ХАРЬКОВСКИЙ
ИНСТИТУТ
ИНФОРМАЦИОННЫХ
ТЕХНОЛОГИЙ

Учебный курс

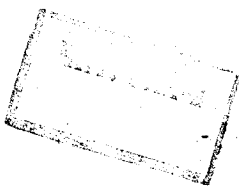
С.В.Глушаков • С.В.Смирнов • А.В.Коваль

ПРАКТИКУМ ПО C++

FOLIO • www.bookpost.com.ua

С.В.Глушаков
С.В.Смирнов
А.В.Коваль

ПРАКТИКУМ ПО C++



Харьков
ФОЛИО
2006

ББК 32.97
Г 55

Серия «Учебный курс»
основана в 2002 году

Шеф-редактор
С. В. Глушаков

Художник-оформитель
С. И. Правдюк

ISBN 966-03-2661-0

Введение

В начале 80-х годов сотрудник фирмы AT&T Bell Laboratories Бьярн Страуструп (Bjarne Stroustrup) разработал язык программирования C++. Этот язык был построен на базе языка программирования C и включал объектно-ориентированные конструкции, такие как классы, производные классы и виртуальные функции, заимствованные из языка simula67. Целью разработки C++ было «ускорить написание хороших программ и сделать этот процесс более приятным для каждого отдельно взятого программиста» (Б.Страуструп. «Язык программирования C++». М., Наука, 1991).

Название C++, которое придумал Рик Маскити (Rick Mascitti) в 1983 году, отражает факт происхождения этого языка от C (вначале он назывался «C с классами»). С момента своего появления C++ получил широкое распространение и завоевал признание многих тысяч профессиональных программистов. В 1989 году Бьярн Страуструп опубликовал вместе с Маргарет Эллис «Справочное руководство по языку C++», послужившее основой для разработки проекта стандарта ANSI C++, разработанного комитетом ANSI X3J16. В начале 90-х годов к работе этого комитета подключился комитет WG21 Международной организации по стандартизации (ISO), и была начата работа по созданию единого стандарта ANSI/ISO C++. Результатом этой работы стал стандарт International Standard for Information Systems – Programming Language C++ (ISO/IEC JTC1/SC22/WG21), опубликованный в начале 1998 года. Большинство новейших компиляторов C++ сейчас в большей или меньшей мере соответствуют этому стандарту. И нет сомнения в том, что в ближайшее время все компиляторы этого языка будут приведены в соответствие стандарту.

Данное руководство по C++ будет полезно студентам, изучающим этот язык, начинающим программистам, желающим повысить свою квалификацию, и профессионалам, которые хотят получить справку по конкретному вопросу.

Помимо языка C++ в книге описаны некоторые важнейшие функции языка C, определенные в стандарте ANSI C, а также Unicode-строки и ставшая составной частью языка стандартная библиотека шаблонов (STL).

Первая половина книги содержит материал, общий для языков C и C++, вторая часть посвящена объектно-ориентированному программированию.

В данном издании большое внимание уделяется практическому применению C++. Каждый раздел содержит множество примеров и практикум, включающий задания для анализа и самостоятельной проработки.

Из-за ограничений, связанных с верстkой, в ряде случаев в листингах программ приходилось делать разрывы строк. В каждом таком случае продолжение разорванной строки начинается с символа ↳.

Авторы выражают надежду, что предлагаемая вниманию читателей книга окажет существенную помощь в изучении языка.

Контактные телефоны: (0572) 21-28-72, (057) 715-69-09
Адрес: 61050, г. Харьков, пл. Фейербаха, 7, кабинет 2.201
Адрес для переписки: 61050, Харьков-50, а/я 9436
E-mail: *info@xiit.kharkov.ua*

Раздел 1
Типы данных C++

Как и любой язык программирования, C++ поддерживает различные типы данных, с помощью которых программисту предоставляется возможность оперирования с некоторым ограниченным набором простейших математических объектов. Предлагаемый раздел поможет Вам ознакомиться с основными конструкциями языков C и C++.

