются из строки, присоединяя к ней следующую строку ;
- распознаются директивы препроцессора, а каждый комментарий заменяется одним символом пустого промежутка;
- выполняются директивы препроцессора и макроподстановки;
- ESC последовательности (например \n, \t и т.п.) заменяются на их числовые коды;
- смежные символьные строки котенируются, т.е. соединяются в одну строку.

Для управления препроцессором, т.е. задания необходимых действий используются директивы, начинающиеся с символа #. Директив много, рассмотрим некоторые из них.

Директива **# define** используется для работы с макросами. **Макрос** – это средство замены одной последовательности символов другой. Например, при заменах в тексте используется директива

 # define идентификатор строка_замещения

т.е. **# define stroka** “\n Бит не синяк под глазом, а единица информации!”

После такой команды все обращения к идентификатору **stroka** приведут к замене этого идентификатора текстом строки **Бит не синяк под глазом, а единица информации!**

Если строка длинная, то её можно обратной наклонной чертой \ переносить на другую строку программы, а в выходном тексте этот символ \ будет удалён. Предусмотренные директивой # define препроцессорные замены не выполняются внутри текстов, ограниченных кавычками, апострофами и разделителями (/* ,*/), но строка замещения может содержать такие ограничители.

Директиву **# define** используют для обработки макроопределений следующим образом: **# define max(a,b) (a<b ? b : a)**

Это пример нахождения максимального из двух чисел. При таком определении вхождение в программу **max(X,Y)** заменяется выражением

(X < Y ? Y : X) – при истинном значении **X < Y** возвращает значение **Y**, иначе возвращает значение **X**. Между именем макроса **max** и списком параметров **(a,b)** не должно быть пробелов.

Директива **# include** используется для включения текста файлов. Если имя файла в угловых скобках, то он в стандартных системных каталогах, если в кавычках, то препроцессор сначала рассматривает каталог пользователя и только затем обращается к стандартным системным каталогам. Имена стандартных или подготовленных специально файлов должны иметь общепринятое расширение суфиксом **.h**. Файлы с таким суффиксом **.h** размещают в заголовке до исполнения операторов.

Директивы условной компиляции таковы

 # if … текст_1 , **#else текст_2** , **#endif** .

Конструкция **#else текст_2** не обязательна. Текст 1 включается в компилируемый текст в том случае, если проверяемое условие истинно. Если оно ложно, то при наличии директивы **#else текст_2** на компилятор передается Текст 2. Если директивы **#else текст_2** нет, то весь текст от **#if** до **#endif** опускается.

Контрольные вопросы

1. Какими символами в С++ обозначаются директивы препроцессора, комментарии, окончание строки?
2. Что такое тело функции?
3. С какой функции начинается выполнение программы?
4. Приведите примеры идентификаторов в С++.
5. Перечислите типы констант в С++.
6. Приведите примеры унарных операций в С++.
7. Приведите примеры бинарных операций В С++.
8. Что такое операция «Запятая»?
9. Приведите пример условной операции.
10. Для чего используются операции new и delete?
11. Перечислите основные стадии препроцессорной обработки.
12. Для чего используют в С++ директиву препроцессора #define?

Глава 2. СКАЛЯРНЫЕ ТИПЫ И ВЫРАЖЕНИЯ

2.1. Основные и производные типы

В программировании переменную обычно определяют парой обозначений “имя” – ”значение”. **Именем** является адрес, то есть идентификатор участка памяти для этой переменной, а **значение** определяет тип переменной, определяющий состав этого участка памяти, то есть множество допустимых значений и набор операций, для которых переменная служит операндом. Тип данных определяет размер выделяемого по них участка памяти, как показано в табл. 3.

Имена типов переменных описываются ключевыми словами:

- **char –** символьный;
- **short** – короткий целый;
- **int** - целый;
- **long** –длинный целый;
- **float** – вещественный;
- **double** – вещественный с удвоенной точностью;
- **void** – отсутствует значение (пустой).

При определении переменных им можно приписывать начальные значения, например, **double pi** = 3.141592653589793. Можно использовать несколько слов, например, **long double man, girl** вводит имена man, girl вещественного типа повышенной точности, не присваивая им начальных значений. Применяемые отдельно или с другими именами типов слова **unsigned** – беззнаковый и **signed –** знаковый позволяют выбрать способ учета знакового разряда. Например, следующие определения позволяют переменным принимать только целые положительные значения:

- **unsigned int** a,b,c // Значения от 0 до 65535
- **unsigned long** d,e,f // Значения от 0 до 429467295
- **unsigned char** h, n, m // значения от 0 до 255

Слово **signed** обычно опускается в определениях, иначе его потребовалось бы ставить перед большинством имен, и пишут не **signed int,** а просто **int.** В свою очередь, слово **unsigned** без определения типа означает **unsigned int.**

Переменные одного типа занимают в памяти одно и то же количество байт, и это легко проверить с помощью операции **sizeof**.

Используя спецификатор **typedef,** можно вводить в программе обозначения для сложных описаний типов, например

```
typedef unsigned char COD;
COD symbol;
```

Вместо **unsigned char** введен новый тип **COD** и его переменная **symbol** со значениями от 0 до 255.

Таблица 3 Основные типы данных

Тип данных	Размер, бит	Диапазон	Назначение
unsugned char	8	0 … 255	Небольшие целые и коды символов
char	8	-128…127	Малые целые и ASCII коды
enum	16	-32768…32767	Наборы целых значений
unsigned int	16	0…65535	Большие целые, счетчики циклов
short int	16	-32768…32767	Небольшие целые, управление циклами
int	16	-32768…32767	Небольшие целые, управление циклами
unsigned long	32	0…4294967295	Астрономические расстояния
long	32	-2147483648… …2147483647	Большие числа
float	32	3.4E-38…3.4E+38	Научные расчеты (7цифр)
double	64	1.7E-308… …1.7E+308	15 значащих цифр
long double	80	3.4E-4932… …1.1E+4932	19 значащих цифр

Из базовых типов с помощью операций *** & [ ] ( )** можно конструировать множество производных типов, например:
long int X[5]; пять объектов типа **long int** - X[0], X[1], X[2], X[3], X[4].
int f1 (void); функция не требует аргументов и, выполнив операцию, возвращает значение типа **int.**
void f2 (double); функция принимает аргумент **double** и не возвращает значений, то есть делает что-то, например очищает экран.
char*ptr; определяет указатель на объекты типа **char.**
int & (long); ссылка на функцию, возвращающую значение **int** и принимающую параметр **long.**
union NM {int b; char[c];}; объединение **NM** – объединяет целую переменную **b** и два элемента символьного массива **c[0]** и **c[1].**
struct ST {int a; char b; float c;} определяет тип структуры ST с тремя компонентами разных типов: a – целая, b – символьная, c – вещественная.

class A{int V; float G (char); }; тип A – класс, компонентами которого служат целая переменная V и вещественная функция G с символьным аргументом.

Возможные производные типы принято разделять на скалярные, агрегатные и функции. К скалярным относятся арифметические типы, перечислимые типы, указатели и ссылки, а к агрегатным – массивы, структуры, объединения и классы. Пример программы работы с переменными выглядит так:

```
// Вычисление площади поверхности параллелепипеда
# include <stdio.h >
# include <conio.h >
void main ( )
{       float l,w, h; //длина, ширина и высота параллелепипеда
        float  s;   // площадь поверхности параллелепипеда
        printf (“\nВычисление площади поверхности”) ;
        printf (“параллелепипеда\n”) ;
        printf (“Введите исходные данные: \ n”) ;
        printf (“Длина (см)   ->”) ;
        scanf (“%f ”,&l);
        printf (“Ширина (см)   ->”);
        scanf (“%f ”,&w);
        printf (“Высота (см)   ->”)
        scanf (“%f ”,&h);
        s = (l*w + l*h  + w*h)*2;
        printf (“Площадь поверхности:  %6.2f кв.см\ n”,s) ;
        printf (“\ n\ nДля завершения нажмите <Enter>”);
        getch ( );
}
```

2.2. Объекты и их атрибуты

Объект – это некоторая область памяти, а **переменная** – поименованная часть объекта, занимающая указанный участок памяти. При определении значения переменной во время компиляции или во время операции присваивания в соответствующей ей области памяти помещается некоторый код. Операция присваивания в леводопустимой части выражения содержит имя переменной (*l*-значение), а в правой – некоторое выражение, например F=73. Леводопустимое выражение в общем случае определяет

ссылку на какой-либо объект. Частным случаем ссылки служит переменная.

Кроме *l*-выражения для объекта задается тип, который:

- определяет нужное для объекта значение памяти;
- задает совокупность операций, допустимых для объекта;
- интерпретирует двоичные коды при обращении к объекту;
- используется для контроля типов при присваивании.

К леводопустимым выражениям относятся имена переменных, принадлежащих массивам; скалярных и символьных переменных; имена указателей; уточненные имена компонентов структур, объединений, классов; ссылки на объекты; выражения с операцией разыменования *; вызовы функций, возвращающих ссылки на объекты.

В левой части операторов присваивания нельзя использовать имена функций, массивов, констант и вызовы функции не возвращающей ссылки.

Для объектов кроме типов задается класс памяти, область действия, видимость объекта, продолжительность существования объектов и их имен, вид компоновки. Эти атрибуты взаимосвязаны и должны быть явно указаны или выбираются при описании конкретного объекта.

Типы определяют размер памяти, набор допустимых операций и правила интерпретации двоичных данных, как показано в таблице. Класс памяти определяет размещение объекта в памяти и продолжительность его существования. Для задания класса памяти при описании объекта используют спецификаторы:

auto (автоматически выделяемая, локальная память). Задается при определении объектов блока, например в теле функции. Этим объектам память выделяется при входе в блок и освобождается при выходе из него. Вне блока объекты класса **auto** не существуют.

static . Объект с таким спецификатором будет существовать только в пределах файла, где он определен. Присваивается переменным и функциям.

register аналогичен **auto,** но для размещения используются регистры, а не участки основной памяти. Если регистры заняты, то компилятор обрабатывает их как объекты автоматической памяти.

extern (тип компоновки и продолжительность существования). Объект класса extern глобальный и доступен во всех модулях программы. Приписывается переменным или функциям.

Область действия идентификатора зависит от того, где и как описаны идентификаторы. Если в блоке, то от точки описания до конца блока. Идентификаторы делятся на 4 группы: уникальные имена меток, используемые в операторе **goto;** уникальные имена структур, объединений, классов и перечислимых данных, уникальные имена переменных и функций;

названия типов, вводимых служебным словом **typedef.** Имена компонентов в разных структурах, объединениях и классах могут повторяться.

Вычислим с помощью функции **fact( )** факториал.

```
//Расчет факториала 5!
#include<conio.h>
#include<iostream.h>
long fact (int Z)
{
   long m = 1;
   if ( Z < 0 ) return 0;
   for (int i = 1; i<Z; m = ++ i * m);
   return m;
}
      main ( )
      {
          clrscr();
          int j = 1, K = 5;
          long fact ( int K = 0 ); //  Прототип функции
           for (; j <= K; j ++)
          cout << "\n arg = " <<j << " arg ! = " << fact (j);
                getch ();
      }
```

Результат:
```
arg =1   arg! = 1
arg=2    arg! = 2
arg=3    arg! = 6
arg=4    arg! = 24
arg=5    arg! = 120
```

Видна независимость меток (идентификаторов), так имя К в прототипе функции и К переменная имеют разные сферы действия и полностью независимы.

Видимость объекта связана с возможными повторными определениями идентификатора внутри вложенных функций. Область действия может превышать видимость, но не наоборот. Например:

```
//Переопределение внешнего имени внутри блока
#include <conio.h>
#include<iostream.h>
void main()
{
  clrscr();
  int k = 0,j = 15 ;
      { cout << "\nВнешняя для блока переменная k = “<< k;
```

```
        char k = 'A'; //Определена внутренняя переменная;
        cout << "\n Внутренняя переменная k = " << k;
        cout << "\nВидимая в блоке переменная j = " <<j;
        j=30; //Изменили значение внешней переменной ;
        } //Конец блока;
    cout << "\nВне блока: k = "<< k  << ", j = " << j;
    getch ();
}
```

Результат выполнения:

```
Внешняя для блока переменная k = 0
Внутренняя переменная k = A
Видимая в блоке переменная j =15
Вне блока: k = 0, j = 30
```

Внутри блока сохраняется сфера действия внешних для блока имен до их повторного описания (переменная k). Определение объекта внутри блока действует до выхода из блока и изменяет видимость объекта с тем же именем – объект невидим (Видим внутри k = A, а остается вне блока как и было k = 0). Внутри блока видимы определенные вне блока объекты (переменная j =15), после выхода из блока видим их переопределенными внутри блока (j = 30).

Объект, определенный в программе позже своего первого использования, должен быть описан в той функции, где он используется с атрибутом **extern.**

Есть три вида продолжительности существования объектов в памяти: статическая, локальная и динамическая. Статическая выделяется вначале программы и существует до конца. Её имеют все функции и файлы. Остальным придается спецификаторами класса памяти **static** и **extern.** При статической продолжительности объект может быть не глобальным.

При локальной продолжительности переменные называются автоматическими и создаются при входе в блок или функцию, где они описаны и уничтожаются при выходе. Объекты с локальной продолжительностью олжны быть явно инициализированы, иначе их начальное значение не предсказуемо. Для задания локальной продолжительности используют спецификатор памяти **auto**.

Объекты с динамической продолжительностью получают память и уничтожаются с помощью явных операторов в процессе выполнения программы. Для создания используют операции **new** или функции **malloc ( )**, а для уничтожения операция **delete** или функция **free ( )**. Прототипы функций включены в заголовочный файл **alloc.h**.

```
//Динамическое выделение памяти
#include <conio.h>
```

```
        #include<iostream.h>
        #include <alloc.h>
        void main()
{
   clrscr();
   int*t;
   int*m=(int*)malloc(sizeof(int));
   *m=10;
   t=m;
   m=(int*)malloc(sizeof(int));
   *m=20;
   cout << "\n Второе значение *m="<<*m;
   free(m);
   cout <<"\nПервое значение *m=" <<*t;
   free (t);
   getch();
}
```

Результат: Второе значение *m=20
Первое значение *m=10

Тип компоновки, или связывания определяет соответствие идентификатора конкретному объекту в программе, текст которой размещен в нескольких файлах (модулях). Тип компоновки устанавливается компилятором при определении объектов и описании имен по контексту, месту объектов, наличию спецификаторов класса памяти **static** и **extern**. Для параметров функций и имен объектов, локализованных в блоке без спецификатора **extern,** тип компоновки не существует.

2.3. Определения и описания

Аттрибуты объектов приписываются ему с помощью определений и описаний. **Определения** устанавливают аттрибуты, резервируют для объектов память и связывают с объектом идентификатор, то есть имя объекта. **Описания** поясняют идентификаторы программы компилятору. Описаний может быть много, но определение только одно.

Определение переменных содержит указание типа и имеет следующий формат:

s m тип имя1 иниц.1, имя2 иниц.2, …;

где **s** – спецификатор класса памяти (**auto, static, register,typedef**),

m – модификатор **const** или **volatile**,

тип – один из основных типов (**char, enum, int, float, double и др**.),

имя – идентификатор,

иниц. – необязательный инициализатор, определяющий начальное значение объекта. Синтаксис **иниц.** переменной означает инициализирующее выражение. Чаще инициализирующим выражением является константа. Скобочная форма инициализации разрешена только внутри функции, т.е. для локальных объектов.

Описание может быть определением, если: описывает переменную, содержит инициализатор, описывает класс, описывает структуру или объединение, полностью описывает функцию. Описание не может быть определением, если: описывает прототип функции, имеет спецификатор **extern,** описывает имя класса, описывает имя типа вводимого пользователем (**typedef).**

Примеры описаний:

extern int q; //Внешняя переменная

float fg (int, double); // Прототип функции

struct st; // Имя структуры

typedef unsigned char simbol; // Новый тип **simbol**

Примеры определений:

float dim=12.0; // Инициализированная переменная

double euler (2.7182); // Инициализированная переменная

const float pi=3.14159; // Константа

float x2 (float x) {returnx*x;}; // Функция

struct {char a; int b;} st; // Структура

enum {1,2,3,4,5}; // Перечисление

Пример:

```
// Определения и описания переменных
#include <conio.h>
#include<iostream.h>
float pi = 3.1415; // Определение с явной инициализацией
int s0; // Определение с инициализацией по умолчанию
void main()
{
    clrscr();
    extern int s0;// Описание s0
    extern char s1; // Описание s1
    int s2(4); // Описание s2 с явной инициализацией
    cout <<"\ s1 = " << s1;
    cout <<"\ s2 = " << s2;
    cout <<"\ pi = " << pi;
    cout <<"\ s0 = " << s0;
```

```
    getch();
}
```

char s1='7'; // Явная инициализация

Результат выполнения: s1=7; s2=4; pi=3.1415; s0=0

Значение функции s2 задано в скобках. Внешние переменные (вне текста функции main( )) так инициализировать нельзя, так как компилятор воспримет это как ошибку. Внешние переменные, определенные до текста функции, доступны в ней без дополнительных описаний, поэтому строка **extern int s0;** в данной программе лишняя, а описание **extern char s1**; необходимо.

Спецификатор (определитель класса) памяти **auto** встречается редко, так как его запрещено использовать во внешних описаниях, а внутри блока локальные объекты являются объектами автоматической памяти. **Register** тоже запрещен во внешних описаниях, но внутри блоков или функций его использование обосновано. Спецификатор **typedef** нельзя употреблять в определениях, но можно в описаниях новых наименований типов.

Модификаторы **const** и **volatile** сообщают компилятору об изменчивости или постоянстве определяемого объекта. Если переменная описана как **const**, то её нельзя изменять во время выполнения программы. Её инициализируют при определении. Объекту с модификатором **const** запрещены также операции (++) и (--). Указатель с модификатором **const** нельзя изменить.

Модификатор **volatile** имеет значение при работе с классами. Диалекты С++ имеют модификаторы: **cdecl, pascal –** для функций и переменных; **near, far** – для указателей, функций, переменных; **interrupt**- для функций и обработки прерываний; **huge –**для указателей и функций.

2.4. Выражения и преобразования типов

Выражение – это последовательность операндов, разделителей и знаков, задающая вычисление. Порядок выполнения задается приоритетом (рангом) операции и правилами их группирования. В С++ можно расширить действие стандартных операций (перегрузка overload), придавая им нестандартный смысл.

Первичные выражения содержат: константы, (выражения), имена, **::** идентификатор (здесь :: – указатель области видимости), :: имя функции операции, this. Константы (целые, вещественные, строковые, перечислимые) мы рассматривали ранее, теперь рассмотрим имена.

К именам относятся:

- **идентификатор**, если он введен с помощью определения (часто применяют идентификатор как имя переменной);
- **имя функции-операции,** если вводится в связи с расширением действия операции;
- **имя класса**, обозначая обращение к компоненту класса;
- **имя функции приведения**, когда преобразующие функции являются компонентами классов;
- **квалифицированное (уточненное) имя**, связанное с именем класса.

Без понятия класс (которое изучим позднее) можно в качестве имен использовать только спецификаторы.

Выражение, заключенное в скобки внутри основного выражения, – это рекурсия, изменяющая порядок операций. Кроме первичных выражений вводятся постфиксные (позднее прикрепленные, т.е. позднее выполняемые) выражения, например в m++. Сначала определяется m, а затем оно увеличивается на единицу.

Преобразование типов

Основные типы **char** (символьный), **short** (короткий целый), **int** (целый), **long** (длинный целый), **float** (вещественный), **double** (вещественный с удвоенной точностью), **void** (отсутствие значения) и их производные иногда требуют преобразований в ходе выполнения программы. К этому следует относиться с большой осторожностью, так как возможна потеря данных, например, при преобразовании из float в int из-за выхода за диапазон возможных значений.

Хорошо если в одном выражении оперируем с данными одного типа. Если же они разного типа, то это приводит к ошибкам. Например, в программе определили типы данных: **float** C**, int** X = 21, **int** Y = 4, а при решении уравнения C=X/Y компилятор, оперируя с целыми числами, выдаст в результате целое число, и получим 21/4 =5. Для правильного решения нужно изменить тип и записать его в программе путем явного приведения типа к требуемому

C= (float)(X/Y),

и результат будет C=5.25.

Преобразования, гарантирующие сохранение значимости имеют вид:

signed char → short → int → long,

float → double → long double,

unsigned char → unsigned short → unsigned int → unsigned long.