Тема 1.1
Структура программы

Как известно, любая *программа* представляет собой некую последовательность инструкций машинного кода, управляющих поведением определенного *вычислительного средства*. Это может быть работа с дисплеем, средствами хранения информации, звуковыми устройствами системы, внешними устройствами (клавиатурой, мышью, принтером, модемом) и т.д. Для упрощения процесса разработки *программного обеспечения* (ПО) создана не одна сотня языков программирования. Каждый из них имеет сильные и слабые стороны и призван решать ряд определенных задач.

Все существующие средства программирования можно разделить на две основные категории:

- языки программирования низкого уровня;
- языки программирования высокого уровня.

К первой группе относят семейство языков Ассемблера (например, Turbo Assembler, Macro Assembler). Эти средства разработки программного обеспечения позволяют получить наиболее короткий и быстродействующий код (разумеется, при условии грамотного использования всей мощи, предоставляемой операционной системой). Однако следует отметить, что процесс программирования на языке низкого уровня – занятие весьма кро-

потливое, утомительное и занимает гораздо больше времени, чем при использовании языка высокого уровня. Кроме того, программы, написанные на Ассемблере, достаточно тяжелы для восприятия, вследствие чего вероятность возникновения ошибок в них значительно выше.

В свою очередь, этих недочетов лишены языки программирования высокого уровня, к которым относится и C++. Вместе с тем, данной группе языков присущи недостатки другого рода, например такие, как значительное увеличение размера и времени выполнения исполняемого кода. Связано это с тем, что при написании программ на языке высокого уровня результирующий машинный код генерируется из исходного текста компилятором и в исполняемом модуле может образовываться «балласт», состоящий из функций и процедур подключаемых библиотек, которые могут неэффективно использоваться самой программой. Другими словами, компилятор самостоятельно принимает решения (зачастую неоптимальные) по подключению библиотечных функций.

Кроме того, следует отметить, что при компиляции программы с языка высокого уровня существует так называемая неоднозначность результирующего кода. Ведь если в Ассемблерных программах каждая инструкция преобразуется в машинный код однозначно, то программа, написанная на языке высокого уровня (и содержащая набор операторов, функций, процедур и т.д.), может компилироваться по-разному в зависимости от версии используемого компилятора и конкретной реализации библиотек функций.

История появления языка C++ берет начало с 1972 года, когда Деннисом Ритчи и Брайаном Керниганом был разработан язык программирования C, сочетающий в себе возможности языков высокого и низкого уровня, позднее утвержденный Американским Национальным Институтом Стандартизации (ANSI).

В 1980 году благодаря стараниям Бьярна Страуструпа на свет появился прямой потомок ANSI C – язык C++, вобравший в себя положительные черты еще нескольких языков программирования. Необходимо отметить, что C++, в отличие от C, позволяет программисту разрабатывать программы (или приложения) с использованием как традиционного структурного, так и объектно-ориентированного подхода.

Программирование на C++ включает такие ключевые понятия языка, как идентификаторы, ключевые слова, функции, переменные, константы, операторы, выражения, директивы препроцессо-

ра, структуры, массивы и ряд других элементов. Все они будут описаны в данной книге.

Рассмотрим элементарную, ставшую классической, программу на C++, результат работы которой – вывод на экран строки Hello, World!

Первая строка приведенного на рис. 1.1 листинга (директива препроцессора #include...) подключает заголовочный файл iostream.h, содержащий объявления функций и переменных для потокового ввода/вывода. Имя подключаемого модуля указывается в косых скобках (< > – заголовочный файл находится в каталоге \INCLUDE\ конкретной среды разработки) либо в кавычках (" " – файл находится в том же каталоге, где и включающий его модуль разрабатываемой программы с расширением *.c или *.cpp).

Далее следует описание единственной в примере функции main().

Надо отметить, что любая программа на C++ обязательно включает в себя функцию main(), с которой и начинает свое выполнение.

```
#include <iostream.h>                  Подключение заголовочного
                                       файла
int main()                             Описание главной функции
{                                      Начало блока
  cout << "Hello, World!\n";           Вывод строки символов
  return 0;                            Возврат из функции
}                                      Конец блока
```

Рис. 1.1. Пример программы *Hello, World*

Ключевое слово int указывает на то, что по завершении своей работы функция main() вернет операционной системе целочисленное значение. Помимо этого, в скобках могут быть указаны параметры командной строки, обрабатываемые в программе.

Тело самой функции содержит оператор консольного вывода последовательности символов cout << и оператор возврата из функции return.

В отличие от ANSI C, в C++ для организации консольного ввода/вывода применяются операции >> и <<, известные в C как правый и левый сдвиг соответственно, хотя, безусловно, допустимо использование традиционных функций языка C. Как будет показано в дальнейшем, данные операции в C++ по-прежнему

выполняют сдвиги бит в переменных, однако их возможности расширены за счет перегрузки операций.

Существуют стандартные потоки для ввода информации с клавиатуры, вывода данных на экран, а также для вывода в случае возникновения ошибки. Помимо этого, приложения поддерживают работу со стандартным потоком вывода на печать и дополнительным консольным потоком. В общем случае каждый из перечисленных потоков может быть представлен как некоторый виртуальный файл (байт-поток), закрепленный за определенным физическим устройством. Стандартный поток ввода/вывода может быть переопределен с тем, чтобы вывод, например, осуществлялся не на экран, а в заданный файл (перенаправление ввода-вывода).

В C++ стандартный поток ввода связан с константой cin, а поток вывода – с константой cout (для использования этих констант подключается заголовочный файл **iostream.h**). Таким образом, для вывода информации в стандартный поток используется формат

```
cout << выражение;
```

где *выражение* может быть представлено переменной или некоторым смысловым выражением.

Например:

```
int variable = 324;
cout << variable; // вывод целого
```

Для консольного ввода данных используют формат записи:

```
cin >> переменная;
```

При этом переменная служит приемником вводимого значения:

```
int Age;
cout << "Введите Ваш возраст: ";
cin >> Age;
```

Таким образом, переменная Age принимает введенное с консоли целое значение. Ответственность за проверку соответствия типов вводимого и ожидаемого значений лежит на программисте.

Одновременно (через пробел) можно вводить несколько значений для различных переменных. Ввод заканчивается нажатием клавиши Enter. Если введенных значений больше, чем ожидается в программе, часть вводимых данных останется во входном буфере.

В случае если в приемник должна быть введена строка символов, ввод продолжается до первого символа пробела или ввода Enter:

```
char String[80];
cin >> String;
```

Так, при вводе строки "Да здравствует C++!" переменная String воспримет только подстроку "Да". Остальная часть строки останется в буфере до тех пор, пока в программе не встретится следующий оператор ввода.

Ниже показано, как ввод одной строки с символом пробела "12345 67890" разделяется и заполняет две совершенно разные переменные – String1 и String2.

```
#include <iostream.h>
int main()
{
  char String1[80];
  char String2[80];
  cout << "Input string: ";
  cin >> String1;
  cout << String1 << '\n';
  cout << "Input string: ";
  cin >> String2;
  cout << '\n' << String2 << '\n';
  return 0;
}
```

В результате работы программы получим, например:

```
Input string: 12345 67890
12345
Input string:
67890
```

Как видно из примера, строка символов вводилась лишь один раз, хотя в программе оператор ввода встречается дважды. При этом первые пять вводимых символов были помещены в переменную String1, а последующие символы – в переменную String2.

Тема 1.2
Комментарии

При создании программного продукта разработчик всегда ясно представляет, что будет выполнять та или иная часть его программы. Однако, если программа довольно сложная (большой размер исходного текста, нетривиальность алгоритма), через какое-то время бывает трудно вспомнить всю цепь логических рассуждений, предшествующих написанию кода. Особенно сложно бывает разобраться в текстах программ других разработчиков.

Чтобы избежать подобных неприятностей, в процессе составления программного кода используются так называемые комментарии. Текст комментариев всегда игнорируется компилятором, но позволяет программисту описывать назначение какой-либо части программы. Размер комментариев ограничен только размером свободной памяти, хотя, конечно, не стоит превращать текст программы в литературное произведение.