Пример программы преобразования числа в денежный формат

```
# include <stdio.h >
# include <conio.h >
void main ( )
{
```

```
        float  f;   // дробное число
        int r       // целая часть числа (рубли)
        int k       // дробная часть числа (копейки)
        printf (“\ n Преобразование числа в денежный формат\ n”);
        printf (“Введите дробное число ->”);
        scanf (“%f ”,& f);
        r =   (int)f;
        k = f * 100 – r*100;
        printf (“ %6.2f руб. – это %i руб. %i коп.\ n”,  f,  r,  k) ;
        printf (“\ nДля завершения нажмите <Enter>”);
        getch ( ) ;
}
```

Контрольные вопросы

1. Перечислите основные типы переменных в С++.
2. Что такое объект и переменная в С++?
3. Какие типы переменных называются агрегатными?
4. Для чего в С++ используются спецификаторы auto, static, register, extern?
5. Объясните понятия область видимости и область действия.
6.Перечислите виды продолжительности существования объектов в памяти?
7. Объясните назначение определений и описаний в С++.
8. Какой спецификатор используется для пользовательского типа переменных?
9. Для чего используется модификатор const?
10. Приведите пример преобразования целого типа в вещественный.

Глава 3. ОПЕРАТОРЫ С++

3.1. Последовательно выполняемые операторы языка С++

Каждый оператор заканчивается и идентифицируется разделителем – точкой с запятой. Любое выражение, после которого стоит точка с запятой, воспринимается компилятором как отдельный оператор. Исключением являются выражения, входящие в заголовок цикла **for.**

Часто оператор служит для вызова функции, не возвращающей никакого значения. Например, определим коды различных символов клавиатуры:

```
// Обращение к функции как оператор-выражение
#include <conio.h>
#include <iostream.h>
void cod (char c)
{
cout << "\n"<< c << " = " << (unsigned int) c;
}
void main ( )
{
  clrscr( );
  void cod (char); //Прототип функции
  cod ('g');//Оператор-выражение
  cod ('K');// Оператор-выражение
  getch ( );
}
```

Решение: g=103, K=75 (это, например, коды ASCII символов g и K в десятичной системе).

Часто оператор-выражение является выражением присваивания. Оператор присваивания является частным случаем оператора-выражения. Пустой оператор просто точка с запятой, перед которой нет никакого выражения, не предусматривает никакого действия и нужен только для синтаксиса. Пустой оператор применяют в качестве тела цикла, когда все циклические действия определены в его заголовке, например при вычислении факториала 5!

```
for (int i = 0, p = 1; i < 5; i++, p*= i);.
```

Перед каждым оператором можно ставить отделяемую от него двоеточием метку (идентификатор, например UR) с целью локализации метки в сфере действия функции

```
UR: x= 3+7*5;.
```

Метки можно ставить в любом месте программы, где разрешает синтаксис. Заключенная в фигурные скобки последовательность операторов называется **составным оператором**. Если в этих же скобках есть описания и определения, то составной оператор превращается в **блок** с локализованными в нем определенными объектами. Синтаксически и блок, и составной оператор являются отдельными операторами, но не должны заканчиваться точкой с запятой. Для них ограничитель – фигурная скобка. А внутри блока обязательны точка с запятой после каждого оператора.

{int d; char g='3';} // это блок.

{func(x-6.0,34); s=7*6-2;} // это составной оператор

3.2. Операторы выбора

Это два оператора: условный оператор **(if…else –** если … иначе**)** и переключатель (**switch –** переключатель**),** служащие для выбора пути выполнения программы. Синтаксис условного оператора прост:

if (выражение) оператор 1; **else** оператор 2; .

Выражение должно быть скалярным и может иметь арифметический тип или тип указателя. Если оно не равно нулю, то условие истинно и выполняется оператор 1, иначе выполняется оператор 2. В качестве операторов нельзя использовать описания и определения, но могут быть составные операторы и блоки:

if (X>0) {X=-X; f(X*^);}
else {int i = 2; X*=1;f(X);}

При работе с блоками (то есть с составными операторами с описаниями и определениями) следует помнить, что объекты в них локализуются и вне их не существуют, поэтому такая запись, как

if (j>0) {int i; i=6*j; }
else i = -j

будет неверной, так как переменная i локализована в блоке и не существует вне его.

Допустима сокращенная форма условного оператора без **else** и оператора 2. В этом случае при равенстве условия нулю никакое действие не выполняется

if (a<0) a=-a; .

Операторы 1 и 2 могут быть условными, что позволяет организовывать цепочку проверок условий. Синтаксис предполагает, что каждое **else** соответствует ближайшему к нему предшествующему **if.** Например, выделяем отступами:

```
int max 3 (int x, int y, int z)
{
        if (x<y) if(y<z) return z;
        else return y;
        else if (x<z) return z;
        else return x;
}
```

Пример использования условных операторов:

```
// Контроль веса
# include <stdio.h >
# include <conio.h >
void main ( )
{
        float  w;       // вес
        float   h;      // рост
        float   opt;    // оптимальный вес
        float  d;       // отклонение от оптимального веса
    printf (“\ Введите в одной строке, через пробел,\ n ”);
    printf (“рост (см) и вес (кг), затем нажмите <Enter>”);
    printf (“->”);
        scanf (“%f %f”,  &h,  &w );
        opt = h - 100
        if  (w = = opt)
    printf (“Ваш вес оптимален!”);
        else
          if (w < opt)
       {
       d = opt - w
       printf (“Вам надо поправиться на %2.2f кг.\n”, d);
        }
        else
          {
            d = w - opt ;
            printf (“Вам надо похудеть  на %2.2f кг.\n”, d);
           }
    printf (“\ nДля завершения нажмите <Enter>”);
    getch ( ) ;
}
```

Переключатель является удобным средством для мультиветвления и реализуется следующим синтаксисом:

switch (переключающее выражение)

```
{
case константное выражение1: операторы 1;
case константное выражение 2 : операторы 2;
…
case константное выражение n : операторы n;
        default : операторы;
}
```

Управляющая конструкция **switch** передает управление тому из помеченных с помощью **case** (вариант) операторов, для которого константное выражение совпадает со значением переключающего выражения. В одном переключателе все константные выражения должны быть разными, но иметь один тип. Если значение переключающего выражения не совпало ни с одним из константных, то выполняется переход к оператору с меткой **default** (по умолчанию)**.** При отсутствии метки **default** и при несовпадении ни с одним из константных выражений в переключателе не выполняется ни один из операторов. Выход из переключателя с помощью оператора **break** (завершить).

```
// Выводит название дня недели
# include <stdio.h >
# include <conio.h >
void main ( )
{   int nd;  // номер дня недели
    puts (“\nВведите номер дня недели (1..7”);
    printf (“->”);
        scanf (“%i,  &nd);
        switch (nd)
    {   case  1:  puts (“Понедельник”); break;
        case  2:  puts (“Вторник”); break;
        case  3:  puts (“Среда”); break;
        case  4:  puts (“Четверг”); break;
        case  5:  puts (“Пятница”); break;
        case  6:  puts (“Суббота”); break;
        case  7:  puts (“Воскресенье”); break;
        default: puts (“Число должно быть в диапазоне 1...7 ”);
    getch ( ) ;
    }
}
```

Без фигурных скобок смысл переключателя теряется, так как со служебным словом **case** тогда можно использовать только один оператор. В переключателе могут находиться описания и определения объектов, то есть он может быть блоком.

3.3. Операторы цикла

Операторы цикла задают многократное исполнение операторов тела цикла. Определены три разновидности таких операторов:

- цикл с предусловием (**while** – пока):
  while (выражение-условие)
  тело цикла;
- цикл с постусловием:
  do (выполнить)
  тело цикла
  while (выражение - условие);
- итерационный цикл:
  for (для)(инициализация цикла;
  выражение - условие;
  список выражений)
  тело цикла.

Тело цикла не может быть описанием или определением. Это либо отдельный (даже пустой) оператор, завершающийся точкой с запятой, либо составной оператор, либо блок в фигурных скобках. Выражение-условие во всех операторах есть скалярное выражение (чаще всего отношение или арифметическое выражение). Оно определяет условие продолжения выполнения итераций, если не равно нулю. Инициализация в цикле **for** всегда завершается точкой с запятой, отделяясь от выражения-условия, которое также завершается точкой с запятой. Список выражений в цикле **for** – это последовательность скалярных выражений, разделенных запятыми.

Прекращение выполнения цикла возможно в двух случаях:

- Нулевое значение проверяемого выражения-условия;
- Выполнение в теле цикла оператора передачи управления **(break, goto, return)** за пределы цикла.

Оператор **while** (повторять **пока** истинно условие) называется циклом с предусловием и при входе в цикл вычисляется выражение-условие. Если оно отлично от нуля, то тело цикла выполняется. Затем вычисление условия и выполнение цикла повторяются, пока значение выражения-условия не станет равным нулю.

```
int i = 0; // Счетчик
int s = 0; // Будущая сумма
while (i < K)
s += ++i * i; // Цикл вычисления суммы квадратов чисел, т.е. s=s+i*(i+1).
```

Если сравнение с пустым указателем (нулем), то три проверки эквивалентны:

while (po != NULE)…
while (po) …
while (po != 0) …

Операторы тела цикла должны воздействовать на выражение-условие (обычно с помощью операций ++ или --), иначе возникает зацикливание.

Оператор **do** (выполнить) называют с постусловием. При входе в цикл **do** обязательно выполняется тело цикла, затем вычисляется выражение-условие и, если оно не равно нулю, снова вычисляется тело цикла. Такой цикл удобен, когда обработку нужно заканчивать после появления концевого признака. Например, переписываем заданную указателем **star** строку в другую, наперед заданную (**nov**)

```
void copy (char*star, char*nov)
{
do * nov = * star++;
while(* nov++);
}
```

Выражение-условие непременно должно меняться, иначе возникает зацикливание.

В цикле **for** инициализация цикла – это последовательность определений (описаний) и выражений, разделяемых запятыми. Все выражения вычисляются только один раз при входе в цикл. Чаще всего устанавливаются начальные значения счетчиков и параметров цикла. Если выражение-условие равно нулю, то выполнение цикла прекращается. Если нет выражения-условия, то следующая за ним точка с запятой остается и предполагается, что выражение-условие истинно. И при отсутствии инициализации цикла соответствующая точка с запятой остается, а выражения из списка выражений вычисляются на каждой итерации после выполнения операторов тела цикла и до следующей проверки выражения-условия. Тело цикла может быть блоком, отдельным оператором, составным или пустым оператором.

for (оператор 1; выражение-условие; оператор 2) оператор 3 – тело цикла;

У цикла **for** множество применений. Например, различные варианты решения задачи определения суммы квадратов первых К членов натурального ряда:

```
for (int i = 1; s = 0; i <=K; i++) s += i * i;
for (int i = 0; s = 0; i <= K; s += ++i * i);
for (int i = 0; s = 0; i <=K; ) s += ++i * i;
for (int i = 0 ; s = 0; i <=K;)
    { int j; j = ++i; s += j * j;}
```

Во втором операторе тело цикла – пустой оператор, в третьем нет списка выражений. Очередность выполнения цикла **for** следующая:

- определяются объекты и вычисляются выражения, включенные в инициализацию цикла;
- вычисляется выражение-условие;
- если оно отлично от нуля, то выполняются операторы тела цикла;
- затем вычисляются выражения из списка выражений, и вновь вычисляется выражение-условие;
- цепочка действий повторяется, пока изменение выражения-условия не приведет к нулевому (ложному) значению.

Широко применяется вложение любых циклов в любые циклы. При этом в инициализации внутреннего цикла **for** можно использовать переменную с тем же именем, что и во внешнем цикле. До внутреннего цикла действует инициализация этой переменной внешнего цикла, а после инициализации внутреннего цикла определяется другое значение переменной и оно остается до конца тела внешнего цикла. Пример использования цикла для вычисления числа “ПИ” .

```
// Вычисление числа “ПИ” с использованием
// свойства ряда 1  -1/3 + 1/5 – 17 + …
# include <stdio.h >
# include <conio.h >
void main ( )
{
        float  x;        //  член ряда
        int n;           //  количество суммируемых членов
        float summ;   // частичная сумма
        int  i;          // счетчик циклов

        // при суммировании достаточно большого  (~1000) количества
        // элементов ряда, значение суммы стремится к “ПИ”/4
        printf (“Вычисление суммы ряда 1 -1/3 + 1/5 – 1/7 + …\n ”);
printf (“Введите кол-во суммируемых членов ряда->”);
        scanf (“%i,  &n);
summ  = 0
for (i = 1; i <= n; i ++)
{ x = (float) 1/ (2*i – 1);
if ((i % 2) = = 0) x = -1 * x;
        sum += x;}
        printf (“сумма ряда:   %2.6f\n”, summ );
        printf (“ вычисляемое значение”);
        printf (“числа ПИ =  %2.6f\n”, summ * 4);
```

```
printf (“\ nДля завершения нажмите <Enter>”);
getch ( ) ;
}
```

3.4. Операторы передачи управления

К ним относят четыре оператора:

- безусловного перехода **goto,**
- возврата из функции **return,**
- выхода из цикла или переключателя **break;**
- перехода к следующей итерации цикла **continue.**

Рассмотрим их поочередно. Оператор безусловного перехода имеет вид:**goto идентификатор;** и передача управления разрешена на любой помеченный оператор в теле функции. Однако есть важное ограничение по применению. Его суть в том, что запрещена передача управления через описания, содержащие инициализацию объектов, кроме вложенных блоков, которые можно обходить целиком. Например,

```
goto A; // ошибочный переход через инициализируемую k
int k=12;
goto A; // Верный переход, минуя блок
{int x=2; x=k*x+x;}
A: cout << “\n  x = ” << x;
```

Все операторы блока достижимы для принятия управления, но то же самое правило – нельзя передавать в блок управление, обходя инициализацию. Поэтому оператор **goto** применяют нечасто, обычно для выхода сразу из двух циклов или когда из разных мест необходимо переходить к одному участку программы. При этом придерживаются правил:

- не входить в блок извне ;
- не передавать управление операторам, размещенным после служебных слов **if** и **else;**
- не входить извне внутрь переключателя **switch;**
- не передавать управление внутрь цикла.

Оператор возврата из функции имеет вид: **return выражение;** или просто **return;**

Выражение, если оно есть, должно быть скалярным. Например, следующая функция вычисляет и возвращает площадь квадрата своего аргумента: **float squ (float x) {return x*x;}**

Если возвращаемое значение функции типа **void,** то в операторе **return** нет выражения, например, выводит на экран и не возвращает в точку вызова никакого значения.

Оператор принудительного выхода из цикла или переключателя **break** осуществляет передачу управления (переход) следующему за циклом или переключателем оператору. В отличие от **goto** оператор, которому передается управление, не помечается – он просто следующий после цикла или переключателя. Оператор **break** используют только в переключателях и в циклах, когда условие продолжения итераций нужно проверять в середине тела цикла. Он позволяет выйти только из самых внутренних циклов или переключателей. Например:

```
while ( i < j)
 { i++;
   if (i ==j) break;
   j--;
  }
```

С помощью оператора **break** и переключателей легко реализуется разветвление, например, печать любой, но только одной цифры:

```
void main( )
{
int ic;
cout << “\n Введите восьмеричную цифру: ”;
cin >>ic;
cout << “\n” <<ic;
switch (ic)
{ case 0: cout << “нуль”; break;
case 1: cout << “один”; break;
case 2: cout << “два”; break;
default: cout << “Это не восьмеричная цифра!”;
}
cout << “\n Конец программы”;
}
```

Будет напечатана только введенная цифра, а без операторов **break** выполнялись бы все пункты, начиная с введенного значения.

Циклы и переключатели могут быть многократно вложены, но оператор **break** обеспечивает выход только из самого внутреннего цикла или переключателя. Например, подсчитаем сколько единиц **k1** и нулей **k0** в символьном массиве

```
void main (void)
{
 char c[ ] = “ABC100111”;
```

```
    int k0 = 0, k1 = 0;
    for (int i = 0; c [ ] != '\0'; i++)
    switch (c [ i ] )
    { case '0': k0++; break;
    case '1': k1++; break;
    default: break;
    }
    cout << "\n  В строке " << k0 << "  нуля," << k1 << "  единицы.";
}
```

Результат: В строке 2 нуля, 4 единицы.

Оператор **break** тут передает управление из переключателя, но не за пределы цикла, и цикл продолжается до завершения. При многократном вложении циклов оператор **break** не может передать управление из самого внутреннего в самый внешний, и тогда пользуются оператором **goto.**

Оператор **continue** используют только в операторах цикла для завершения текущей итерации и начала проверки условия продолжения цикла, то есть условий начала следующей итерации. В тело операторов цикла помещают пустой оператор с меткой **contin:** Если в теле цикла есть оператор **continue** и он выполняется, то его действие эквивалентно действию оператора безусловного перехода на метку **contin:** Например, для трех операторов цикла это выглядит так:

```
while (foo)          do                    for ( ; foo; )
{ ….                 { …                    { …
contin:               contin:                 contin:
}                    } while (foo)           }
```

Здесь точки – операторы тела цикла.

Контрольные вопросы

1. Дайте определение составного оператора и блока операторов.
2. Приведите пример условных операторов if…else.
3. Приведите пример оператора выбора switch.
4. Назовите три разновидности операторов цикла в C++.
5. Приведите пример использования оператора while в C++.
6. Приведите пример использования оператора for в C++.
7. Назовите все операторы передачи управления в C++.
8. В каких случаях используется оператор передачи управления goto?
9. Что является аргументом оператора передачи управления return?
10. Приведите пример использования операторов continue и break.

Глава 4. АДРЕСА, УКАЗАТЕЛИ, МАССИВЫ, ПАМЯТЬ

4.1. Указатели и адреса объектов

Указатели являются специальными объектами в С++. Например, указатели new и delete для динамического управления памятью. **Указатели** делятся на две категории – указатели **на объекты** и указатели **на функции**. Это связано с различием свойств и правил применения.

Указатели на объекты имеют вид:

type * имя указателя;

где type – обозначение типа; * – операция обращения по адресу, операндом которой является указатель. Например: float * XX.

Признаком указателя является символ * (операция разименования, то есть получения значения через указатель). Этот символ помещают перед именем и если требуется поместить несколько указателей на объекты одного типа, то символ * помещают перед каждым именем. Например, int *ff, *db, *sa, k; вводит три указателя на объекты и на одну переменную k целого типа. Имя переменной уже является ссылкой на какой-то объект и операция * в указателе на переменную не требуется.

При определении указателя целесообразно выполнять его инициализацию одним из двух способов:

type * имя _указателя = инициализирующее выражение

либо

type * имя указателя (инициализирующее выражение)

Инициализирующим выражением может быть константное выражение, представляющее собой:

- явно заданный адрес участка памяти;
- указатель, уже имеющий значение;
- выражение, позволяющее получить адрес объекта с помощью операции **&.**

Если инициализирующее выражение равно нулю, то следует обращаться к значению пустого указателя, имеющего обозначение NULL, например, char*pt(NULL); // Нулевой указатель на объект типа char

Примеры определения указателей:

char ds= ’g’; // Символьная переменная ds типа char, инициализированная значением символьной константы ‘g’

char * csc = &ds; // Инициализированный указатель csc на объект типа char (здесь & операция получения адреса).

Доступ к значению переменной ds теперь возможен как по её имени, так и с помощью адреса, являющегося значением указателя csc этой пере-

менной. Присвоив указателю адрес конкретного участка памяти, можно менять содержимое этого участка, например, csc = ‘b’ сделают значение переменной ds равное b. Необходимо только помнить о преобразовании вводимого значения к имеющемуся уже типу указателя, например,

p = new char; // Выделили память для переменной типа char и связали указатель p с этим участком памяти

p =(char*)83; // Преобразовали числовое значение 83 к типу char и связали указатель p с участком памяти, выделенным кодом ASCII символу s.

Заметим, что здесь требуется дополнительно связать числовое значение и тип указателя (char *).

После таких операторов можно использовать p для записи в память нужных символьных значений p = ‘f’; или cin >> p;. Оператор cin >>*p; преобразует набираемый пользователем на клавиатуре символ в символьное значение и присваивает это значение указателю p.

Операцию разименования * вместе с указателем p можно определить как получение значения, размещенного по адресу, равному значению указателя. Если *p находятся слева от знака операции присваивания или в операторе ввода данных, то это означает – разместить значение по адресу, равному значению указателя.

Указатель и его значения могут быть объявлены константами с помощью модификатора **const:**

type * const имя указателя инициализатор;

Так как значение указателя-константы изменить нельзя, то имя указателя константы можно считать наименованием фиксированного участка основной памяти. Но с помощью операции разименования само содержимое участка памяти, на которую смотрит указатель-константа можно менять.

Например: const int one = 1; // определение константы

int const *point = &1; // Указатель на константу 1

Недопустимы операторы вида

*point = 7; // Неверная попытка изменить константу

Но операторы вида

point =&cc; или point = 7;

вполне допустимы, так как разрывают связь указателя **point** с константой 1, не меняя её изображения в конкретном участке памяти.

Можно определить неизменный (постоянный) указатель на константу, например:

const float pi = 3.141593; // Поименование константы

float const * const point = π // Указатель point на константу pi.

Здесь нельзя изменить значение константы, обращаясь к ней с помощью выражения ***point**, и нельзя изменить значение указателя **point**, т.к. он всегда смотрит на константу 3.141593.

Работа с указателями постоянно требует применения операции **&** - **получение адреса объекта**, но для неё есть ограничения:

- нельзя определять адрес неименованной константы, например, недопустимы выражения &3.141593 или &'?';
- нельзя определять адрес значения, получаемого при вычислении скалярных выражений, т.е. недопустимы записи вида &(b-d)*f !=72 или &(a+y*x);
- нельзя определять адрес переменной, относящейся к классу памяти register (автоматически выделяемая регистровая память), то есть ошибочна запись int register mu = 25;// Определили mu

int *prt = μ// Попытка дать адрес указателю prt

Таким образом можно сделать вывод, что операция получения адреса **&** применима только к объектам, имеющим имя и размещенным в памяти. Однако допустимо получать адрес именованной константы, например:

```
const float Eu = 2.718282;
float *pEu = (float *) &Eu;
```

При этом необходимо применять явное преобразование типов, т.к. &Eu имеет тип const float *, а не float *.

4.2. Типы указателей и операции над ними

В C++, в отличие от других языков программирования, каждый указатель связан с некоторым типом, как основным (char, int, float, long, double, short, unsigned, signed, void), так и производным, например:

```
unsigned long int *gfh = NULL; // gfh – указатель
long double hg = 1.7; //  hg – переменная
```

Если получение адреса **&** дает однозначный результат, зависящий от размещения объекта в памяти, то операция разыменования ***указатель** зависит и от типа объекта, т.к. при этом требуется информация о размещении и о размерах участка памяти, который будет использоваться. Компилятор получает информацию о размерах участка памяти от типа указателя и при неверном указании типа объекта неизбежны ошибки.

Явное преобразование типов (например, long *gf = **(long *)**&K) при работе с разными указателями в одном выражении необходимо для всех указателей, кроме имеющих тип **void *,** для которого операция преобразо-

вания типов применяется по умолчанию. Тип **void** не имеет значения и указатель **void *** не требует сведений о размере соответствующего ему участка памяти.

Указатель void * как бы универсальный, но он требует связывания с объектами разных типов путем их приведения. Например:

```
// Приведение типов
# include <conio.h>
# include <iostream.h>
void main( )
   {
      clrscr( );
      void * vp;
      int i = 77;
      float Euler = 2.718282;
      vp =&i; // Настроились на -  int
      cout << "\n vp = " << vp << "\t *(int*) vp = " << * (int* ) vp;
      vp = & Euler; // Настроились на -  float
      cout << "\n vp = " <<vp << " *(float*) vp = "<< *(float*) vp ;
      getch ( );
   }
```

Решение: vp = 0xfff4 *(int*) vp = 77
vp = 0xfff0 *(float *) vp = 2.718282

Программы должны работать с максимальным количеством типов, поэтому указатели типа void* в языке С вообще называют родовыми. Разрешено неявное (умалчиваемое) преобразование значения любого неконстантного и не имеющего модификатора **volatile** (отмечающего, что в процессе выполнения программы значение объекта может изменяться между явными обращениями к нему) указателя к указателю типа **void*.** Например, неверна последовательность операторов void *vp; int *ip; ip = vp; так как к одному и тому же объекту будет доступ с помощью указателей разных типов.

Операции над указателями можно определить как:

операции разименования или доступа по адресу (*);
пребразование (приведение) типов;
присваивание;
получение (взятие) адреса (&);
сложение и вычитание (аддитивные операции);
инкремент или автоувеличение (++);
декремент или автоуменьшение (--);
операции отношений (сравнения).

Указатель как объект имеет адрес соответствующего участка памяти, значение которого доступно с помощью операции &, применяемой к указателю.

unsigned int *ee1 = NULL, *ee2;
ee2 = (unsigned int *) &ee1;

Здесь указатель ee1 получает значение 0 при инициализации, а указателю ee2 присваивается адрес указателя ee1. При этом необходимо явное преобразование типа в операторе присваивания, т.к. тип ee2 не определен.

Вычитание при аддитивных операциях применимо к указателям на объекты одного типа и к указателю и целочисленной константе. **Разность двух указателей дает в байтах значение разности длин двух участков памяти**, а не разность числовых значений этих указателей. Суммировать указатели в C++ нельзя.

Декремент (--) и инкремент (++), вычитая или прибавляя единичную константу, перемещает указатель к соседнему объекту с меньшим или большим адресом. Размещение операций – или ++ до либо после указателя говорит только о том, когда выполняется эта операция до или после использования указателя.

Так как указатель – это объект в памяти, то можно определять указатель на указатель сколько необходимо раз, но при этом помнить, что ассоциативность операции разыменования * справа налево и понимать смысл запутанных выражений с адресами и указателями.

4.3. Массивы и указатели

Массив – это структурированные данные одного типа с расположением элементов подряд друг за другом.

type имя массива [константное выражение];

Константное выражение (если оно присутствует) определяет размер массива, т.е. количество элементов в массиве. Иногда описывают массив без указания размера, но это в том случае, если он определен в другой части программы и ему выделена память и возможно присвоены начальные значения. Если массив статический или внешний, то ему по умолчанию (без вмешательства программиста) автоматически присваиваются нулевые значения. Например:

```
void f (void)
{       static float G[7]; // Внутренний статический массив
        long double C[8]; // Массив автоматической памяти
}
```

int H[9]; // Внешний массив

Массивы G[7] и H[9] будут инициализированы нулевыми значениями. Явная инициализация массива разрешена только при его определении, как с указанием числа элементов, так и без указания

Char CG [ ] = {‘A’, ‘B’, ‘C’};

Int [7] = {5,4,8,2,9 };

Char STR [] = “FGHDHGF”;

При отсутствии константного выражения в квадратных скобках обязательно указывать начальные значения в фигурных скобках в правой части. Если элементов в фигурных скобках меньше, чем число в квадратных, то остальные элементы либо не определены, либо они нулевые при внешнем или статическом массиве. В массиве STR элемент STR[7] равен ‘\0’, а всего в этом массиве 8 элементов.

Доступ к индексированным переменным выполняется по номеру индекса, начиная с нуля. Все элементы массива имеют одинаковый размер в байтах, а частное sizeof (имя массива)/sizeof(имя массива [0]) определяет количество элементов в массиве и тогда простой фрагмент программы дает значения всех элементов. Например:

```
Int m[] = {10,20,30,40};
For (int i = 0; i< sizeof(m)/sizeof(m[0]); i++}
Cout << “m[“ <<i<<”] = “<< m[i] << “   “;
```

Даст результат m[0] = 10 m[1] = 20 m[2] = 30 m[3] = 40.

Операндами для операции [ ] служат имя массива и индекс. Имя массива есть указатель-адрес начала массива. Индекс – это выражение целого типа, выражающее смещение от начала массива. Первый элемент всегда имеет индекс 0, так как индекс определяет не номер элемента, а его смещение относительно начала массива.

Обращение к элементу массива является постфиксным выражением вида IM[NI], где IM – указатель на массив, NI – индексированный элемент массива. Другой вид доступа к тому же элементу массива *(IM+NI), а так как от перестановки членов суммы ничего не меняется, то и *(NI+IM), а значит и NI[IM].

Если элемент массива имеет отрицательное значение, то это означает, что указатель показывает не на начало массива. Например:

```
char S[] = “COH”;
char * U = & A [2];
cout << “\n” << U[0]  << U[-1]  <<U[-2];
```

При этом на экран будет выведено слово НОС.

Так как имя массива является указателем константой, то его изменить нельзя. Поэтому доступ, например, ко второму члену массива float J []

с помощью выражения *(++J) будет ошибкой, а с помощью выражения *(J+1) будет верным.

При работе с элементами массива можно не использовать квадратные скобки. Например:

```
//Элементы массива
#include <conio.h>
#include<iostream.h>
void main()
{
        clrscr();
        char x[] = "VOLK";
        int i =0;
        while ( * (x + i) != '\0')
        cout << * (x+i++) << "\n";
        getch ();
}
```

В результате выполнения цикла получим слово VOLK, написанное в столбик, т.к. оператор цикла while выполняется, пока символ массива не будет равен ‘\0’. Тот же результат будет при заголовке цикла while (*(x+i)) либо при операторе cout << ‘\n’ << x[i++]; так как в цикле используется текущее значение i, которое при каждом вычислении x+i++ увеличивается на 1.

Инициализацию массивов можно выполнять списком, последовательно указывая значения каждого элемента массива. При этом в конце необходим символ ‘\0’, придающий одномерному массиву свойства строки, например:

```
char stroka [] = {‘D’, ‘G’, ‘E’, ‘V’, ‘\0’};
```

Совместно с массивами применяют и указатели, например:

```
long eva [] = {12, 23, 13, 40, 88};// Определили массив
long *kuku = eva; // Определили указатель и связали его с массивом
int *fif = new int[5]; // Определили указатель и выделили участок памяти
```

Здесь массив eva инициализирован списком начальных значений, а массив, связанный с указателем fif, получил нужную память, но его элементы не инициализированы.

В отличие от имени массива, указатель на массив не сообщит размер памяти элементов массива в байтах и операция sizeof возвратит длину только самого указателя в байтах. Если массив типа char, то последним элементом массива является символ ‘\0’, закрывающий строку.

К элементам массива, связанным с указателем, доступ либо с помощью операции [ ], либо с помощью операции разыменования *. В случае определения массива с помощью указателя этот указатель является пере-

менной и доступен изменениям. При определении указателя ему может быть присвоено значение другого указателя, уже связанного с массивом того же типа и доступ возможен уже с помощью двух разных имен. Однако это не присваивание одному массиву значений элементов другого массива, а только введение дополнительного имени указателя.

Чтобы в процессе выполнения программы изменить значения элементов массива, требуется обязательно выполнить операции присваивания. При этом длина заполняемого массива должна быть не меньше длины копируемого массива, а выход за границу индекса приводит к ошибке. Например, заменит содержимое массива строки **str** на строку **pstr** оператор

while (str++ = pstr++);

или его аналог

for (int i = 0; str [i] = pstr [i]; i++);

Вообще-то для копирования строк в C++ есть функция **strcpy( ),** прототип которой находится в заголовочном файле **<string.h>** и действие

char * strcpy (char * str2, char * str1);

копирует байты строки **str1** в строку **str2.** Это вызвано тем, что в C++ нет специального типа данных “строка” и каждая символьная строка представляется в памяти ЭВМ в виде одномерного массива типа **char.** Как правило строка используется для инициализации указателя типа **char *** или массива типа **char:**

char * pointer = “инициализирующая строка”;
char arra [ ] = “инициализирующая строка”;

В обоих случаях, если строка используется для инициализации указателя или массива, то адрес первого элемента строки становится значением указателя-переменной **pointer** или указателя-константы (имени массива) **arra.**

4.4. Многомерные и динамические массивы

Многомерный массив это массив из массивов вида

type имя массива [M1] [M2] … [MN],

где N – размерность массива; M1 – количество в массиве элементов размерности N-1 каждый и т.д. Например:

int ara [4] [3] [6] = {0,1,2,3,4,5,6,7};

Это трехмерный массив из 4 элементов, каждый из которых – двумерный массив с размерами 3 на 6. Начальные значения получили только первые 8 элементов массива (из 24), т.е.

ara [0] [0] [0] ==0

ara [0] [0] [1] ==1
ara [0] [0] [2] ==2
ara [0] [0] [3] ==3
ara [0] [0] [4] ==4
ara [0] [0] [5] ==5
…
ara [0] [1] [0] ==6
ara [0] [1] [1] ==7,

а остальные 16 элементов остались неинициализированными. Если необходимо инициализировать часть элементов не в начале многомерного массива, то можно вводить дополнительные фигурные скобки, выделяя ими последовательности значений, относящиеся к одной размерности, например:

```
int A [4] [5] [6] = {
                      { {0} },
                      { {100}, {110, 111}  },
                      { {200}, {210, 220, 230, 231} }
                    {;
```

задает только некоторые элементы:
A [0] [0] [0] == 0, A [1] [0] [0] == 100, A [1] [1] [0] == 110, A [1] [1] [1] == 111, A [2] [0] [0] == 200 и т.д.

Если многомерный массив при определении инициализируется, то его самая левая размерность может в скобках не указываться, т.к. количество элементов компилятор определит по числу членов в инициализирующем списке, например:

```
float  macy [ ] [5] = { {1},
                        {2},
                        {3}
                      };
```

формирует двумерный массив macy с размерами 3 на 5, но определяет явно не все его значения, а только члены **macy [0] [0] ==1, macy [1] [0] == 2, macy [2] [0] == 3.**

Доступ к элементам массива, как и у одномерных, с помощью индексированных переменных и с помощью указателей. Размер памяти для указателя 2 или 4 байта, поэтому возможность оперировать в программах указателями, а не самими массивами (которые могут иметь Мегабайтные размеры) значительно отличает С++ от других языков программирования. Например, для массива **float AR [3] [12] [7]** выражение *** (AR +1)** даст адрес элемента **AR [1],** т.к. * это разыменование, т.е. указание на адрес **(AR+1),** который в данном случае определится суммой номера адреса самого массива (а для массивов имя массива является его адресом и совпада-

ет с адресом первого, т.е. нулевого элемента), т.е. **(0+1) ==1**. Аналогично ***(AR +8)** дает адрес элемента AR [8]. Для трехмерного массива **int AD [x] [y] [z]** выражение ***(*(*(AD+2) +1) +0) == 210** дает трехмерный адрес элемента **210**.

Указатель (pointer) – это переменная, в которой хранится адрес оперативной памяти компьютера другой переменной. **Массивы указателей** оказались очень удобным инструментом для операций с матрицами, отсутствующим в других языках программирования. Например, задав массив указателей на матрицы, при умножении двух матриц компилятор только указывает на них и через регистры проходит по 4 байта на каждый указатель, иначе компилятор должен был бы пропустить через регистры все элементы матриц в полном объеме, а это задача долговременная.

При задании массива матриц требуется строго соблюдать синтаксис. Например, выражение **int * arra[6];** вводит массив указателей на объекты типа int, а выражение **int (* frr) [6]** вводит указатель **frr** на массив из 6 элементов.

По определению массива его элементы должны быть однотипными и иметь одинаковый размер. Например, создаем список учеников как двумерный массив типа **char.** В определении для элементов массива необходимо задать предельные размеры каждого из двух индексов – количества фамилий (например, 25) и максимальное число символов (19) в фамилии **char spisok [25] [20]**.

Пусть в списке 3 фамилии

char spisok [ ] [20], без указателей потребует памяти:

spisok (имя массива → **К о ш к и н** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 – 20 байт
указатель константа) **М ы ш к и н** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 – 20 байт
К о л б а с к и н – С ы р о в 0 0 0 0 0 – 20 байт

Тогда для списка потребуется 60 байт.

Применим указатели типа **char *,** каждый из которых будет указателем переменной, занимающей 2 байта. Одномерный массив указателей типа **char *** запишем как:

char * spisok [ ] = {“Кошкин”, ” Мышкин”, ”Колбаскин – Сыров”};

Для работы с ним потребуется память в 36 байт, а именно:

spisok (имя массива - → * → **К о ш к и н 0** = 2+7 байт
указатель константа) * → **М ы ш к и н 0** = 2+7 байт
* → **К о л б а с к и н - С ы р о в 0** = 2 +16 байт.

Всего 36 байт и указатели типа **char.**

После каждой строковой константы требуется 1 дополнительный байт (0 после фамилии) для её завершения.

Видно, что даже для примитивного по объему списка выигрыш по скорости работы велик, поэтому работа с массивами указателей весьма

эффективна. Применение массивов указателей позволяет рационально решать множество задач матричного исчисления и задачи сортировки сложных объектов с разными размерами, например, для упорядочения списков по алфавиту.

Массивы динамической памяти

Динамическая память выделяется операцией **new тип массива,** но размеры массива должны быть определены.

long (*lp) [2] [4]; // Определили указатель lp на массив без имени

lp = new long [3] [2] [4]; // Выделили память для массива 3*2*4 байт

В определении без круглых скобок обойтись нельзя. Указатель **lp** становится средством доступа к памяти с размером **3*2*4*** sizeof (long) байтов. Указатель **lp** в отличие от имени массива (которого и нет в этом примере) есть переменная, что позволяет менять ее значение и перемещаться по элементам памяти. Изменять значение указателя на динамический массив нужно осторожно, так как указатель, значение которого определяется при выделении памяти, используется затем для освобождения памяти операцией **delete [ ] lp;**

Инициализация динамических массивов, в отличие от определения массивов, не относящихся к динамической памяти, не производится. Поэтому для выделения памяти динамическим массивам их размер должен быть определен полностью, как, например, **new long [3] [2] [4];**

Есть еще одно ограничение на динамический массив. Только самый левый его размер может быть задан переменной (в нашем примере **lp**), а остальные размеры многомерного массива задаются константой. Обойти это ограничение можно только применением массивов указателей.

Контрольные вопросы

1. Какие две категории указателей в C++ вы знаете?
2. Приведите пример инициализации указателя В C++.
3. Приведите пример операции разыменования в C++.
4. Поясните операции инкремента и декремента над указателями.
5. Дайте определение массива.
6. Что является указателем на массив?
7. Приведите примеры инициализации массива.
8. Как в C++ задаются многомерные массивы?
9. Приведите примеры применения массивов указателей.
10. Что является массивом динамической памяти?

Глава 5. ФУНКЦИИ, УКАЗАТЕЛИ, ССЫЛКИ

5.1. Определения функций

Последовательность инструкций по выполнению ряда действий в программировании называют **функцией**. Функции назначается имя, вызов которого приводит к выполнению этих инструкций. Однажды определив функцию, нам не потребуется после повторять все входящие в нее инструкции.

Компьютеру нужно объяснить действие функции и её имя. Действие функции – это последовательность инструкций, значит нужно еще показать, где первая, а где последняя инструкция. Выделяют первую и последнюю инструкции с помощью тела функции, имеющего открывающую и закрывающую фигурные скобки { }. Внутри этих скобок (тело функции) строки инструкций обычно сдвигают вправо. Компьютеру на это “громко чихать”, а программисту удобнее, и это очень хороший стиль.

Следует помнить, что если в конце строки с именем функции поставить точку с запятой, то этим даем понять компилятору, что наша инструкция на этом заканчивается (еще до начала тела функции). Поэтому после имени функции следуют круглые скобки, а точка с запятой не ставится. В круглых скобках при необходимости указывается список формальных параметров.

По сути **функция – это набор действий**, которому мы присвоили собственное имя. Эти действия могут выполняться с помощью различных объектов. Когда создается функция, то обычно неизвестны реальные объекты, которые используются при выполнении функции. Подразумеваемые объекты называют **параметрами,** а реальные объекты, которые используются при выполнении функции называют **аргументами.**

В языках Алгол, Паскаль, Фортран и др. делается различие между программами, подпрограммами, процедурами, функциями, а в C++ этого нет, так как используются только функции. Собственно любая программа на C++ это совокупность функций. Из них только одна с именем **main (главная)** должна включаться в каждую программу, но только один единственный раз, так как именно эта функция создает точку входа в откомпилированную программу.

Кроме функции **main** в программу может входить множество различных не главных функций. Всем именам функций по умолчанию присваивается класс памяти **extern**, то есть статическая продолжительность существования и глобальность. Глобальность означает, что каждая функция при определенных условиях доступна в модуле и даже во всех моду-

лях программы, но для доступности в модуле функция должна быть определена или описана до её первого вызова.

В **определении функции указывается** последовательность действий, выполняемых при её вызове, имя функции, тип функции (то есть тип возвращаемого от неё значения) и совокупность формальных параметров (аргументов). Имя функции и совокупность параметров называют **сигнатурой функции.** Сигнатура функции зависит от количества параметров, их типов, порядка размещения в спецификации формальных параметров.