В C++ используется две разновидности комментариев.

Первый, традиционный (заимствованный из ANSI C) многострочный комментарий, представляет собой блок, начинающийся с последовательности символов «слеш со звездочкой» (/*) и заканчивающийся символами «звездочка слеш» (*/). Как следует из названия, данный вид комментария может располагаться на нескольких строках. Комментарии этого типа не могут быть вложенными друг в друга.

Второй вид – однострочный комментарий – следует за «двойным слешем» (//) до конца текущей строки. Этот тип комментария может быть вложенным в многострочный комментарий.

Кроме объяснения текста программы комментарии можно использовать для временного исключения из программы некоторой ее части. Этот прием обычно используется при отладке.

Например:

```
int main()
{
  // примеры комментария
  int a = 0; // int d;
  /*int b = 15;*/
  int c = 7;
  /* <- начало комментария
  a = c;
  конец комментария -> */
  return 0;
}
```

В приведенном примере компилятор проигнорирует объявление переменной b и d, а также присвоение переменной a значения переменной c.

Тема 1.3
Переменные и типы данных

Суть фактически любой программы сводится к вводу, хранению, модифицированию и выводу некоторой информации. Авторы книги рекомендуют обратиться к книге «Программирование в среде Win-

dows. Учебный курс» издательства «Фолио», в которой более подробно описаны вопросы переменных, понятия алгоритмов и т.д.

Для того чтобы программа могла на протяжении своего выполнения сохранять определенные данные и оперировать с ними, используются *переменные* и *константы*.

Одним из базовых свойств программы является *идентификатор*.

Под *идентификатором* понимается имя переменной или константы, имя функции или метка. В программе идентификатор может содержать прописные и строчные латинские буквы, цифры и символ подчеркивания, обязательно начинается с буквы или символа подчеркивания и не должен совпадать с ключевым словом (с учетом регистра). Так, в приведенном выше примере представлены идентификаторы a, b, c и d.

Следующим бызовым понятием любого языка программирования является *ключевое слово*.

Ключевые слова – это зарезервированные языком идентификаторы, имеющие специальное назначение. В табл. 1.1 приводится список ключевых слов языка C++.

Таблица 1.1
Ключевые слова

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

Чтобы переменную можно было использовать в программе, она должна быть предварительно объявлена. При этом в процессе объявления переменной осуществляется создание ее идентификатора.

Переменная – объект программы, занимающий в общем случае несколько ячеек памяти, призванный хранить данные. Переменная обладает именем, размером и рядом других атрибутов (таких как видимость, время существования и т.д.).

При объявлении переменной для нее резервируется некоторая область памяти, размер которой зависит от конкретного типа пе-

ременной. Здесь следует обратить внимание на то, что размер одного и того же типа данных может отличаться на компьютерах разных платформ, а также может зависеть от используемой операционной системы. Поэтому при объявлении той или иной переменной нужно четко представлять, сколько байт она будет занимать в памяти ЭВМ, чтобы избежать проблем, связанных с переполнением и неправильной интерпретацией данных.

Ниже приведен перечень базовых типов переменных и их размер в байтах. Следует учесть, что размер, указанный в табл. 1.2 для каждого типа, должен быть проверен для конкретного ПК.

Таблица 1.2

Базовые типы данных для ПК на базе платформы Intel

Тип	Размер, байт	Значение
bool	1	true или false
unsigned short int	2	от 0 до 65 535
short int	2	от -32 768 до 32 767
unsigned long int	4	от 0 до 4 294 967 295
long int	4	от -2 147 483 648 до 2 147 483 647
int (16 разрядов)	2	от -32 768 до 32 767
int (32 разряда)	4	от -2 147 483 648 до 2 147 483 647
unsigned int (16 разрядов)	2	от 0 до 65 535
unsigned int (32 разряда)	4	от 0 до 4 294 967 295
char	1	от 0 до 256
float	4	от 1.2e-38 до 3.4e38
double	8	от 2.2e-308 до 1.8e308
void	2 или 4	-

Объявление переменной начинается с ключевого слова, определяющего его тип, за которым следует собственно имя перемен-

ной и (необязательно) инициализация – присвоение начального значения.