Определение функции имеет вид:

тип_функции имя_функции (спецификация_формальных_ параметров) тело функции

Тип функции – это тип возвращаемого функцией значения. Поясню, что это такое. Например функция расчета сдачи должна оперировать с тремя значениями: уплаченной суммой, ценой товара и требуемой сдачей. Функция должна рассчитать разность, а результат поместить в третью переменную, то есть возвратить в основную программу. Первые две переменные осуществляют ввод данных в функцию, а третья – вывод из функции по ссылке. Есть функции, не возвращающие никакого значения – их тип **void.**

Чтобы функция возвращала значение, необходимо:

- в заголовке функции определить тип её возвращаемого значения. Если он отличен от **void,** то функция возвращает результат даже если тип не указан (в C++ это равносильно заданию типа **int**);
- используя ключевое слово return, включить в тело функции инструкцию, определяющую возвращаемый результат.

Спецификация формальных параметров – это либо пусто, либо **void**, либо список отдельных параметров, в конце которого можно поставить многоточие. Спецификация формального параметра имеет вид:

тип имя_параметра

тип имя_параметра = умалчиваемое_значение

Более того, синтаксис разрешает параметры без имен, если последние не используются в теле функции.

Тело функции – это блок или составной оператор, т.е. последовательность описаний и операторов, заключенная в фигурные скобки. Важным оператором является оператор возврата в точку вызова – **return выражение;** или **return.** Выражение в операторе **return** определяет возвращаемое функцией значение, и именно оно будет результатом обращения к функции. Тип возвращаемого значения определяется типом функции. Если функция не возвращает никакого значения (тип **void),** то выражение в операторе **return** опускается, да и сам оператор становится необязателен в теле функции.

Примеры определения функций с разными сигнатурами:

void print (char * name, int value) // Ничего не возвращает
{ cout << "\n" << name << value; // Нет оператора **return }**
float min (float a, float b) // В функции два оператора
{ if (a<b) return a; // Возвращает минимальное
return b; // из значений аргумента **}**
float cube (float x) // Возвращает значение типа **float**
{ return x*x*x; // Возведение в куб вещественного числа **}**
int max (int n, int m) // Вернет значение типа **int**
{ return n < m ? m : n; // Вернет максимальное из значений **}**
void write (void) // Ничего не возвращает
{ cout << "\n Введение:"; // Всегда печатает одно и то же **}**

Заголовок последней функции может не иметь **void** в списке параметров, т.е. будет **void write ( )**

В С++ при обращении к функции формальные параметры заменяются фактическими со строгим соответствием по типам, что требует до первого обращения к функции поместить либо её определение, либо её описание или прототип, содержащие сведения о типе результата (возвращаемого значения) и о типах всех параметров. Наличие описания или прототипа позволяет компилятору контролировать соответствие типов.

Прототип (описание) функции может внешне совпадать с заголовком её определения, но в конце стоит точка с запятой:

тип_функции имя_функции (спецификация_ формальных_ параметров);

Кроме точки с запятой есть еще одно отличие – необязательность имен формальных параметров в прототипе, даже когда они есть в заголовке функции.

Примеры прототипов функций:

void print (char * , int); // Опустили имена параметров
float min (float a, float b);
float cube (float x);
int max (int, int m); // Опустили одно имя **void write (void);** // Список параметров может быть пустым
double Norma (double, double, double); // Опущены имена всех параметров.

Обращение к функции (вызов функции)

Обращение к функции или её вызов – это выражение с операцией круглые скобки. Операндами служат имя функции (либо указатель на функцию) и список фактических параметров:

имя_функции (список_фактических параметров).

Значением выражения “вызов функции” является возвращаемое функцией значение, тип которого соответствует типу функции.

Фактический параметр (аргумент) функции – это в общем случае выражение. Фактические параметры передаются из вызывающей программы в функцию по значению, т.е. вычисляется значение каждого выражения, представляющего аргумент, и именно это значение используется в теле функции вместо соответствующего формального параметра. Поэтому список фактических параметров – это либо пусто, либо **void,** либо разделенные запятыми фактические параметры. Приведем пример:

```
int max (int n, int m) // Определение до вызова функции
 {
    return n < m ? m : n;
    void main (void)        // Главная функция
        {
          void print (char *, int );        // Прототип до определения
           float cube ( float x = 0 );      // Прототип до определения
           int sum = 5, k = 2;              //  Вложенные вызовы функции
           sum = max ( ( int) cube (float (k) ), sum ) ;
           print ( “\n sum = ”, sum) ;
         }
void print ( char * name, int value ) // Определение функции
    { cout << “\n” << name << value; }
      float cube ( float x )       // Определение функции
    { return x * x * x; }
 }
```

Результат выполнения программы **sum**=8.

При вызовах функций **max ( )** и **cube ( )** преобразования типов необходимы, чтобы согласовать типы формальных и фактических параметров. Взяты разные формы записи преобразований – каноническая и функциональная.

Для функции **max ( )** прототип не нужен, т.к. её определение дано в том же файле до вызова функции. Прототипы функций **print ( )** и **cube ( )** необходимы, т.к. определения функций размещены после обращения к ним. Если переместить определение функции **max ( )** в конец программы и не ввести прототип в тело функции **main ( ),** то будет ошибка, на которую укажет компилятор.

При наличии прототипов вызываемые функции не обязаны размещаться в одном модуле с вызывающей функцией, а могут находиться в библиотеке объектных модулей или оформляться в виде отдельных модулей. Это относится и к стандартным функциям компилятора, но их описа-

ния в виде прототипов необходимо включать в программу дополнительно. Обычно это делают с помощью препроцессорных команд

 # include < имя_файла>, например **# include <iostream.h>**

При разработке своих программ, содержащих большое количество функций в разных модулях, следует создать свой отдельный файл с прототипами функций и описаниями внешних объектов (переменных, массивов и др.), который препроцессорной командой

 # include “ имя_файла”

включать в начало каждого из модулей программы. В отличие от библиотечных функций имя этого файла записывается не в угловых скобках, а в кавычках.

В этом случае не происходит увеличение размеров программы, т.к. прототипы нужны только для компиляции и не переносятся на объектный модуль, т.е. не увеличивают машинного кода. Например, заголовочный файл с прототипами функций будет таков:

```
// Example.hpp – прототип функций
float min ( float a, float b ) ;
float cube (float x = 1 );
int max ( int, int m = 0 );
void write ( void );.
```

Начальные значения параметров функции

В определении функции можно использовать значения по умолчанию. Их используют, когда при вызове функции этот параметр опущен. Но есть ограничение – если параметр имеет умалчиваемое значение, то все параметры справа от него также должны иметь умалчиваемые значения.

5.2. Функции с переменным количеством параметров

Функция – это группа инструкций, которой назначено имя. Эти инструкции выполняются при вызове функции по имени. Однажды определив функцию, мы не будем в дальнейшем повторять все эти инструкции. Но компьютер не помнит её всегда, поэтому требуется включать код функции (вызов по её имени) в каждую программу, где она необходима.

В C++ допустимы функции, количество параметров и типы параметров которых до вызова функции не определены. При определении и описании таких функций список формальных параметров заканчивается многоточием. Формат прототипа таких функций имеет вид

тип имя (спецификация_явных_параметров, …);

Здесь тип – это тип возвращаемого функцией значения; имя – это имя функции; спецификация явных параметров – это список отдельных параметров, количество и типы которых фиксированы и известны в момент компиляции. Запятая после этого списка не обязательна, а многоточие извещает компилятор, что дальнейший контроль соответствия типов параметров при обработке вызова функции проводить не нужно.

Сложность в том, что у переменного списка параметров нет имени и непонятно становится, как найти его начало и где список заканчивается. Для преодоления этого используют два разных метода. Один основан на добавлении в конец списка реально использованных параметров специального параметра индикатора с уникальным значением, показывающим конец списка. В теле функции параметры перебираются и их значение сравнивается с этим признаком.

Второй способ основан на передаче в функцию числа реального количества фактических параметров с помощью одного из явно задаваемых параметров. Оба метода основаны на использовании адресной арифметики, то есть указателей.

Например: функция суммирует значения своих параметров типа int

```
long summa (int k, …) // k – число суммируемых параметров
{  int * pik = & k;
     long total = 0;
      for ( ; k; k-- ) total += * (++pik);
      return total;
}
void main ( )
     {cout << “\n summa ( 2,6, 4) = “<< summa (2,6,4);
       cout << “\n summa (6,1,2,3,4,5,6) = “<< summa (6,1,2,3,4,5,6);
     }
```

Результат summa (2,6,4) = 10

summa (6,1,2,3,4,5,6) = 21

Здесь для доступа к списку параметров использован указатель **pik** типа *** int**. Ему присваивается адрес явно заданного параметра **k**, т.е. он устанавливается в памяти на начало списка. Затем указатель **pik** перемещается по адресам параметров и с помощью операции разименования *** pik** выполняется выбор их значений. Значение аргумента цикла суммирования k уменьшается на единицу после каждой итерации вплоть до нуля. Первый член указывает число суммируемых параметров.

Рассмотрим пример функции для вычисления произведения переменного количества параметров. Признаком окончания списка служит параметр с нулевым значением.

Double prod (double arg, …)

```
    {double aa = 1.0; // Формируемое произведение
    double *prt = &arg; // Настроили указатель на первый параметр
    if (*prt == 0.0) return 0.0;
    for ( ; *prt; prt++) aa *= *prt;
    return aa;}
    void main ( 0 )
    {
    double prod ( double, …); // Прототип функции
    cout << "\n prod (2.0, 4.0, 3.0, 0.0) = "<< prod (2.0, 4.0, 3.0, 0.0);
    cout << "\n prod (1.5, 2.0, 3.0, 0.0) = "<< prod (1.5, 2.0, 3.0, 0.0);
    cout << "\n prod (1.4, 3.0, 0.0, 16.0, 100.0, 34.5, 0.0) = " << prod (1.4,
3.0, 0.0,
    16.0, 100.0, 34.5, 0.0);
    }
```

Результат выполнения программы:

```
prod (2.0, 4.0, 3.0, 0.0) = 24
prod (1.5, 2.0, 3.0, 0.0) = 9
prod (1.4, 3.0, 0.0, 16.0, 100.0, 34.5, 0.0) = 4.2
```

Все параметры при обращении к функции **prod ( )** должны иметь тип **double.** После нулевого значения параметра все следующие по списку параметры игнорируются.

Для обеспечения мобильности программ с других языков в компиляторы С++ включают также дополнительный файл **stdarg.h**, содержащий макрокоманды и определяющий специальный тип данных **va_list** для обработки переменных списков параметров. В теле функции определяют объект этого типа, обладающий свойствами указателя, который связывают с началом переменного списка. В качестве второго аргумента используется последний из явно определенных параметров, стоящий перед многоточием.

5.3. Рекурсивные функции

Рекурсивные функции – это функции в которых для вычисления i-го члена, необходимо вычислить предварительно i-1 член. Различают рекурсию прямую и косвенную. Прямая рекурсия, – если в теле функции явно используется вызов этой функции, а косвенная, – если содержит обращение к другой функции, содержащей в свою очередь вызов первой функции. Например, расчет факториала – прямая рекурсия.

```
long fact (int k)
{
```

```
  if (k < 0) return 0;
  if (k == 0) return 1;
  return k * fact (k-1);
}
```

Для отрицательного аргумента результат по определению факториала не существует и функция возвращает нулевое значение, Для нуля функция возвращает значение 1, т.к. по определению факториал **0! =1**. И значение организуется вычислением произведения K*(K-1)*(K-2)*…*3*2*1*1.

Рекурсивные обращения прекращаются только при вызове функции **fact (0),** который приводит к последнему значению 1 в произведении, т.к. последнее выражение, из которого вызывается функция, имеет вид **1* fact (1-1).**

Рекурсивная функция полезна при возведении в целую степень вещественного числа (в С++ отсутствует явная операция возведения в степень, и эта операция в вычислительном файле **math.h** выполняется специальной функцией **pow,** например:

```
#include <conio.h>
#include<iostream.h>
#include <math.h>
void main(   )
{   clrscr( );
    int A = 9.0;
    float B = 0.5;
    double C;
    C=pow (A, B);
    cout <<C;
    getch ( );
}
```

Результат 3.

Такая операция **pow** основана на сложном представлении возведения в степень через логарифмы и экспоненциальные функции.

С помощью рекурсивных функций вообще-то сравнительно легко реализуется вычисление целой степени вещественного ненулевого числа

```
#include <conio.h>
#include<iostream.h>
double expo (double , double);
void main( )
{  clrscr( );
    double Z = expo (2, 10);
    cout << Z;
    getch ( );
```

```
}
double expo(double A,double B)
  {    if ( B == 0) return 1;
       if (A==0) return 0;
       if (B > 0) return A * expo (A, B-1);
       return expo (A, B+1) / A;
}
```

Результат: 1024.

5.4. Подставляемые функции

С помощью специального служебного слова **inline** можно определить функцию как подставляемую. При многократных обращениях к какой-либо функции в программе много времени расходуется на передачу управления к вызываемой функции и возврата из неё. Подстановка вместо вызова функции кодов операторов тела функции с помощью спецификатора **inline** исключает эти затраты, что важно в сложных задачах моделирования или в компьютерной графике, где объемы программ требуют большого времени работы. На применение подставляемых функций есть ряд ограничений и если они не выполняются, то компилятор проигнорирует спецификатор **inline** и вызовет функцию как обычную. К таким ограничениям относятся:

- большой размер встраиваемой функции;
- рекурсивность встраиваемой функции;
- размещение обращения до определения;
- неоднократное встраивание в одно и то же выражение;
- наличие цикла во встраиваемой функции.

5.5. Функции и массивы

Массивы могут быть параметрами функций и функции могут возвращать указатель на массив в качестве результата. Возникает задача определения количества элементов массива, которые использовались в качестве рабочих параметров функции. Со строками просто, т.к. массивы типа **char [ ]** имеют в конце каждой строки символ **‘\0’** и он считается концом строки. С помощью функции **len ( )** можно определять размер строки.

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

```
        int len (char e [ ])
{   int m = 0;
    while (e[m++]);
     return m-1;
}
        void main ( )
        { char e [ ] = "<Бит - не синяк под глазом, а единица информации";
          cout << len (e);
        }
```

Результат: 39 символов.

Для не символьных массивов нужно либо использовать массивы заранее определенного размера, либо передавать значение размера массива в функцию явным образом, что делают с помощью дополнительного параметра. К сожалению в С++ невозможно по имени массива определять его размерность и размеры по каждому измерению. Например, запись **double kryk [6] [4] [2]** описывает не трехмерный массив, а одномерный с именем **kryk** , включающий **6** элементов типа **double [4] [2],** каждый из которых в свою очередь является одномерным массивом из **4** элементов типа **double [2].** И каждый из этих элементов является массивом из двух элементов типа **double**. Поэтому в С++ для работы с массивами, например с матрицами, используют специально формируемые динамические массивы указателей, передачу матриц через параметры и другие сложности.

5.6. Указатели на функцию

Функция характеризуется типом возвращаемого значения, именем и сигнатурой (списком параметров). Сигнатура определяется в свою очередь количеством, порядком следования и типами параметров. При использовании имени функции без последующих скобок и параметров имя функции выступает в качестве указателя на эту функцию и его значением служит адрес размещения функции в памяти. Это значение можно присвоить другому указателю, и он может также применяться для вызова функции. Однако в новом указателе должен быть тот же возвращаемый функцией тип и та же сигнатура, что и в старом. Указатель на функцию имеет вид

тип_функции (* имя_указателя) (спецификация_параметров);.

Например, **float (* sts) (int);** // определение указателя **sts** на функцию с параметрами типа **int**, возвращающую значение типа **float**. Если записать без первых круглых скобок, то компилятор воспримет **float * sts**

(int); как прототип некоей функции с именем **sts** и параметром типа **int**, возвращающий значение указателя типа ***float**.

Более сложный пример синтаксиса: **char * (* fscs) (char *, int);** // Определение указателя **fscs** на функцию с параметрами типа **указатель на char** и типа **int,** возвращающего значение типа **указатель на char.** В определении указателя на функцию тип возвращаемого значения и сигнатура должны совпадать с соответствующими типами и сигнатурами тех функций, адреса которых предполагается присваивать вводимому указателю при инициализации или оператором присваивания.

Приведем пример программы, иллюстрирующей использование указателей на функцию.

```
//  Определение и использование указателей
#include <conio.h>
#include<iostream.h>
void f1 (void)                          //  Определение f1
{   cout << "\n Выполняется f1 ( )";
}
void f2 (void)                          //  Определение f2
{ cout << "\ n Выполняется f2 ( )";
}
void main( )
        { clrscr( );
          void (*btr)(void);            // btr – указатель на функцию
          btr = f2;                     // Присваивается адрес f2 ( )
          (*btr)( );                    // Вызов f2 ( ) по её адресу
          btr = f1;                     // Присваивается адрес f1 ( )
          (*btr)( );                    // Вызов f1 ( ) по её адресу
          btr ( );                      // Вызов эквивалентен (*btr) ( );
          getch (  );
        }
```

В этом примере функция типа **void**, которой последовательно присваиваются адреса **f1** и **f2,** вызывается с помощью указателя **btr** на неё:

(* имя_указателя) (список_фактических_параметров)

А значением имени-указателя служит адрес функции. С помощью операции разименования * обеспечивается обращение по адресу к этой функции. Ошибочна запись обращения без скобок, напрямую в виде ***btr();,** т.к. операция **( )** имеет более высокий приоритет чем операция * и будет сделана попытка сначала выполнить её, а уж потом вызвать функцию, что неверно и компилятор “ругнётся”.

При определении указатель на функцию может быть инициализирован адресом функции, тип и сигнатура которой соответствуют определяе-

мому указателю. При присваивании указателей на функцию также требуется соблюдать соответствие типов возвращаемых значений функции и сигнатур для указателей левой и правой частей оператора присваивания. Это должно выполняться и при вызове функции с помощью указателей, т.е. типы и количество параметров должны соответствовать формальным параметрам вызываемой функции.

Покажем на примере вызов функций **A,B,C,D** и выполнение их операторов. В каждой итерации указатель **E** получает адрес одной из функций и изменяет значение переменной **z.** Цикл продолжается пока значением переменной не станет пробел

```
#include <conio.h>
#include<iostream.h>
int A (int N, int M) { return N+M;}
int B (int N, int M) { return N/M; }
int C (int N, int M) { return N*M ; }
int D (int N, int M) { return N-M;}   // Определение функций A, B, C, D
void main()
{    clrscr ();
     int (*E) (int,int);// Указатель E на функцию
     int x = 8, y = 2;
     char z = '+';
     while ( z != ' ')
       { cout << "\n  x = " << x << ", y = " << y;
         cout<< ". \n  z =  ' " << z << " ' " << " равен";
       }
         switch (z)
           { case '+': E = A; z = '/'; break;
             case '-': E = D; z = ' '; break;
             case '*': E = C; z = '-'; break;
             case '/': E = B; z = '*'; break;
           }
       cout << ( x = (*E)(x,y)); // Вызов по адресу через указатель}
    getch (  );
}
```

Результат: x = 10, 5, 10, 8.

Указатели на функцию можно объединять в массив, который удобно использовать при разработке различных программ, управляемых с помощью меню. При этом предлагаемые услуги оформляются в виде функций, адреса которых помещаются в массив указателей на функции. Пользователь, выбирая из меню, например номер пункта, как по индексу выбирает

из массива нужный адрес функции, а обращение к функции по этому адресу обеспечивает выполнение требуемых действий.

Указатели на функцию – исключительно удобный механизм, когда объектами обработки являются функции. Например, создавая процедуры для вычисления определенного интеграла задаваемой пользователем функции нужно передавать в программу функцию. В этом случае удобно организовывать связь между функцией, реализующей метод обработки, например численного интегрирования, и той функцией, которую интегрируем, через указатели на функцию.

Наряду с достоинствами применения указателей на функции есть у них и существенный недостаток – синтаксис определения конструкций, включающих указатели на функции, сложен и создает неудобства даже опытным программистам.

5.7. Ссылки

Ссылка – это другое имя уже существующего объекта. Дело в том, что когда мы вызываем функцию с аргументом, то компьютер создает копию объекта и затем использует её в функции. С самим же объектом ничего не происходит, независимо от тех операций, которые выполняются в функции. Когда нужно изменить исходный аргумент функции, требуется поработать с самим объектом, и это делается с помощью ссылки. Передача по ссылке может приводить к модификациям оригинала, и если это нежелательно, то ссылки лучше не применять.

Раз ссылка – другое имя уже существующего объекта, то для его отличия введен символ ссылки **& (символ амперсанда)** и для определения используется такая запись

тип & имя_ссылки инициализатор

В качестве инициализирующего выражения должно выступать имеющее значение леводопустимое выражение, т.е. имя объекта, имеющего место в памяти компьютера. Приведем пример определения ссылок:

Int E = 12; // Определена и инициализирована переменная E

Int & B = E; // Значением ссылки B является адрес переменной E

В определении ссылки ее символ **&** не является частью типа. Ссылка ведет себя так же, как и переменная подобного типа. И если переменная рассматривается как связанная пара **имя переменной – значение переменной**, то и ссылка должна представляться парой **имя ссылки – значение переменной** и поэтому понятна необходимость инициализации ссылки при её определении. Тут большая схожесть с константами – раз ссылка

есть имя связанное с объектом, уже размещенным в памяти, то требуется с помощью начального значения определить тот участок памяти, на который указывает ссылка.

После определения с инициализацией имя ссылки становится псевдонимом уже существующего объекта и для нашего примера оператор **B +=15** увеличит на число 15 значение переменной **E.** Имена переменной и ссылки на неё полностью равноправны, т.к. соотносятся с одним и тем же участком памяти.

Например:

```
void main ( )
{
    long A = 1234567; // Выделили память для переменной A
    long &  B = A; // Определили синоним A
    cout << "\n B = "<< B << "\t A = " << A;
    A = 7654;
    cout << "\n B =" << B << "\t A =" << A;
}
```

Результат выполнения программы будет:

B = 1234567 A = 1234567

B = 7654 A = 7654

Присваивание значения переменной приводит к изменению значения ссылки и, наоборот, присваивание значения ссылке приводит к изменению значения переменной. Но ссылка не полноценный объект, подобный переменным или указателям, т.к. после инициализации её значение изменить нельзя и она всегда смотрит на тот участок памяти, с которым она связана инициализацией. Ни одна из операций не действует на ссылку, а относится к тому объекту, с которым она связана. В принципе это основное свойство ссылки.

Ссылки аналогичны исходным именам объектов и есть аналогия с указателями, но отсутствует необходимость в явном разименовании, что обязательно при обращении к значению переменной через указатель. Например:

```
double a[ ] = {10.0, 20.0, 30.0, 40.0}; // Массив
double * pa = a;                  // pa – указатель на массив a
double & ra = a [ 0 ];            // ra – ссылка на первый элемент массива
double * & rpd = a;               // ссылка на указатель (на имя массива)
```

Для этого примера соблюдаются равенства:

pa ==& ra, * pa == ra, rpd == a, ra == a[0]/

Определим операцией **sizeof** объемы занимаемой памяти:

sizeof * pa ==4 байта – размер указателя типа double *;

sizeof (pa) == 8 байт –размер элемента массива **double [0],** связанного со ссылкой **ra** при инициализации;

sizeof (rpd) == 4 размер указателя на массив с элементами **double.**

Результат применения операции **sizeof** к ссылке дает не ее размер, а размер именуемого ею объекта и каждая операция к ссылке есть операция над тем объектом, с которым она связана.

Есть ограничения на применение ссылок:

- ссылка не может иметь тип **void** и определение **void & имя ссылки** – запрещено;
- ссылку нельзя создавать с помощью операции **new,** т.е. для ссылки нельзя создавать участок памяти;
- не определены ссылки на другие ссылки;
- нет указателей на ссылки;
- невозможно создавать массив ссылок.

После определения ссылка не может быть оторвана от объекта инициализации никаким способом, т.е. значение ссылки не может изменяться. Однако один и тот же объект может быть адресован любым числом ссылок и указателей, например:

```
long v = 7;             // Определена переменная v
long& ref1 = v;         // Ссылка ref1 связана с v
long * ptr = &ref1;     // Указатель ptr адресует v
long & ref2 = ref1;     // Ссылка ref2 указывает на v
```

После этого к значению переменной **v** можно добраться четырьмя способами: с помощью имени **v**, ссылки **ref1,** разименования указателя ***ptr,** ссылки **ref2.**

Применение ссылок в C++ дает возможность повысить эффективность обмена с функциями через аппарат параметров. При использовании ссылки в качестве формального параметра обеспечивается доступ из тела функции к соответствующему фактическому параметру, т.е. к участку памяти, выделенному для фактического параметра-аргумента. При этом параметр ссылка дает те же возможности, что и указатель, но не требует операции разименования *, а фактическим параметром должен быть не адрес (как для параметра-указателя), а обычная переменная. С другой стороны ссылка на функцию, обладая всеми правами на имя функции, является его псевдонимом. Изменить значение ссылки на функцию невозможно, поэтому указатели на функцию имеют гораздо большую степень применения, чем ссылки.

5.8. Перегрузка функций

Цель перегрузки в том, чтобы одна и та же функция по разному выполнялась и возвращала разные значения при обращении к ней с разными по типам и количеству фактическими параметрами. Для обеспечения перегрузки нужно для конкретного имени определить, сколько разных функций связано с ним, т.е. сколько вариантов сигнатур допустимы для обращения с этим именем. Распознавание перегруженных функций выполняется по их сигнатурам, поэтому перегруженные функции должны иметь одинаковые имена, а спецификации параметров должны быть различными (по типам, количеству, по расположению).

5.9. Шаблоны функций

Шаблоны семейства однотипных функций создаются для автоматизации создания функций, которые могут обрабатывать разнообразные данные. Шаблон в отличие от перегрузки определяется один раз, но обязательно параметризируется тип возвращаемого функцией значения и типы любых параметров, количество и порядок размещения которых заданы. Для этого используют список параметров шаблона.

В определении шаблона семейства функций используют служебное слово **template.** Список формальных параметров шаблона заключают в угловые скобки <>, а каждый формальный параметр обозначают служебным словом **class,** за которым следует идентификатор. Приведем пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template <class type>
type abs (type X)   {return X>0 ? X: -X;}
```

Сам шаблон состоит из двух частей:

- заголовка **template <список параметров шаблона>;**
- определения функции, в котором тип возвращаемого значения и типы любых параметров обозначаются именами параметров шаблона, введенных в заголовке.

Те же имена используют для обозначения типов локальных объектов в теле определения функции.

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по вызовам из текста программы. Например, если программист приведет обращение **abs (-5.4),** то компилятор сформирует такое определение функции:

double abs (double X) { return X > 0 ? X : - X;}

и будет организовано выполнение именно этой функции, а в точку вызова в качестве результата вернется числовое значение –5.4.

Рассмотрим шаблон семейства функций для обмена значений двух передаваемых им параметров.

```
template < class T >
void swap  ( T* x,  T* y)
      {    T z = *x;
            x=y;
            *y = z;
       }
```

Если в программе есть такой шаблон и появится, например, последовательность операторов **long k = 4, d = 8; swap (&k, &d);**
то компилятор сформирует определение функции

```
void swap ( long* x, long* y)
{ long z = *x; *x = *y; *y = z;
}
```

и будет выполнено обращение именно к этой функции и значения k и d поменяются местами.

Если в той же программе будут операторы
double a = 2.33, b = 3.22; swap (&a, &b); то сформируется и выполнится функция

```
void swap (double* x, double* y)
{   double z = *x;
    *x = *y; *y = z;
}
```

При использовании шаблонов нет необходимости готовить заранее варианты функций с перегруженным именем, т.к. по существу механизм шаблонов функций позволяет автоматизировать подготовку определений перегруженных функций. Компилятор автоматически по вызовам из программы формирует необходимые определения именно таких типов параметров, которые использованы в обращениях.

Основные свойства параметров шаблона:

- имена параметров шаблона должны быть уникальными во всем определении шаблона;
- список параметров шаблона функции не может быть пустым;
- в списке параметров может быть несколько параметров, каждый из которых начинается со служебного слова **class,** например:

  template <class type1, class type2, class type3>;
- нельзя использовать в заголовке шаблона параметры с одинаковыми именами;

- имя параметра шаблона (в наших примерах **type1, type2, type3)** имеет в определяемой шаблоном функции все права имени типа, т.е. определяет тип возвращаемого функцией значения;
- одно имя нельзя использовать для обозначения нескольких параметров шаблона, но в разных шаблонах функций могут быть одинаковые имена, поскольку действие параметров шаблона заканчивается в конце определения шаблона и после этого идентификатор свободен.

Как и для обычных функций, для шаблонов есть описания и определения. В качестве описаний используется прототип шаблона:

template < список параметров шаблона> тип имя функции (спецификация параметров)

Например: **template <class F>**
void swap (F, F);

Контрольные вопросы

1. Что такое функция?
2. Из чего состоит сигнатура функции в C++?
3. Что такое тип функции?
4. Что такое прототип функции?
5. Как задаются начальные значения параметров функции?
6. Приведите пример функции с переменным числом параметров.
7. Приведите пример рекурсивной функции.
8. Что такое подставляемые функции в C++?
9. Как использовать массивы в качестве параметров функции?
10. Как применяются указатели на функцию?
11. Что такое ссылки в C++?
12. Что такое перегрузка функций в C++?
13. Что такое шаблоны функций в C++?

Глава 6. СТРУКТУРЫ И ОБЪЕДИНЕНИЯ

6.1. Структура как тип данных

Структура – это производный тип данных, сконструированный из других типов языка С++. Другими словами, – это объединенное в единое целое множество поименованных элементов в общем случае разных типов (в отличие от массивов, элементы которых являются однотипными).

Имя нового производного типа структуры задается самим программистом. Определение структуры начинается со служебного слова **struct,** за которым помещается выбранное имя производного типа, а затем в фигурных скобках, закрытых точкой с запятой, описания элементов, входящих в структуру. Элементы могут быть как базовых, так и производных типов. Приведем пример определения структуры для библиографической карточки:

```
struct card {  char *author;
               char *title;
               char *city;
               char *firm;
               int year;
               int pages;
               };
```

Новый производный тип назвали **card**, в котором будут элементы базового типа **int** и производного **char *.** Для нового структурного типа описывают структурированные объекты, например: **card rec1, rec2, rec3;** Здесь определены три объекта с именами **rec1, rec2, rec3,** каждый из которых в качестве элементов содержит свои собственные **char *author; char *city; …**.

Если структура определяется однократно, то нет необходимости вводить именованный структурный тип, а можно определять объекты одновременно с определением их компонентного состава.

Следующий оператор определяет две структуры с именами **A, B, массив структур с именем CC и указатель pst** на структуру:

```
struct {char N[12]; int value;} A, B, CC[8], *pst;
```

В **A, B** и в каждый элемент массива **CC[0], …,CC[7]** входят в качестве элементов массив **char N[12]** и целая переменная **value,** а имени у структуры нет.

Для обращения к конкретным объектам структуры используют уточненные имена, определяемые конструкцией с точкой:

имя структуры.имя элемента структуры

Например, для определенной выше структуры **B** оператор **B.value = 19;** присвоит переменной **value** значение **19**. Для ввода значения переменной **value** структуры **CC[3]** можно использовать оператор **cin>> CC[3].value;** Также можно вывести в выходной поток **cout** значение переменной из любой структуры.

При определении структур возможна их инициализация. Введя предварительно конкретный тип структуры, можно инициализировать и конкретную структуру этого типа, например card:

card book = { “Феер К.”, “Беспроводная цифровая связь”, “Москва”, “Радио и связь”, 2000, 520 };

Такое определение эквивалентно следующей последовательности операторов:

card book;
book. author = “Феер К.”;
book. title = “Беспроводная цифровая связь”;
book. city = “Москва”;
book. firm = “Радио и связь”;
book. year = 2000;
book. pages = 520;

Необходимо отличать имя конкретной структуры (в наших примерах: **book, rec1, rec2, rec3, A, B, CC[0],…,CC[7]**) от имени структурного типа (в нашем случае **card**). С именем структурного типа не связан никакой конкретный объект и идентификатор **card** в нашем примере – это ярлык **(tag – тэг)** или этикетка структур, которые будут в дальнейшем определены в нашей программе. В приведенных примерах внутренний состав объектов, представляющих собой библиографические карточки, определен с помощью структурного типа с именем **card.** Этим вводится формат данных, входящих в будущие однотипные структуры и этот формат имеет права типа, введенного самим программистом.

Определение структурного типа можно совмещать с определением конкретных структур этого типа, например:

struct PRIM {char *name; long sum;} X,Y,Z;

Здесь определен структурный тип с придуманным именем **PRIM** и три структуры **X, Y, Z** с внутренним строением, включающим переменную **sum** типа **long** и элементы производного типа **char *.** Ошибочно будет использовать имя структурного типа для наименования элемента структуры: **PRIM.sum = 306L;**
верно будет **Y.sum = 306L;**

Разрешается определять указатель на структуры:

имя структурного типа *имя указателя на структуру;

Определяемый указатель может быть инициализирован. Значением указателя на структуру может служить адрес структуры того же типа. Например:

card *ptrcard = &rec2;

определен указатель **ptrcard,** которому присвоено значение адреса одной из структур **rec2** типа **card** и появляется дополнительная возможность (кроме обычной **rec2.pages**) доступа к элементам структуры rec2. Это достигается операцией ->

имя указателя - > имя элемента структуры

Например, количество страниц в структуре **rec2** будет значением выражения

ptrcard -> pages

Можно обратиться к элементу структуры путем разименования её указателя и формирования уточненного имени такого вида. При этом операция разыменования должна относиться только к имени указателя, например **(*ptrcard).pages**

Все три подхода дадут один и тот же результат и именуют элемент **int pages** конкретной структуры **rec2,** имеющей тип **card.**

С помощью служебного слова **typedef** можно вводить новые обозначения (идентификаторы) для типа данных и в качестве производного типа можно использовать структурный тип. Новое имя типа структуры **Akul** будет равноправным с основным именем **PRIM** структурного типа.

typedef struct PRIM {char *name; long sum;} Akul

Другими словами, имеется две возможности для определения имени структурного типа.

На элементы структур есть одно ограничение – они не могут иметь тот же самый тип, что и определяемый структурный тип. Однако разрешено элементом определяемой структуры ставить указатель на структуру определяемого типа, например:

struct kur { kur *d; int h; double w;}

Функция так взаимодействует со структурой:

- функция может возвращать структуру как результат:

struct help {char *name; int number;};

help func1 (void); // Прототип функции;

- функция может возвращать указатель на структуру:

help *func2 (void); // Прототип функции;

- функция может возвращать ссылку на структуру:

help & func3 (void); // Прототип функции.

Через аппарат параметров информация о структуре может передаваться в функцию непосредственно, или с помощью указателя, либо с помощью ссылки:

void func4 (help str); // Прямое использование
void func5 (help *pst); // С помощью указателя
void func6 (help& rst); // С помощью ссылки.

Использование ссылки на объект позволяет избежать дублирования объекта в памяти.

6.2. Объединения разнотипных данных

С помощью служебного слова **union** вводятся родственные структурам **объединения.** Принципиально они отличаются от структур тем, что все элементы объединения имеют один и тот же адрес, для которого выделяется память, определяемая размером занимаемой памяти максимального по объему памяти элемента. Определения структуры и объединения похожи

struct {long L; int D; float F;} STR; // Определение структуры
union {long L; int D; float F;} UNI; // Определение объединения

В принципе объединение можно рассматривать как структуру, все элементы которой размещены в одном и том же участке памяти с нулевым смещением от начала. Например, размещение в памяти объединения **UHH**

union {double X; int Y;} UHH;

представлено на рис. 2.

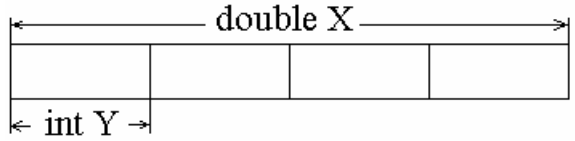

Рис. 2. Пример размещения в памяти объединения

Как и для структур, для объединения может быть введен произвольный тип, определяющий внутреннее строение всех объединений, относящихся к этому типу, например:

union имя_объединяющего_типа {элементы обединения};
union masha {double X; int Y;};

Введя тип объединения, можно определять конкретные объединения, их массивы, указатели и ссылки на объединения:

masha Ok, BU[4]; // объединение и массив объединений
masha *Luda; // Указатель на объединение
masha & vany = Ok; // Ссылка на объединение

Соответственно и обращаться к объединению можно различно:
имя_объединения. имя_элемента // Обращение по уточненному имени

(*указатель_на_объединение) . имя элемента // Обращение через указатель

ссылка_на_объединение. имя элемента // Обращение через ссылку.

Основное назначение объединений в обеспечении доступа к одному и тому же участку памяти с помощью объектов разных типов. Это бывает необходимо для выделения из объекта определенной его части.

Массивы и структуры могут быть элементами объединений. При определении конкретных объединений разрешена их инициализация, причем инициализируется только первый элемент объединения. При определении объединения без указания имени объединяющего типа разрешено не вводить даже имя объединения – создается безымянное (анонимное) объединение:

union {int X; int Y; double Z; char GG[5];};

6.3. Битовые поля структур и объединений

Компонентами структур и объединений могут быть битовые поля, которые представляют собой **знаковое или беззнаковое целое значение,** занимающее в памяти фиксированное число бит. Битовые поля не имеют адресов, т.е. для них не определена операция **&** , нет указателей и ссылок на битовые поля и их нельзя объединять в массивы.

Назначение битовых полей в том, что с их помощью можно обеспечить доступ к отдельным битам данных, т.е. они позволяют формировать объекты с длиной не кратной байту, что позволяет плотно упаковывать информацию.

Определение структуры с битовым полем имеет такой вид:

```
struct          { тип_поля   имя_поля : ширина_поля;
                  тип_поля   имя_поля : ширина_поля;
                       …
                } имя_структуры
```

Тип поля – это один из базовых целых типов – **int, signed int, char, short, long** и их знаковые и беззнаковые варианты. Пример структуры с битовыми полями:

```
struct {  int a: 5;
          int b :7;
       } many;
```

Обращение к битовым полям производится как к обычным структурам: **имя_структуры . имя_поля**

или **указатель_на_структуру -> имя_поля**

или**ссылка_на_структуру . имя_поля**
или **(*указатель_на_структуру) . имя_поля**
Например для структуры **many** и указателя на неё ***ckok**

```
many.a = 1;
ckok = & many;
ckok ->b =25;
```

Подробно применение битовых полей изучать не будем, запомним только, что с их помощью можно плотно упаковать информацию и достичь нужного бита в объекте.

Контрольные вопросы

1. Дайте определение структуры в С++?
2. Приведите пример определения структуры.
3. Каким образом осуществляется доступ к элементам структуры?
4. Как определить указатель на структуру?
5. Что такое объединение в С++?
6. Приведите пример объединения в С++.
7. В чем отличие структур от объединений?
8. Объясните понятие “битовые поля”.
9. Как применяются битовые поля?
10. Приведите пример динамического массива с использованием структур и указателей.

Глава 7. КЛАССЫ

7.1. Понятие класса

Для серьёзных задач требуются новые типы данных с большей полнотой отображающие особенности решаемой задачи. Для этих типов необходим и новый инструментарий – набор операций, удобных для обработки данных. В С++ введена возможность создавать свои типы данных и определять операции над ними с помощью понятия **класс.** Класс – это производный структурированный тип, введенный программистом на основе уже существующих типов. Это более широкое понятие чем структура. В С++ введена особая область видимости – класс из-за того, что данные класса не обязательно описывать до их первого использования в принадлежащих классу функциях. То же и для самих функций, принадлежащих классу – они могут обращаться друг к другу до определения внутри тела класса. И все компоненты класса видны во всех операторах его тела

Класс определяют конструкцией

ключ_класса имя_класса {список_компонентов};

где: **ключ класса** – одно из служебных слов: **class, struct, union;**

имя класса – произвольно выбираемый идентификатор;

список компонентов – определения и описания типизированных данных и принадлежащих классу функций.

Компонентами могут быть данные, классы, функции, перечисления, битовые поля, имена типов, дружественные классы и функции. Для простоты сначала будем считать, что компоненты – это базовые и производные типы и функции. Список компонентов в фигурных скобках называют телом класса. Перед телом заголовок из ключа и имени. Определение класса всегда заканчивается точкой с запятой.

В качестве ключа можно использовать слово **struct,** но класс шире структуры, хотя бы возможностью вводить компонентные (принадлежащие классу) функции. Например, введем класс “комплексное число”:

```
struct   complex1 // Вариант класса “комплексное число”
{  double real; // Вещественная часть
                  double imag; // Мнимая часть
// Определить значение комплексного числа
          void define (double re = 10.0,  double  im = 8.0)
      { real = re; imag = im; }
// Вывести на экран значение комплексного числа
                   void display (void)
            { cout << “ real = ” << real;
```