Одно ключевое слово позволяет объявить несколько переменных одного и того же типа. При этом они следуют друг за другом через запятую (,). Заканчивается объявление символом точка с запятой (;).

Имя переменной (идентификатор) не должно превышать 256 символов (разные компиляторы накладывают свои ограничения на количество распознаваемых символов в идентификаторе). При этом важно учитывать регистр букв (Abc и abc – не одно и то же)! Конечно, имя должно быть достаточно информативным, однако не следует использовать слишком длинные имена, так как это приводит к опискам.

Хотя начальная инициализация и не является обязательной при объявлении переменной, все же рекомендуется инициализировать переменные начальным значением. Если этого не сделать, переменная изначально может принять непредсказуемое значение.

Установка начального значения переменной осуществляется с помощью оператора присваивания (=).

Рассмотрим подробно основные типы переменных.

Переменная типа bool занимает всего 1 байт и используется, прежде всего, в *логических* операциях, так как может принимать значение 0 (false, ложь) или отличное от нуля (true, истина). В старых текстах программ вы можете встретить тип данных BOOL и переменные этого типа, принимающие значения TRUE и FALSE. Этот тип данных и указанные значения не являлись частью языка, а объявлялись в заголовочных файлах как unsigned short, с присвоением начальных значений 1 и 0 соответственно. В новой редакции C++ bool – самостоятельный, полноправный тип.

Часто в программе бывает необходимо указать, что переменная должна принимать только целые значения. *Целочисленные* переменные (типа int, long, short), как следует из названия, призваны хранить целые значения, и отличаются только размером допустимого значения. Целочисленные переменные могут быть *знаковыми* и *беззнаковыми*.

Знаковые переменные могут представлять как положительные, так и отрицательные числа. Для этого в их представлении один бит (самый старший) отводится под знак. В отличие от них, *беззнаковые* переменные принимают только положительные значения. Чтобы указать, что переменная будет беззнаковой, исполь-

зуется ключевое слово unsigned. По умолчанию целочисленные переменные считаются знаковыми (signed, чаще всего опускается; используется при *преобразовании типов данных*).

Из табл. 1.2 видно, что переменная типа int в разных случаях может занимать в памяти различное число байт.

Символьный тип данных char применяется, когда переменная должна нести информацию о коде ASCII. Этот тип данных часто используется для построения более сложных конструкций, таких как строки, символьные массивы и т.д. Данные типа char также могут быть знаковыми и беззнаковыми.

Для представления *чисел с плавающей запятой* применяют тип данных float. Этот тип, как правило, используется для хранения не очень больших дробных чисел и занимает в памяти 4 байта: 1 бит – знак, 8 бит – экспонента, 23 бита – мантисса.

Если вещественное число может принимать очень большие значения, используют переменные *двойной точности*, тип double.

Переменная типа void не имеет значения и служит для согласования синтаксиса. Например, синтаксис требует, чтобы функция возвращала значение. Если не требуется использовать возвращенное значение, перед именем функции ставится тип void.

Приведем несколько примеров объявления переменных:

```
int a = 0, A = 1;
float aGe = 17.5;
double PointX;
bool bTheLightIsOn = false;
char LETTER = 'Z';
void MyFunction();     // возвращаемое функцией
                       // значение игнорируется
```

Тема 1.4
Константы

Константы, так же как и переменные, представляют собой область памяти для хранения данных, с тем лишь отличием, что значение, присвоенное константе первоначально, не может быть изменено на протяжении выполнения всей программы. Константы бывают *литеральными и типизованными*, причем литеральные константы делятся на *символьные, строковые, целые и вещественные.*

Символьные константы представляются отдельным символом, заключенным в одинарные кавычки (апострофы): 'e', '@', '<'.

Строковые константы – это последовательность символов, заключенная в двойные кавычки: "Это пример не самой длинной строковой константы!".