```
            cout << " imag = " << imag
        }
};
```

В отличие от структурного типа в класс (тип) **complex1**, кроме компонентных данных **(real, imag),** включены две компонентные функции **define ( )** и **display ( ).**

Класс – это тип, введенный программистом. Класс "комплексное число" настолько важен, что включен в стандартные библиотеки заголовочным файлом **complex.h.**

Можно определять и описывать объекты класса и создавать производные типы

```
complex1   X1,  X2,  D; // 3 объекта класса complex1
complex  *point = &D; // Указатель на объект класса complex1
complex1   G[ 6 ]; // Массив объектов класса complex1
complex1  &name = 5; // Ссылка на объект класса complex1
```

Каждый класс служит для определения объекта. Для описания объекта используется конструкция

имя_класса имя_объекта ;

Определение объекта класса преусматривает выделение ему участка памяти и деление этого участка на части, соответствующие отдельным элементам объекта, каждый из которых отображает отдельный компонент данных класса. То есть и в объект V1 и в объект G[6] класса complex входит по два элемента типа double, представляющих вещественные и мнимые части комплексных чисел. Как только объект класса определен, можно обращаться к его компонентам с помощью квалифицированных имен

имя_ объекта . имя_класса : : имя_компонента

Но чаще имя класса с указанием области действия : : опускают и обращаются как и в случае структур с помощью уточненного имени

имя_объекта . имя_элемента

Возможности те же что и у структур, например, присвоение значения элементам объектов класса **complex1**

```
X1. real = G [ 6 ] . real = 1.222;
X1. imag = 2.3; G [ 6 ] . imag = 0.0;
```

Уточненное имя принадлежащей классу (компонентной) функции позволит вызвать её для обработки данных именно того объекта, имя которого указано в уточненном имени:

имя_объекта . обращение_к_компонентной_функции

Таким образом можно определять значения компонентных данных для определенных выше объектов, например

X2.define (3.3, 32.0); // Комплексное число 3.3+j*32.0

А вызов принадлежащей классу **complex1** функции **X2. Display ( )** приведет к печати real = 3.3, imag = 32.0.

Доступ к элементам класса возможен и через указатели операцией косвенного выбора компонента (->), например:

Указатель_на_объект_класса->обращение_к_компонентной_функции

Например, товар на складе магазина будем считать классом. Компонентами класса будут:

- название товара,
- оптовая (закупочная) цена,
- торговая наценка,
- функция ввода данных о товаре,
- функция вывода на дисплей сведений о товаре с указанием розничной цены.

Определение класса:

```
// GOODS.CPP – класс “товар на складе”
# include <iostream.h>
struct goods                    // Определение класса “товары”
{
        char name [40];     // Наименование товара
        float price;        // Оптовая цена
        static int percent; // Торговая наценка в %
        // Компонентные функции
        void input ( )      // Ввод данных о товаре
        {
         cout << “Наименование товара:     ”; cin >> name;
         cout << “Закупочная цена:     ”;  cin >> price;
        }
    void display ( )    // Вывод данных о товаре
    {
     cout << “\n ” << name;
     cout << “, розничная цена:    ”;
     cout << long (price * (1.0 + goods : : percent  *  0.01)) ;
    }
} ;
```

Торговая наценка здесь определена как статический компонент класса и нуждается в инициализации, т.е. в конструкции вида

тип_имя_класса : : имя_компонента   инициализатор;

И эта инициализация должна быть размещена сразу после определения класса.

```
int goods : : percent = 10;
void main ( void )
```

```
{
   goods wares [ 5 ] = {  { “Мужской костюм”, 2600},
                          { “Женская кофта”, 1100},
                          { “Калькулятор”, 350}
                       };
int k = sizeof (wares) / sizeof (wares [0]);
cout << “\n Введите сведения о товарах:     \n”;
for (i = 3; i < k; i++) wares [ i ] . input ( );
cout << “\n Список товаров при наценке”  << wares [ 0 ]. percent << “%”;
for ( i = 0 ; i < k; i++) wares [ i ].display ( );
}
```

Результат выполнения программы:

Наименование товара: Мыло <enter>

Закупочная цена: 35 <enter>

Мужской костюм, розничная цена: 2860

Женская кофта, розничная цена: 1210

Калькулятор, розничная цена: 385

Мыло, розничная цена: 38.5

7.2. Конструкторы, деструкторы

Для инициализации объектов класса в его определение можно включать специальную компонентную функцию, называемую **конструктор.** Формат ее определения следующий:

Имя_класса (список_формальных_параметров)
{операторы_тела_конструктора ; }

Имя конструктора совпадает с именем класса, и он автоматически вызывается при определении или размещении в памяти с помощью оператора **new** каждого объекта класса. *Без конструктора для каждого нового создаваемого объекта класса пришлось бы явным образом присваивать значения данным объекта, т.е. переменным.* Конструктор автоматизирует инициализацию вновь создаваемых объектов.

Например, для класса **complex1** конструктор имеет вид

complex1 (double re = 0.0, double im = 0.0)
{real = re; imag = im;}

По умолчанию за счет начальных значений параметров формируется комплексное число с нулевыми действительной и мнимой частями.

Недостаток классов, введенных с помощью служебного слова **struct** , – общедоступность компонент и сокрытие данных внутри объекта не обес-

печивается. Для изменения видимости можно использовать спецификаторы доступа: **private** – собственный, **public** – общедоступный, **protected** – защищенный. Появление любого из этих спецификаторов в тексте определения класса означает, что до конца определения, или до следующего спецификатора все компоненты класса имеют указанный статус. Защищенные **(protected)** компоненты классов нужны при создании иерархии классов.

Использование в качестве ключа при создании классов служебного слова **union** приводит к появлению специфических свойств у формируемых классов. Эти свойства позволяют использовать одни и те же участки памяти для разных целей, т.к. в каждый момент времени исполнения программы объект объединение содержит только один компонент класса, определенного с помощью слова **union.** Все компоненты этого класса также доступны, но доступ можно менять с помощью всё тех же спецификаторов**: private, public, protected.**

*Если в качестве ключа использовалось слово **class,** то все компоненты этого класса являются собственными **(private),** а значит недоступными для внешнего обращения.* Классы, недоступные за пределами определения, используются редко, поэтому доступ к компонентам также можно менять теми же служебными спецификаторами.

Без явного вызова программиста **конструктор всегда автоматически вызывается при определении или создании объекта класса.** При этом используются умалчиваемые значения параметров конструктора. Конструктор не задает тип возвращаемого значения, а превращает фрагмент памяти в объект того типа, который предусмотрен определением класса.

В классе может быть несколько конструкторов, но только один из них с умалчиваемыми значениями параметров. Параметром конструктора не может быть его собственный класс, но может быть ссылка на него. Нельзя получить адрес конструктора. Есть два способа инициализации данных объекта с помощью конструкторов. Первый – это передача значений прямо в тело конструктора – уже показан, а второй предусматривает создание списка инициализаторов данных объекта. Этот список помещается между списком параметров и телом конструктора, как показано ниже:

имя_класа (список_параметров) :
список_инициализаторов_компонентных_данных
{тело_конструктора}

Каждый инициализатор списка относится к компоненту и имеет вид:

имя_компонента_данных (выражение)

Динамическое выделение памяти для объектов класса имеет и свои неудобства. Если объект какого-либо класса формируется как локальный внутри блока, то появляется необходимость возвращения выделенной для

него памяти системе после того, как он перестал существовать. Это обеспечивает специальный компонент класса – деструктор (разрушитель объектов). Его формат таков:

~имя_класса ( ) {операторы_тела_деструктора}

Начинается деструктор всегда со знака тильды ~ , за которым без пробелов помещается имя класса. Обычно в теле деструктора помещают оператор **delete X, Y, [A].** У деструктора нет параметров, и он не возвращает никаких значений.

7.3. Компонентные данные и компонентные функции

При определении класса в его теле определяются, либо описываются данные класса и принадлежащие ему функции. Определение данных аналогично обычному описанию объектов базовых и производных типов, поэтому данные класса считают его элементами. Как обычно описания элементов одного типа объединяют в одном операторе, но не допускается их инициализация, например:

class point (float X, Y, Z; long a, b, c;);

Инициализация не допускается, так как при его определении еще не существует участков памяти, выделенных компонентным данным. Память ведь выделяется не для класса, а только для объектов класса. Для инициализации должен использоваться автоматический или вызываемый конструктор соответствующего класса.

Принадлежащие классу функции имеют полный доступ к его данным, и для обращения к элементу класса из тела компонентной функции достаточно указать его имя.

Непосредственное использование имен компонентов недопустимо. Для обращения к элементу объекта нужно использовать операции выбора компонентов класса (‘ **.** ’ или **->**). Первая позволяет формировать уточненное имя по известному имени объекта

имя_объекта . имя_элемента

Вторая операция даёт обращение к компонентным данным объекта по заданному указателю на объект:

указатель_на_объект -> имя_элемента

Статические компоненты класса

Данные класса тиражируются при каждом определении объекта этого класса. Чтобы компонент не тиражировался, а был в единственном числе, он определяется в программе как статический, например **static int per-**

cent. Когда компонент инициализирован к нему обращаются обычными операциями выбора компонентов‘ **. ’** или **‘->’**. Без имени объекта обычную компонентную функцию вызвать нельзя, а к статической можно обратиться по её имени

имя_класса : : имя_статической_функции

В отличие от обычных компонентных данных статические компоненты класса необходимо описывать и инициализировать вне определения класса как глобальные переменные.

Указатели на компоненты класса

Указатели на компоненты класса .* и ->* при определении не адресуют никакого участка памяти, т.к. память выделяется не классу, а его объектам. Поэтому указатель на класс не адресует никакого конкретного объекта. Возникают вопросы – зачем нужны такие указатели и как они получают значения? Начнем с того, что указатели на компоненты класса поразному определяются для компонентов данных и для компонентов функций. Указатели на принадлежащие классу функции определяются следующим образом:

тип_возвращаемого_функцией_значения (имя_класса : : * имя_ указателя на функцию) (спецификация_параметров_функции);

Например, в классе complex определены компонентные функции (методы) **double& re ( ), double& im ( ).** Вне класса можно описать указатель ptCom

double& (complex : : *ptCom) ( ) ;

А описав, можно почти обычным способом задать его значение

ptCom = &complex : : re;

Теперь для любого объекта **A** класса **complex** можно вызвать принадлежащую классу функцию **re ( ).**

complex A (10.0, 2.4) ; // Определение объекта **A**
(A.*ptCom) ( ) = 11.1; // Изменили вещественную часть **A**
cout << (A.*ptCom) ( ); // Вывод **A . real** на печать
ptCom = &complex : : im; // Настроили указатель на мнимую часть
cout << (A.*ptCom) ( ); // Вывод на печать мнимой части **A.imag**

В примере использован указатель на функцию без параметров, возвращающую значения типа **double&** , и его не настроить на принадлежащие классу **complex** функции с другой сигнатурой.

Указатель на компонентные данные имеет вид

тип_данных (имя_класса : : имя_компонента)

и при этом компонент класса должен быть общедоступным **(public).**

Указатель на компоненты класса можно использовать как фактический параметр при вызове функции

имя_объекта . *указатель_на компонент_данных

имя_объекта . * указатель на компонентную функцию (метод)

Компонентная функция должна обязательно быть описана в теле класса, и в отличие от обычных функций она имеет доступ ко всем компонентам класса с любым статусом доступа. Но если определение (не только прототип) размещено также в теле класса, то компилятор воспринимает функцию как подставляемую (inline) и при каждом вызове код функции встраивается непосредственно в точку вызова. Но подставляемые функции имеют ряд ограничений (не могут быть рекурсивными, не могут содержать циклы, переключатели и т.д.).

Поэтому есть более успешный способ определения принадлежащих классу функций. Внутри тела класса помещается только прототип компонентной функции, а её определение – вне класса, как определение любой другой функции, входящей в программу.

Программист должен сообщить компилятору, к какому именно классу относится функция с помощью бинарной операции **: :** (указания области видимости). Формат применения таков

тип имя_класса : : имя_компонентной_функции (спецификация_ формальных_параметров) {тело_принадлежащей классу функции}

Эта конструкция называется квалифицированным именем компонентной функции и означает, что функция есть компонент класса и лежит в области его действия. А в теле класса помещается её прототип

тип имя_функции (спецификация и инициализация параметров) ;

Пример программы**:**

```
#include <graphics.h> // прототипы графических функций
#include <conio.h> // прототип функции getch ( )
// Описание класса  point
class point   { // класс с именем точка
              public:
              int x,y;
              // Прототипы компонентных функций
              point (int,int ); //прототип конструктора
              int& give_x ();int& give_y () ;  // прототипы функций доступа к
              X и Y
              void show();  // прототип функции
              void move (int,int); //прототип функции
              void hide ( ); // прототип функции
              };
                    // Определение функций класса point
        point :: point (int xi= 0, int yi = 0)  { x = xi ; y = yi; }
        int& point::give_x (void) {return x;}
        int& point :: give_y (void) {return y;}
```

```
        void point :: show (void) {putpixel (x,y,getcolor ( ) );}
        void point :: hide (void) {putpixel (x,y,getbkcolor ( ) );}
        void point :: move (int xn = 0, int yn = 0)
        {   hide ( );
            x =xn; y = yn;
            show ( );
        }
                                    //Работа программы
void main ( )
    {   point A(200,50); // создается невидимая точка A
        point B; // создается невидимая B
point D (500,200); // создается невидимая D
                        // Переменные инициализации графики
        int dr = DETECT, mod;
        initgraph ( &dr, &mod, "C:\\TEMP.bgi");
              A.show (); // Показать A
              getch ( ) ;
              B.show ( ); // показать B
              getch ( ) ;
              D.show ( ); // показать D
              getch ( );
              A.move (100,100 ); // переместить A
               getch ( );
               B.move (50,60); // переместить B
               getch ( );
               closegraph ( ); // Закрыть графический режим
    }
```

7.4. Указатель this

Когда функция, принадлежащая классу, вызывается для обработки, этой функции автоматически и неявно передается указатель на тот объект, для которого она вызвана. Этот указатель тайный, но у него есть имя **this** и он скрытно определен в каждой функции класса

имя_класса * const this = адрес_обрабатываемого_объекта

Слово **this** является служебным, и явно определить или описать этот указатель нельзя и не нужно. Изменять его нельзя, но в каждой функции класса он указывает именно на тот объект, для которого вызывается функ-

ция. *Объект, адресуемый указателем **this,** становится доступным внутри принадлежащей классу функции именно через этот указатель.*

Указателем **this** пользуются, когда в теле принадлежащей классу функции нужно явно задать адрес того объекта, для которого она вызвана. Без указателя **this** трудно обойтись, когда в классе нужна функция, включающая конкретный объект класса в список, поскольку нужно включать в список указатель на тот объект, который сейчас обрабатывается. Это включение должна осуществлять функция компонент класса, но конкретное имя включаемого объекта в момент написания этой функции еще не известно, так как его позже выбирает сам программист.

Когда указатель использован в функции, принадлежащей классу, например **VOSK,** то он имеет по умолчанию тип **VOSK * const** и всегда равен адресу объекта, для которого вызвана компонентная функция. Если в программе для некоторого класса **Y** определить объект

Y faza (4);

то при вызове конструктора класса **Y**, создающего объект **faza**, значением указателя **this** будет **&faza.**

7.5. Друзья классов

При управлении доступом чаще всего используют общедоступные **(public)** компоненты классов. Однако бывают собственные **(private)** и защищенные **(protected)** компоненты классов. Собственные компоненты локализованы в классе и недоступны извне. Дружественными функциями называют такие, которые, не являясь компонентом класса, имеют доступ к его защищенным и собственным компонентам. Для получения таких прав друга функция должна быть описана в теле класса со спецификатором **friend.** При наличии такого описания класс предоставляет права доступа к защищенным и собственным компонентам.

Например:

```
# include <conio.h>
// Класс – "символ в заданной позиции экрана"
class charlocus
{  int x, y; // Координаты свидания на экране
   char cc;    // Символ участника свидания
   friend void friend_put (charlocus *, char); // Прототип дружественной функции для замены символа
charlocus (int xi, int yi, char ci)    // Конструктор
{x = xi; y = yi; cc = ci;}
```

void display (void) // Вывести символ на экран
{**gotoxy (x,y); putch (cc);** // **gotoxy** ( ) помещает курсор в позицию экрана с координатами x и y (в строчном режиме x = от 0 до 79, y = от 0 до 24
// **putch (int s)** выводит на экран в положение курсора указанный символ
}

};

void friend_put (charlocus *p, char c) // Дружественная функция замены символа в объекте;

{ p->cc = c;}

void main (void)

{ **charlocus D (20, 4, 'd ');** // Создать объект

charlocus S (10,10, 's'); // Создать объект

clrscr ( ); // Очистить экран

D. display ( );

getch ( ) ; // Чтение кода из буфера клавиатуры без вывода его на экран (т.е. пауза в исполнении программы, позволяющая видеть смену изображений)

S.display ( );
getch ( );
friend_put (&D, '*');
D.display ( );
getch ( );
friend_put (&S,'#');
S.display ( );
getch ( );
}

Программа последовательно выводит на экран **d** (в позицию 20, 4), затем **s** (в позицию 10, 10), затем * (в позицию 20, 4) и # (в позицию 10,10).

Выполнение программы заключается в следующем: создаются два объекта **D** и S, для которых определяются места на экране и символы (**d, s**). Функция класса **charlocusss : : display ( )** выводит символы в указанные места. Функция **friend_put** заменяет символы объектов на экране, т.е. как дружественная получает доступ к собственным данным класса **charlocus** и изменяет значение символа того объекта, адрес которого будет передан ей как значение первого параметра.

Дружественные функции имеют особенности по применениям:

- не может быть компонентной функцией класса, к которому она дружественная,
- может быть глобальной,
- может быть компонентной другого класса,
- может быть дружественной с несколькими классами,

- класс может быть дружественным другому классу, т.е. все компонентные функции являются дружественными другому классу (но дружественный класс должен быть определен вне тела класса, предоставляющего дружбу).

Механизм дружественных функций упрощает интерфейс между классами.

7.6. Перегрузка стандартных операций

Одна из особенностей C++ состоит в возможности распространения стандартных операций на операнды, не приспособленные для этого. Например, операция + не предназначена для других целей, а мы хотим её использовать для объединения символьного массива – соединения нескольких строк текста в единую строку.

Для распространения действия операции на новые типы данных программист вводит специальную функцию, называемую операция-функция, в таком формате:

класс,_т.е._тип_возвращаемого значения operator знак_операции (спецификация параметров операции функции) {операторы тела операции функции}

При необходимости добавляется и прототип операции-функции

тип_возвращаемого значения operator знак_операции (спецификация параметров операции функции)

Например, для распространения бинарной операции * на объекты класса **Box** можно ввести перегруженную операцию * с заголовком

Box operator * (Box D, Box F)

Если для класса **Box** введена операция-функция с таким заголовком и определены два объекта **H** и **Z** класса **Box,** то это значит, что выражение **H*Z** интерпретируется как вызов функции **operator * (H,Z).**

Если операция-функция определена как принадлежащая классу, то вызвать её можно по имени объекта или указателя на объект и операции выбора компонентов **(->, .).**

Есть ограничения по применению перегрузки операций:

- нельзя вводить собственные обозначения для операций;
- есть операции, не допускающие перегрузок – вот они:
- . прямой выбор компонента структурированного объекта,
- .* обращение к компоненту через указатель на него,
- ?: условная операция,
- :: операция указания области видимости,

- sizeof операция вычисления размера в байтах,
- # препроцессорная операция.

7.7. Наследование классов

Введенный в программу объект призван моделировать свойства и поведение фрагмента какой-либо решаемой задачи, связывая в целое данные и методы, относящиеся к этому фрагменту. Объекты взаимодействуют между собой с помощью сообщений, передающих информацию. В ответ на сообщение объект делает какое-то действие, предусмотренное набором компонентных функций своего класса. Объекты одного класса имеют разные имена, но одинаковые по типам и данным. Т.е. класс выступает в роли типа, позволяющего вводить нужное количество объектов, имена которых задает программист.

Объекты разных классов и сами классы могут наследоваться, т.е. находиться в состоянии иерархии. Иерархия позволяет создавать новые классы (производные или порожденные) на основе имеющихся (базовых или порождающих).Производные классы получают наследство – данные и методы своих базовых классов, и могут пополняться собственными методами и данными. Наследуемые компоненты не перемещаются в производный класс, а остаются в базовом. Сообщение, обработку которого не смогли выполнить методы производного класса, автоматически передается в базовый. Если для обработки нужны данные, которых нет в производном классе, то их незаметно от программиста ищут в базовом классе.

Если класс-точка на экране считать базовым, то на его основе можно построить класс-окно на экране. Данными этого класса будут две точки:
- точка, определяющая левый верхний угол
- точка смещения вдоль координатных осей, определяющая размеры окна.

Методы класса-окно на экране:
сместить окно вдоль оси **X** на **DX;**
сместить окно вдоль оси **Y** на **DY**;
сообщить значение координаты **X** левого верхнего угла;
сообщить значение координаты **Y** левого верхнего угла;
сообщить размер вдоль оси **X;**
сообщить размер вдоль оси **Y.**

Конструктор окна на экране:
создать окно на экране с заданным именем по двум точкам, определяющим левый верхний угол окна и его размеры.

Деструктор окна:

уничтожить окно с заданным именем.

Две точки различно используются в классе “окно на экране”. Если у первой координаты **(3,4),** а у второй **(0,0),** то это окно с нулевыми размерами – пустое окно. Наименьшее окно с размерами в 1 пиксель (1,1), независимо от положения левого верхнего угла.

При наследовании некоторые имена методов (компонентных функций) могут быть по-новому определены в производном классе. Для доступа к этим методам в базовом классе в производном используется операция уточнения области видимости **::**.

Любой производный класс может стать базовым для других классов и так формируется иерархия классов. У объектов появляется возможность доступа к данным всех своих базовых классов. Допускается множественное наследование, т.е. возможность наследовать компоненты никак не связанных классов. Например, класс-окно на экране и класс-сообщение формируют новый класс объектов – сообщение в окне.

Важную роль играет статус доступа, так как вне классов доступны только компоненты со статусом **public** – они глобальны. Собственные **private** методы доступны только внутри класса, где они определены. Защищенные **protected** компоненты доступны внутри своего класса и во всех производных классах.

На доступность компонентов влияет не только явное использование спецификаторов **public, private, protected**, но и выбор ключевого слова при определении класса – **class, struct, union**.

В описании и в определении производного класса приводится список базовых классов, из которых он наследует данные и методы. Между именем вводимого производного класса и списком базовых ставится двоеточие

class F : C, B, X {…};

Клас **F** порожден классами **C, B, X** и наследует их компоненты. Наследования не будет, если его имя будет использовано в качестве имени компонента в определении производного класса. В порожденном классе компоненты из базовых со статусом **public** и **protected** получат статус **private**, если новый класс определен словом **class**, и статус **public**, если определен словом **struct**. Явно изменить статус доступа при наследовании можно с помощью спецификаторов **private, public и protected**. Конструктор базового класса всегда вызывается и выполняется до конструктора производного класса.

Например, класс **point** (точка) и его производный класс **spot** (пятно). Наследуемые компоненты класса **point:**

int x,y – координаты точки на экране;

point ( ) – конструктор;

givex ( ), givey ( ) – доступ к координатам точки;

show ( ) – изобразить точку;

move ( ) – переместить точку.

Дополнительно вводится радиус пятна **rad** и его видимость на экране (когда ее нет, **vis ==0**, **и vis ==1**, когда есть видимость) и новые методы: **hide( )** – убрать изображение с экрана, **vary( )** – изменить масштаб изображения на экране, **circle( )** – нарисовать окружность с центром в точке с координатами **(X,Y), floodfill (X,Y,c)** – закрасить ограниченную область, которой принадлежит точка **(X, Y),** цветом, определенным параметром **c**. Конструктор класса **spot( )** имеет три параметра – координаты центра пятна **(Xi, Yi)** и радиус пятна **(Ri).**

Для освобождения памяти используют деструкторы

~ имя_класса

Деструкторы нельзя наследовать, т.е. они не передаются из базового. Вызовы деструктора для объектов класса выполняются неявно, то есть без программиста, но когда при создании объекта ему выделялась память, то необходим явный вызов деструктора. Явное определение деструктора имеет вид

~ spot ( ) { hide ( ); tag = 0; delete [ ] pspot;}

Действия деструктора: убрать с экрана пятно **hide,** установить в ноль признак наличия в памяти битового образа **tag,** освободить память, связанную с указателем **pspot**

Деструкторы не наследуются, а создаются компилятором по умолчанию со статусом доступа **public**, вызывая деструкторы базовых классов. В примере с классами **point и spot** это будет

public: ~spot ( ) { ~point ( ) ; }

В любом классе в качестве компонентов могут быть другие классы и после уничтожения охватывающего класса происходит деструкция охваченного класса, т.е. деструкторы выполняются в порядке, обратном перечислению классов в определении производного класса.

Класс называют прямой базой, если он входит в список базовых при определении класса. Но для производного класса могут быть не прямые предшественники, которые служат базой для классов, входящих у него в список базовых. Косвенное наследование создает иерархию классов. Пусть класс **А** есть базовый для **Б, а Б** есть база для **С.** Тогда **Б** есть непосредственный базовый для **С,** а класс **А** непрямой базовый для **С.** Обращение к элементу **Xa,** входящему в класс **А** и унаследованному последовательно классами **Б и С** можно обозначить как

А :: Xa или как **Б :: Xa**. И в том, и в другом случае обращение будет к элементу **Xa** класса **А.**

А – прямая база для Б;

↑

Б – производный от А класс и прямая база для С;

↑

С – производный класс с прямой базой Б и косвенной А.

А так как класс может порождаться из нескольких базовых, то возникает иерархия, в которой производные классы принято изображать ниже базовых, например:

Class X1{…};
Class X2{…};
ClassX3 {…};
ClassY : public X1, public X2, public X3 { … };

Наличие нескольких прямых базовых классов называют множественным наследованием. Для создания производного класса сначала определяют базовые, например, таким образом:

```
//circ.cpp//определение класса окружность
#include<graphics.h>
class circ
{ int xc,yc,rc; // координаты центра и радиус
public:
//конструктор
circ ( int xi, int yi, int ri )
    { xc = xi; yc = yi; rc = ri;}
//изобразить на экране окружность
void show ( )
    {circle (xc,yc,rc);}
//убрать изображение с экрана
void hide ( )
        { int bk, cc ;
        bk = getbkcolor ( ); //цвет фона
        cc = getcolor( ); //цвет изображения
        setcolor (bk); //сменить цвет рисования
        //рисуем цветом фона:
        circle (xc,yc,rc) ;
        //,:
        setcolop (cc);
        }
};
```

Аналогично определяем класс **square** (квадрат)

```
// square.cpp
#include<graphics.h>
class sqare
{ int xq,yq,lq; // координаты центра и длина стороны
//вспомогательная функция рисования:
```

```
        void rissquare (void)
            {int d = lg2*ri;
            line (xq-d, yq-d, xq+d, yq-d);
            line (xq-d,yq+d, xq+d, yq+d);
            line (xq-d, yq-d, xq-d, yq+d);
            line (xq+d, yq-d, xq+d, yq+d);
            }
        public: square ( int xi, int yi, int li)
        {xq =xi; yq = yi; lq =li; }
        void show ( )
    {rissquare ( ); }
    void hide ( )
            { int bk, cc;
            bk = getbkcolor ( );
            cc = getcolor ( ) ;
            setcolor (bk);
            rissquare ( );
            setcolor (cc);
            }
    };
```

На основании этих базовых **circ** и **square** создаем производный класс – окружность в квадрате с именем **circsqrt**:

```
        #include <conio.h>
        #include “circ.cpp”
        #include “square.cpp”
        // производный класс circsqrt без наследуемых данных
        class circsqrt : public circ, public sqare
        { public:
        // конструктор с пустым оператором в теле
        circsqrt (int xi, int yi, int ri) : circ(xi,yi,ri), square (xi,yi,2*ri) {   }
        //изобразить окружность в квадрате:
        void show (void)
                { circ : : show ( ); square : : show ( ); }
        //  убрать окружность с экрана
        void hide ( )
              { square : : hide ( ); circ : : hide ( ); }
        };
        void main ( )
           { int dr = DETECT, mod;
           initgraph (&dr, &mod, “C:\\ Temp.bgi”);
           circsqrt A1(100,100,60) ;
```

```
circsqrt F4 (400,300,50) ;
A1.show ( ); getch ( );
F4.show ( ); getch ( );
F4.hide ( ) ; getch ( );
A1.hide ( ) ; getch ( );
closegraph ( );
}
```

Определения базовых классов должны предшествовать их использованию, поэтому их описываем вначале, а потом описываем **circsqrt.** Выполнение его конструктора сводится к последовательному вызову конструкторов базовых классов **circ** и **square.** Длину сторон квадрата выбрали равной двум радиусам окружности, чтобы окружность была вписанной. В основной программе формируется два объекта **A1** и **F4** класса **circsqrt.** Они последовательно выводятся на экран и в обратном порядке убираются с экрана, как показано на рис.3.

При множественном наследовании никакой класс не может больше одного раза использоваться как прямой базовый, но может многократно быть непрямым базовым классом.

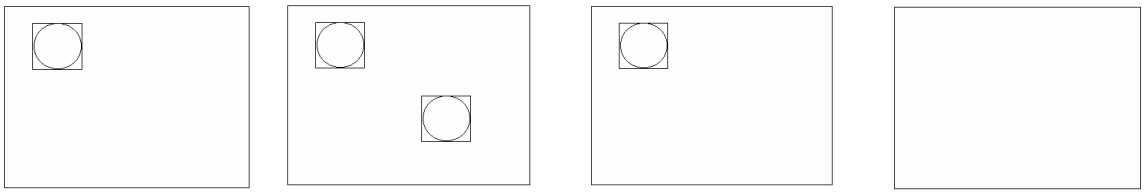

Рис.3. Операции на экране монитора

Если есть несколько объектов класса **X**, то возможно дублирование и включение в производный объект нескольких объектов базового класса, как показано на рис.4.

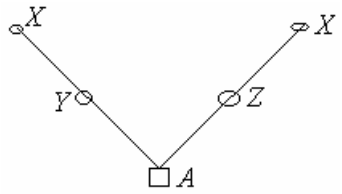

Рис.4. Объект из группы объектов базового класса

Чтобы устранить дублирование, нужно либо четко указывать обращение к конкретному компоненту класса X , используя полную квалификацию **A :: Y :: X :: f ( )** или **A :: Z :: X :: f ( ) ;** либо для устранения дублирования непрямого базового класса при множественном наследовании

этот класс объявляется виртуальным. Для этого в списке базовых классов перед именем класса ставится ключевое слово **virtual.** Например:

Class X { … f ( ) ; …};
Class Y : virtual public X { … } ;
Class Z : virtual public X { … } ;
Class A : public Y, public Z { … } ;

Теперь класс **A** будет включать только один экземпляр **X,** доступ к которому равноправно имеют классы **Y и Z,** как показано графом на рис.5.

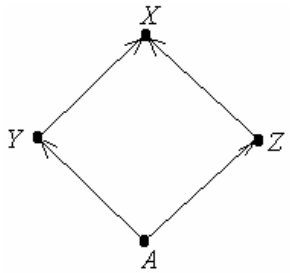

Рис.5. Граф доступа классов

Для устранения неоднозначности при множественном наследовании все же лучший способ – использование квалифицированных имен компонентов.

7.8. Виртуальные функции и абстрактные классы

Когда функция должна по-разному выполняться в производных классах, обращаются к механизму виртуальных функций. При этом в базовый класс помещается основа функции, а конкретика помещается в производные классы. Например, в базовый класс помещается функция описания изображения на экране, а в производные классы – её формы и размеры (куб, треугольник, эллипс и т.д.).

Классы, включающие виртуальные функции, называют полиморфными. Например, в базовом классе figure определена компонентная функция **void show ( ).** Так как внешний вид фигуры в базовом классе не определен, то в каждый из производных требуется включить свою функцию **void show ( )** для формирования изображений на экране. После этого доступ к функции **show ( )** производного класса возможен либо с помощью явного указания области видимости

имя_производного_класса : : show ( )

или с использованием имени конкретного объекта

имя_объекта_производного_класса . show ( )

В обоих случаях выбор нужной функции выполняется при написании текста программы и не изменяется после компиляции.

Если же при описании функции в базовом классе использовать спецификатор **virtual**, то в производных классах эту функцию можно определять по другому (т.е. подменять). После того как функция определена как виртуальная, её повторное определение в производном классе с тем же прототипом создает новую виртуальную функцию уже и без спецификатора **virtual**, и с ним.

При подмене виртуальной функции нужно помнить о необходимости полного совпадения сигнатур, имен и типов возвращаемых значений в базовом и производных классах. Виртуальная функция может объявляться как дружественная (**friend**) в другом классе.

Чистой (пустой) виртуальной функцией называют функцию с таким определением

virtual тип_имя_функции (список параметров) = 0;

Класс, в котором есть хоть одна чистая виртуальная функция, называют абстрактным. Чистая виртуальная функция ничего не делает и недоступна для вызова. Она служит основой для подменяющих её функций в производных классах, и абстрактный класс может использоваться только как базовый для производных классов. Пусть есть абстрактный класс **B**

```
class B { protected : virtual void func (char) = 0;
          void bos (int); }
```

На основе класса B можно по-разному создать производные классы

```
Class D : public B {
…
void func (char);
};
class E : public B {
void bos (int);
};
```

В классе **D** чистая виртуальная функция **func ( )** заменена конкретной виртуальной функцией того же типа. Функция **B : : bos ( )** наследуется классом **D** и доступна в нем и его методах. Класс **D** не абстрактный.

В классе **E** переопределена функция **B : : bos ( ),** а виртуальная **функция B : : func ( )** унаследована.

Механизм абстрактных классов введен для представления общих понятий, которые в дальнейшем предполагается конкретизировать. Особенно это важно в графике, где часто приходится использовать различные геометрические фигуры.

У абстрактных классов по сравнению с обычными ограниченные права, т.к. невозможно создать объект абстрактного класса и их нельзя использовать при явном приведении типов.

Если класс определен внутри блока или другого класса, то его называют **локальным**. Локализация предполагает недоступность его компонентов вне области определения класса (вне тела функции или блока, где он описан). Компонентные функции локальных классов могут быть только встроенные (inline). Например, при определении класса **квадрат** внутри него используется локальный класс – **отрезок.** Исходными данными отрезка будут координаты концов, и из четырех отрезков можно строить квадрат как изображение четырех отрезков.

Можно создавать шаблоны родственных функций и классов, определяющие неограниченное количество родственных функций и классов.

Шаблон функций имеет вид

template <список_параметров_шаблона> определение_функции

Шаблон класса –

template <список_параметров_шаблона> определение класса

Шаблон семейства классов определяет способ построения отдельных классов, аналогично тому, как класс определяет правила построения и формат отдельных объектов. Определение шаблона может быть глобальным.

Контрольные вопросы

1. Дайте определение понятию класс в C++.
2. Чем классы отличаются от структур?
3. Что может являться компонентами класса?
4. Что такое конструкторы и деструкторы?
5. Приведите пример использования конструкторов.
6. Приведите пример использования деструкторов.
7. Что такое статические компоненты класса?
8. Приведите пример использования указателя this.
9. Дайте определение дружественных компонентов класса.
10. Перечислите особенности применения дружественных функций.
11. Приведите пример перегрузки стандартных функций.
12. Что такое наследование классов?
13. Приведите пример виртуальных функций.

Глава 8. ВВОД-ВЫВОД В С++

8.1. Потоковый ввод-вывод

Библиотека ввода-вывода разрабатывалась позднее языка С++, не входит в сам язык и построена на основе механизма классов. Связь библиотеки ввода-вывода с компилируемой программой осуществляет заголовочный файл iostream.h – input, output steam (поток – последовательность байтов символов). Используемые в программах потоки делятся на 3 типа: входные (для чтения информации), выходные (для ввода данных) и двунаправленные, осуществляющие и то и другое.

Потоки принято делить на стандартные, консольные (используются только для среды MC-DOS, поддерживаются классом constream и обеспечивают удобный доступ к терминалу. Имеют возможность работы с клавиатурой, всем экраном и его частями), строковые и файловые. Стандартные и консольные потоки служат для передачи данных от клавиатуры к экрану. Если символы в основной памяти образуют строку, то это строковый поток. Если данные на внешних носителях (например на диске), то это файловый поток, или просто файл.

Библиотека потоковых классов основана на двух классах – **ios и streambuf**, который обеспечивает буферизацию данных во всех используемых производных классах. Методы и данные класса **streambuf** программист обычно не использует, т.к. класс **ios** содержит указатель на него. Наиболее активно используются классы:

Ios – базовый потоковый класс;
Istream – класс входных потоков;
Ostream – класс выходных потоков;
Iostream – класс двунаправленных потоков ввода-вывода;
Istrstream – класс входных строковых потоков;
Ostrstream – класс выходных строковых потоков;
Strstream – класс двунаправленных строковых потоков;
Ifstream – класс входных файловых потоков;
Ofstream – класс выходных файловых потоков;
Fstream – класс двунаправленных файловых потоков;
Constream – класс консольных потоков.

Потоковые классы, их данные и методы становятся видны в программе, если в них включены нужные заголовочные файлы с расширением .h .

Iostream.h для классов **ios, istream, ostream, stream**;
Strstrea.h для классов **istrstream, ostrstream, strstream**;
Fstream.h для классов **ifstream, ofstream, fstream**;

Constrea.h для класса **constream**.

Так как класс **ios** является базовым, то включение в программу любого из заголовочных файлов **strstrea.h, constrea.h, fstream.h** автоматически подключает к программе файл **iostream.h.**

8.2. Стандартные потоки ввода-вывода

Заголовочный файл **iostream.h** содержит определения стандартных потоков ввода-вывода;

cin – объект класса **istream**, связанный со стандартным входным потоком, обычно с клавиатурой;

cout – объект класса **ostream**, связанный со стандартным буферизированным потоком, обычно с дисплеем;

cerr – объект класса **ostream,** связанный с небуферизированным выходным потоком, обычно дисплеем, на который посылаются сообщения об ошибках;

clod – объект класса **ostream,** связанный с буферизированным выходным потоком, в котором посылаются на дисплей сообщения об ошибках.

Каждый раз при включении в программу **iostream.h** происходит формирование объектов **cin, cout, cerr, clod**. В этом же файле описаны классы **istream, ostream** и для них определены оригинальные операции. Операция ввода класса **istream** называется извлечением (чтением) данных из потока и обозначается >>. Операция вывода класса **ostream** называется вставкой, или записью данных в поток и обозначается <<.

Эти операции выполняются, если слева от них находятся объекты соответственно классов **iostream** и **ostream**

cin >> имя_объекта_базового_типа;
cout << выражение базового типа;
cerr << выражение базового типа;
clod << выражение базового типа.

Операция **>>** заключается в преобразовании последовательности символов потока в значение типизированного объекта, частным случаем которого является переменная базового типа (**int, float, long и т.д.).** Операция << выполняет обратные действия. Фактически выполняется перекодировка – двоичные числа преобразуются в символы на экране и наоборот.

Особенности выполнения операций >> << в том, что они имеют невысокий приоритет и запись **cout << 3+7+6;** выведет на экран 16. Чтобы вывести в поток значения выражения с более низким рангом, требуются скобки, например:

cout << (a+b < c); или, например, условная операция

Выражение1 ? выражение 2 : выражение 3

cout << (X <0 ? –X : X);

Если выражение 1 истинно, то выполняется выражение 2, если выражение 1 ложно (равно 0), то выполняется выражение 3. Без скобок не обойтись, т.к. и ? и : имеют ранг ниже <<.

Операция включения << возвращает ссылку на тот потоковый объект, что слева от выражения. Поэтому удобно писать, применяя выражение как имя для результата

cout << “\n x*3 = ” << x*3;

Ввод с клавиатуры **cin >> i >> j >> k >> h;** лучше выполнять построчно, т.е. ввели i, enter, ввели j, enter и т.д., хотя можно и в строку через пробельные символы.

При вводе-выводе целых чисел есть ограничения на длину внешнего и внутреннего представления.

Если **int i; cin >> i; cout << ” \ni = “ << i;** и набрать на клавиатуре **123456789,** то на экране будет – **13035**.

Длинную последовательность целых чисел нужно вводить так:

long q; cin >> q; cout << “\n q = ” << q;

и на экране будет **123456789**.

Следует помнить о границах возможностей разных типов. Определив float pi; при наборе на клавиатуре **3,14159265358979323 85** на экране увидим только **3.141593**. Если вещественное число мало и выходит за рамки разрядной сетки, например**, 0.00000000000000000000001**, на экран выведется **1.0 e-23**. И набирать можно **1.0e-23.**

Можно вводить цифры в восьмеричной и шестнадцатеричной форме, а вывод будет в десятеричной.

8.3. Форматирование данных при обмене с потоками

Умалчиваемые форматы при выводе данных не всегда удобны. Например при выполнении операторов **int i1 = 1, i2 = 2, i3 =3, i4 = 4;**

Cout << “\n” << i1 << i2 << i3 << i4; приведет к результату **1234**

Можно ввести пробелы **cout << “\n” << i1 << ‘ ’<<i2 << ‘ ’ << i3 << ‘ ’ << i4;** и результат будет более наглядным **1 2 3 4**

Форматы представления выводимой информации могут изменяться программистом с помощью флагов форматирования из базового класса ios, реализованных из фиксированных битов чисел типа **long** (и поэтому изменяемых с помощью логических выражений), и манипуляторов. Манипуля-

торы – это специальные функции, позволяющие изменять состояние флага и потока. Их используют в качестве правого операнда операций ввода-вывода >> или <<.

Например, манипулятор **hex** позволяет установить шестнадцатеричное основание счисления вводимых в поток **cout** числовых значений:

cout << “\n Десятичное число :” << 15 << hex;

cout << “\n Шестнадцатеричное число:” << 15;

На экране получим:

Десятичное число: 15

Шестнадцатеричное число: 0xF

Манипуляторы **dec** – десятичный, **hex** – шестнадцатеричный, **oct** – восьмеричный изменяют основание системы счисления до следующего явного изменения. Манипуляторы **ws** действуют только при вводе и извлекают пробелы; **endl** действуют при выводе, включая символ новой строки и сбрасывая буфер этого потока; **ends** – при выводе включают нулевой признак конца строки; **flush** – при выводе сбрасывают буфер выходного потока.

Потоки ввода-вывода **cin (C-input)** и **cout (C-output)** не требуют сложностей для работы, а требуют только специальных операторов **>>** и **<<**. Запись инструкции **cout << ВАНЯ** напишет на экране **ВАНЯ**. Запись **cin >> number** позволит ввести с клавиатуры значение переменной **number**. Синтаксис прост и смущают обычно слова ввод и вывод. Все, что набираем на клавиатуре, – это ввод, а все, что подаем на экран, – это вывод. Стрелки операторов показывают, что делать **cout << F** , то есть от переменной к объекту. И **cin >> number**, т.е. от объекта к переменной.

Заголовочный файл **#include <iostrean.h>** содержит объявления и определения для объектов **cin** и **cout**. Потоки **cin, cout** сами знают, как преобразовывать данные при вводе-выводе. Непосредственно можно использовать типы данных **int, char, float, double**. Эти правила заранее определены для различных типов данных в виде функций, определения которых находятся в библиотеке ввода-вывода, а прототипы размещены в заголовочном файле **iostream.h.**

Потоковый вывод имеет синтаксис из объекта **cout** , оператора **<<** и переменной, или константы, например, **cout << number;** Если выводятся значения нескольких переменных, то перед каждой из них свой оператор вывода **<<**. Это позволяет выводить и наименование переменной, а затем и её саму, например:

cout << “Скорость = ” << 90км/ч даст запись **Скорость = 90 км/ч.**

Потоковый ввод строится из объекта **cin,** оператора **>>** и переменной, которой присваивается вводимое значение, например **cin >> k;** Одной

инструкцией можно вводить значения нескольким переменным, например **cin >> i >> k >> m;**

Чтобы использовать операции обмена **>>** **<<** с данными произвольных типов, необходимо расширить действие этих операций, введя новые операции – функции, например с именем **operator <<.** Формат перегрузки – ссылка на объект потокового класса (тип **istream& или ostream&).**

ostream& operator << (ostream& out, новый_тип имя) (операторы для параметров нового типа). Здесь новый_тип – тип определенный программистом.

Например, если определен класс

struct point {float x; float y; float z;}

то для типа **point** можно определить правила вывода

ostream& operator << (ostream& t, point d)

{return t << " \nx = " << d.x << "y = " << d.y << "z = " << d.z ; }

Результатом операции **t << "\n x = "** – является ссылка на объект типа **ostream.** Эта ссылка используется в качестве левого операнда при выводе **<< d.x** и т.д. Т.е. операция-функция **operator ( )** позволит выводить значения объектов типа **point,** определенного программистом.

Следует помнить, что класс **ostream** и поток **cout,** – стандартные режимы выполнения операции ввода – вывода, определенные в заголовочном файле **iostream.h,** который нужно поместить в начале текста программы до текста операции-функции **operator << ( ).**

Для перегрузки (расширения действия) операции ввода необходимо определить операцию-функцию вида:

```
istream& operator >> (istream& in, новый_тип& имя)
{// Любые операторы для параметров нового типа
in >> … // Ввод значений нового типа
return in; //Возврат ссылки на объект класса istream
}
```

8.4. Манипуляторы

Для форматирования выводимых данных используют манипуляторы из заголовочного файла **iomanip.h (endL** (начало новой строки), **ends** (вставка символа нуль в поток вывода), **flush** (очистка потока) и др., которые действуют до переопределения. Например, при выводе цены нет необходимости считать более двух разрядов после запятой, т.е. необходимо округление. Его достигают манипулятором **setprecision** и, чтобы выводить цену с двумя десятичными разрядами, необходима запись

float price;
cout << setprecision (2);
…
cout << price;

Манипулятор **setw** определяет ширину поля вывода, а пустые позиции занимают пробелы. Манипулятор **setfill** позволяет включать вместо пробелов в пустые места любой другой символ. Например, при выдаче чека в долларах США используют символ *

cout << “US$” << setfill (*) << value;

Манипулятор **setiosflags и resetiosflags** используют для установки флагов:

skipws (игнорирование разделителей в потоке ввода) – обычно установлен по умолчанию;

left (выравнивание влево);

right (выравнивание вправо);

showpoit (отображение десятичных позиций и точки);

scientific (научный формат, т.е. отображение чисел с плавающей точкой);

fixed (формат с фиксированной точкой).

Правила такие: флаг устанавливают с помощью манипулятора **setiosflags,** аргумент которого состоит из символов **ios ::** и устанавливаемого флага. Например: **cin >> resetiosflags (ios : : skipws);** позволяет считывать разделители в потоке ввода.

Работа с файлами аналогична работе с клавиатурой, но объектами **cin** и **cout** уже не обойтись, т.к. файлов для ввода может быть несколько, а объект **cin** всего один. Файлы имеют имена, которые им дают после объявления их в классе **fstream**.

Когда файлы объявлены как объекты класса **fstream,** задать поток ввода и вывода можно также как для объектов **cin и cout**. Например:

```
#include <fsream.h>
void main ( )
{
fstream myfile;
myfile . open (“List.txt”, ios :: out);
myfile << “ВлГУ – это клёво!”;
myfile . close ( );
}
```

В этой программе создан файл с именем **list.txt,** и в него записана строка **ВлГУ – это клёво!**

Объявление файлов класса **fstream** выглядит просто: **fstream идентификатор;**

После объявления файла с ними выполняют операции с помощью функций класса **fstream.** Наиболее важные функции **open ( ), close ( ), eot ( ).**

Доступ к файлу: **open (char имя_файла [ ], int режим доступа);**

Функция получает два аргумента: строку символов (имя файла) и флаг, определяющий режим доступа к файлу, который может быть таков:

Флаг **ios :: out** открывает файл для вывода (запись в файл);

Флаг **ios :: in** открывает файл для ввода (считывание из файла);

Флаг **ios :: app** открывает файл для добавления записи в конец предыдущей информации;

Флаг **ios :: nocreate** открывает файл, только если он существует (этот флаг создан только для компиляторов **Microsoft,** в которых при попытках открыть несуществующий файл создается новый. В компиляторах **Borland** этого нет).

Функция **close ( )** без параметров служит для закрытия файлов и сохранения данных на диске. Функция **eof ( )** расшифровывается как **end of file** и служит для обозначения конца считывания файла, т.к. объем данных файла заранее не известен. Она возвращает **1**, если в процессе считывания данных переходим за границу файла.

8.5. Функции для обмена с потоками

Кроме операций извлечения (чтения из потока) >> и записи в поток << есть и альтернативные средства для обмена с потоками. При выводе основной класс **ostream** и ему принадлежат (т.е. в нём определены) две функции для двоичного вывода данных:

ostream& ostream : : put (char cc);

ostream& ostream : : write (const signed char *array, int n);

ostream& ostream :: write (const unsigned char *array, int n);

Функция **put ( )** помещает в вызванный выходной поток символ, использованный в качестве фактического параметра, т.е. следующие операторы эквивалентны:

cout << ‘ Z ’; cout.put (‘ Z ’);

Функция **write ( )** имеет два параметра – указатель **array** на участок памяти, из которого выводятся данные и целое число n, определяющее количество выводимых из этого участка символов (байт).

Функции **put ( )** и **write ( )** в отличие от << не обеспечивают форматирование выводимых данных, и флаги форматирования к этим функциям не применимы.

Так как эти функции возвращают ссылки на объект того класса, для которого вызваны, то можно организовать цепочку вызовов

char ss[ ] = “Лень замечательное свойство, но лениться думать недопустимо!”;

cout.put (‘\n’. write (ss, sizeof (ss)-1).put(‘!’).put (‘\n’);

На экране (в потоке **cout**) появится – **Лень замечательное свойство, но лениться думать недопустимо!**

Функций, подобных **put ( ) и write ( )**, в классе i**stream** достаточно много (6 перегруженных функций **get ( )** – для извлечения последовательности байтов из входного потока и переноса их в символьный массив или в буфер, функции **peek (), ignore ( ), read ( ), seek ( ), tell ( )** для просмотра, игнорирования или чтения определенных символов из потока).

Строковые потоки (обмены в основной памяти)

Классы **istrstream, ostrsteam, strstream** предназначены для создания потоков, связанных с основной памятью и определяются в заголовочном файле **strstream.h**. В компиляторах MS-DOS в заголовочных файлах буква **m** пропускается, чтобы было не больше 8 символов **(strstrea.h).** В программах это обычно символьные массивы, и в обозначении таких потоков слово **str – string** – строка, и объекты таких классов называют строковыми потоками. Определяются они конструкторами такого вида:

имя класса имя потока (параметры конструктора);

Имя класса – это одно из имен **istrstream, ostrstream,strstream** , имя потока – это идентификатор выбираемый программистом, типы параметров различны для разных классов.

Входные строковые потоки создаются конструкторами класса **istrstream,** обязательным параметром которого является указатель **str** на уже существующий участок основной памяти.

istrstream имя потока (char *str);

Например, строковый поток с именем **inBuf**, связанный с участком памяти, выделенным ранее для символьного массива **buf [ ]**, определяется как

char buf [54];

istrstream inBuf (buf) ;

И после такого определения строковый поток **inBuf** может использоваться как левый операнд операции извлечения **>>.** Извлечение информации из строкового потока с помощью операции >> выполняется от первого непробельного символа до ближайшего пробельного символа. Если необходимо читать и пробельные символы, то используют функции бесформатного обмена **get ( )** и **getline ( ),** позволяющие организовать копирование строк.

Выходные строковые потоки создаются с помощью конструктора класса **ostrstream**

ostrstream имя потока (char *str, int len, int mode) ;

Необязательное имя потока выбирается программистом, указатель **str** адресует уже существующий участок памяти, параметр **int len** определяет размер этого участка, параметр **int mode** – индикатор режима обмена, который определяет размещение информации в связанной с потоком строке. Конкретный режим обмена задается с помощью флагов:

ios : : out – для записи информации с начала строки (обычно устанавливается по умолчанию);

ios : : ate – для записи после признака конца строки **‘\0’** – запись в продолжении строки;

ios : : app – действует аналогично **ios : : ate,** но возможны модификации.

Функция **write ( )** применительно к выходному потоку позволяет записывать в него данные без форматирования, т.е. строка записывается вместе с пробельными символами и символом конца строки **\0.**

Двунаправленные строковые потоки имеют конструктор как для чтения, так и для записи

strstream имя потока (char *buf, int lenBuf, int mode);

***buf** – указатель на участок памяти, **int lenBuf** – размер в байтах участка памяти, **int mode** – индикатор режима обмена (**mode**-режим), в качестве которого используют флаги класса **ios – ios : : in и ios : : out** – определяющие направление обмена и флаги **ios: : ate и ios : : app** – влияющие на размещение указателя позиции чтения/записи в буфере.

Строковые потоки используют по-разному. С их помощью в участок памяти можно заносить разнотипную инфомацию и затем извлекать её по необходимым правилам, а можно строковый поток сделать внешним и использовать его для межмодульного обмена данными.

8.6. Работа с файлами

Библиотека C++ имеет средства для работы с последовательными файлами, т.е. теми, у которых чтение или запись производится байт за байтом от начала к концу. В каждый момент позиции в файле, откуда осуществляется чтение или запись, определяются значениями указателей позиций записи и чтения файла. Позиционирование осуществляется или автоматически, или за счет явного управления, и для этой цели в C++ есть средства. Библиотека C++ позволяет:

- создавать файлы,
- создавать потоки,
- открывать файлы,

- присоединять файлы к потоку,
- обмениваться с файлом с помощью потока,
- отсоединять поток от файла,
- закрывать файлы,
- уничтожать файлы.

Есть альтернативные варианты выполнения этих действий.

Создание файла можно выполнить на нижнем уровне с помощью библиотечной функции

int creat (const char *path, int amode) ;

Функция **creat ( )** по заданному имени файла **path** создает новый файл или очищает и подготавливает для работы уже существующий. Прототип этой функции в заголовочном файле **io.h** Параметр **amode** нужен только для нового создаваемого файла.

Потоки для работы с файлами создаются как объекты классов:

ofstream – для вывода (записи) данных в файл,

ifstream – для ввода (чтения) данных из файла,

fstream – для чтения и записи данных.

Чтобы использовать эти классы, необходимо в текст программы включить заголовочный файл **fstream.h** и после этого можно определять файловые потоки соответствующих этим классам типов. Создав файловый поток, можно прикрепить его к конкретному файлу компонентной функцией **open ( ),** которой можно и открывать файл. Формат функции:

void open (const char *fileName, int mode = умалчиваемое значение, int protection = умалчиваемое значение) ;

Первый параметр – имя существующего или организуемого файла (строка в формате операционной системы). Второй параметр **mode** (режим) – определяет режим работы с открываемым флагом (например, только запись или только чтение). Флаги определены следующим образом:

enum ios : : open_mode

{
in = 0x01, // открыть только для чтения;
out = 0x02, // открыть только для записи;
ate = 0x04, // при открытии искать конец файла;
app = 0x08, // дописывать данные в конец файла;
trunc = 0x10, // создать новый файл вместо существующего;
nocreate = 0x20, // не открывать новый файл (для несуществующего функция open даст ошибку);
noreplace = 0x40, // не открывать существующий файл (для выходного без режимов ate или app выдать ошибку);
binary = 0x80, // открыть для двоичного (нетекстового обмена)
};

Третий параметр **protection** означает защиту и устанавливается обычно по умолчанию. Вызов функции **open (0)** обычно осуществляют с помощью точного имени

имя потока . open (имя файла, режим, защита) ;

Здесь имя потока – это имя одного из объектов классов **ofstream, ifstream, fstream.** По умолчанию второй параметр класса **ofstream** устанавливается **ios : : out,** т.е. файл открывается только для вывода.

Для проверки удачности выполнения функции **open ( )** применяется перегруженная операция **!,** примененная с именем потока, например **!inFile.** Если результат не нулевой, то имеется ошибка

```
if ( !inFile)
{cerr << "Ошибка при открытии файла!\n";
exit (1);}
```

При работе с библиотечными классами ввода-вывода чаще всего используют конструкторы или без параметров, или с явным заданием имени

ifstream f1; // создает входной поток **f1** без параметров

ifstream flow1 ("file.1");// создает входной поток **flow1** для чтения данных, разыскивая файл с именем **file.1** и если он существует, то конструктор аварийно завершает работу.

ofstream flow2 ("file.2");// создается выходной поток с именем **flow2** для записи информации. Если **file.2** не существует, то он будет создан, открыт и соединен с потоком **flow2.** Если существует, то предыдущий вариант будет удален и пустой файл создается заново.

Все файловые классы наследуют функцию **close ( ),** позволяющую очищать буфер, отсоединять поток от файла и закрывать файлы. Автоматически эта функция вызывается по завершению программы, а для изменения режимов работы её нужно явно вызывать.

Основное требование к исключению – известность в точке формирования **(throw)** и в точке обработки **(catch).**

Исключения дают возможность программисту динамически проводить обработку возникающих ситуаций, с которыми не может справиться исполняемая функция. Механизм исключений позволяет вынести анализ и обработку ситуации и любое количество требуемой информации из точки возникновения ситуации в другое место программы, специально предназначенное для подобных обработок. Обработка исключений позволяет определять процедуры обработки ситуаций , которые будут выполняться при продолжении программы.

Следует помнить, что механизм исключений работает только с синхронными событиями, т.е. событиями, возникающими в результате работы самой программы. Например, прерывание работы программы **(CTRL+C)** не является синхронным событием.

Если в контролируемом блоке формируется исключение, то среди обработчиков ищется соответствующий этому исключению обработчик и управление передается ему, а после обработки – в точку окончания последовательности обработчиков (т.е. возврата в контролируемый блок нет). Если исключение создано, но соответствующий ему обработчик отсутствует, то автоматически вызывается библиотечная функция **terminate ( ),** которая завершает выполнение программы.

Исключение воспринимается (захватывается) обработчиком, если тип исключения совпадает или может быть приведен стандартным образом к типу формального параметра обработчика.

Процедура обработки исключений должна быть помещена непосредственно после контролируемого блока и может обрабатывать только одно исключение заданного или преобразуемого к нему типа, который указан в спецификации её параметров. После выполнения процедуры обработки программа продолжится с той точки, где кончается список всех процедур обработки исключений.

Для явного выхода из процедуры обработки можно применить оператор **goto,** однако им нельзя воспользоваться для передачи управления обратно – в процедуру обработки исключений или в контролируемый блок.

Дополнительные возможности по обработке исключений обеспечивают только последние версии компиляторов C++, в которых добавлен механизм динамической идентификации типов, описаны классы, содержащие объекты для работы с исключениями при выделении динамической памяти, функции, глобальные переменные и классы поддержки механизма исключений. Например, функция обработки неопознанного исключения или опции динамического определения типов.

Контрольные вопросы

1. Приведите классификацию потоков данных.
2. Перечислите стандартные потоки данных в C++.
3. Приведите примеры потоковых операций.
4. Приведите пример форматирования данных при потоковом выводе.
5. Что такое манипуляторы в C++?
6. Приведите пример использования манипуляторов.
7. Перечислите альтернативные средства для обмена с потоками.
8. Приведите пример использования функции put и write.
9. Перечислите средства C++ для работы с файлами.
10. Приведите пример чтения/записи из файла.

Заключение

В соответствии с программой учебного курса в настоящем учебном пособии приведены теоретические основы наиболее популярного на сегодняшний день языка программирования C++. Рассмотрены общие методы написания программ и наиболее важные вопросы практического применения языка программирования для решения прикладных задач. Кратко изложены введение в программирование на C++, лексика, выражения и основные операторы языка C++. Показана работа с указателями, адресами, массивами, функциями, ссылками, структурами, объединениями и другими конструкциями языка программирования C++. Особое внимание уделено работе с классами, как наиболее яркой отличительной особенности языка C++.

Материал пособия ориентирован на читателей, имеющих базовую подготовку по информатике, и содержит достаточно полные сведения по языку программирования C++. Все приведенные в данном пособии сведения по языку программирования C++ иллюстрируются примерами действующих программ.

Библиографический список

1. **Подбельский, В.В.** Язык C++ / В.В. Подбельский. – М.: Финансы и статистика, 2003. – 560 с. – ISBN 5-279-02204-7.

2. **Культин, Н.Б.** C/C++ в задачах и примерах / Н.Б.Культин. – СПб.: БХВ-Петербург, 2004. – 288 с. – ISBN 5-94157-029-5.

3. **Франка, П.** C++: учеб. курс / П. Франка. – СПб.: Питер, 2002. – 528 с. – ISBN 5-314-00136-5.

4. **Хижняк, П.Л.** Пишем вирус и антивирус / П.Л. Хижняк. – М.: ИНТО, 1991. – 90 с. – ISBN 5-86028-011-4.

5. Информатика и вычислительная техника: учеб. пособие / под ред. В.Н. Ларионова. – М.: Высш. шк., 1992. – 287 с. – ISBN 5-27902-202-0.

6. Информатика: учеб. пособие для студентов пед. вузов / под ред. Е.К. Хеннера. – М.: Академия, 2001. – 816 с. – ISBN 5-76950-330-0.