Целые константы бывают следующих форматов:

- десятичные;
- восьмеричные;
- шестнадцатеричные.

Десятичные могут быть представлены как последовательность цифр, начинающаяся не с нуля, например: 123; 2384.

Восьмеричные константы – последовательность восьмеричных цифр (от 0 до 7), начинающаяся с нуля, например: 034; 047.

Шестнадцатеричный формат констант начинается с символов 0x или 0X с последующими шестнадцатеричными цифрами (0...9, A...F), например: 0xF4; 0X5D. Буквенные символы при этом могут быть представлены в как в нижнем, так и в верхнем регистре.

Длинные целые константы, используемые в переменных типа long, определяются латинской буквой l или L сразу после константы без пробела: 36L, 012L, 0x52L.

Вещественные константы – числа с плавающей запятой могут быть записаны в десятичном формате (24.58; 13.0; .71) или в экспоненциальной форме (1e4; 5e+2; 2.2e-5, при этом в мантиссе может пропускаться целая или дробная часть: .2e4).

Типизованные константы используются как переменные, значение которых не может быть изменено после инициализации.

Типизованная константа объявляется с помощью ключевого слова const, за которым следует указание типа константы, но, в отличие от переменных, константы всегда должны быть инициализированы.

Рассмотрим небольшой пример:

```
// Объявление переменной
int i;
// Инициализация переменной i
// литеральной целочисленной константой 25600
i = 25600;

// Теперь объявим типизованную
// строковую константу MyCatName
// и инициализируем ее литеральной строковой константой
const MyCatName[] = "Рудик";
```

Символьные константы в C++ занимают в памяти 1 байт и, следовательно, могут принимать значения от 0 до 255 (см. табл.1.2). При этом существует ряд символов, которые не отоб-

ражаются при печати, – они выполняют специальные действия: возврат каретки, табуляция и т.д., и называются *символами escape-последовательности*. Термин «escape-последовательность» ввела компания Epson, ставшая первой фирмой, которая для управления выводом информации на своих принтерах стала использовать неотображаемые символы. Исторически сложилось так, что управляющие последовательности начинались с кода с десятичным значением 27 (0x1B), что соответствовало символу «Escape» кодировки ASCII.

Escape-символы в программе изображаются в виде обратного слеша, за которым следует буква или символ (см. табл. 1.3).

Таблица 1.3

Символы escape-последовательности

Символ	Описание
\\	Вывод на печать обратной черты
\'	Вывод апострофа
\"	Вывод при печати кавычки
\?	Символ вопросительного знака
\a	Подача звукового сигнала
\b	Возврат курсора на 1 символ назад
\f	Перевод страницы
\n	Перевод строки
\r	Возврат курсора на начало текущей строки
\t	Перевод курсора к следующей позиции табуляции
\v	Вертикальная табуляция (вниз)

В качестве примера использования типизованных и литеральных констант вычислим значение площади круга по известному значению радиуса:

```
#include <iostream.h>
int main()
{
  const double pi = 3.1415;
  const int Radius = 3;
  double Square = 0;
  Square = pi * Radius * Radius;
// Выведем вычисленное значение
```

```
// и осуществим перевод строки ('\n')
  cout << Square << '\n';
  return 0;
}
```

В начале главной функции программы объявляются две константы: pi и Radius. Значение переменной Square изменяется в ходе выполнения программы и не может быть представлено как константа. Поскольку значение радиуса задано явно и в тексте программы не предусмотрено его изменение, переменная Radius объявлена как константа.

Тема 1.5
Перечисления

При использовании большого количества логически взаимосвязанных целочисленных констант удобно пользоваться *перечислениями*.

Перечисления имеют вид:

```
enum Name
{
   item1[=def],
   item2[=def],
   ...
   itemN[=def]
};
```

где
enum – ключевое слово (от *enumerate* – перечислять),
Name – имя списка констант,
item1...itemN – перечень целочисленных констант,
[=def] – необязательный параметр инициализации.

Предположим, нам необходимо в программе описать работу светофора. Известно, что его цвет может принимать лишь 3 значения: красный (RED), желтый (YELLOW) и зеленый (GREEN). Для обработки полученных от светофора сигналов заведем три константы с такими же именами – RED, YELLOW и GREEN, проинициализировав их любыми неповторяющимися значениями с тем, чтобы в дальнейшем проверять, какой из этих трех цветов горит.

Например, мы могли бы записать:

```
const int RED = 0;
const int YELLOW = 1;
const int GREEN = 2;
```

Используя перечисления, то же самое можно сделать в одну строку:

```
enum COLOR {RED, YELLOW, GREEN};
```

Константы перечисления обладают следующей важной особенностью: если значение константы не указано, оно на единицу больше значения предыдущей константы. По умолчанию первая константа имеет значение 0.

То же перечисление можно было проинициализировать другими значениями:

```
enum COLOR {RED=13, YELLOW=1, GREEN};
```

При этом константа GREEN по-прежнему имеет значение 2.

Тема 1.6
Преобразования типов

В C++ существует явное и неявное *преобразование типов*.

В общем случае *неявное преобразование типов* сводится к участию в выражении переменных разного типа (так называемая арифметика смешанных типов). Если подобная операция осуществляется над переменными базовых типов (представленных в табл. 1.2), она может повлечь за собой ошибки: в случае, например, если результат занимает в памяти больше места, чем отведено под принимающую переменную, неизбежна потеря значащих разрядов.

Для *явного преобразования* переменной одного типа в другой перед именем переменной в скобках указывается присваиваемый ей новый тип:

```
#include <iostream.h>
int main()
{
int Integer = 54;
float Floating = 15.854;
Integer = (int) Floating;   // явное преобразование типов
cout << "New integer: ";
cout << Integer << '\n';
return 0;
}
```

В приведенном листинге после объявления соответствующих переменных (целочисленной Integer и вещественной Floating) производится явное преобразование типа с плавающей запятой (Floating) к целочисленному (Integer).

Пример неявного преобразования:

```
#include <iostream.h>
int main()
{
int Integer = 0;
float Floating = 15.854;
Integer = Floating; // неявное преобразование типов
cout << "New integer: ";
cout << Integer << '\n';
return 0;
}
```

В отличие от предыдущего варианта программы, в данном случае после объявления и инициализации переменных осуществляется присваивание значения переменной с плавающей Floating целочисленной переменной Integer.

Результат работы обеих программ выглядит следующим образом:

```
New integer: 15
```

То есть произошло отсечение дробной части переменной Floating.

Практикум «Типы данных C++»

Упражнение 1.1 Работа со стандартными потоками ввода/вывода

Программа, представленная в следующем листинге, предлагает пользователю ввести с клавиатуры некоторые данные с последующим их выводом на экран. При этом в переменную InputInt1 будет занесено первое вводимое значение, а в InputInt2 – второе.

Попробуйте вначале ввести значение 123 456, а затем – значение 123 456 789. Обоснуйте полученные результаты. Обратите внимание на то, какие заголовочные файлы подключаются.

Листинг описанной программы имеет следующий вид:

```
#include <iostream.h>
int main()
{
   int InputInt1, InputInt2;
```

```
    cout << "Введите через пробел два целых числа:\n";
    cin >> InputInt1;
    cin >> InputInt2;
    cout << "Переменная InputInt1:\n";
    cout << InputInt1;
    cout << "\n";
    cout << "Переменная InputInt2:\n";
    cout << InputInt2;
    return 0;
}
```

Упражнение 1.2
Комментарии в теле программы

В представленном ниже листинге рассматриваются виды комментария в C++. Основная задача данного примера – иллюстрация различных целей применения комментария.

В самом начале программы следует многострочный комментарий, описывающий цели и задачи реализуемого модуля. Сразу за объявлением переменной x следует однострочный комментарий, описывающий предназначение данной переменной. Далее следует закомментированное объявление переменной y, по какой-либо причине исключенное из участия в вычислительном процессе программы.

Последний комментарий в приведенном ниже листинге иллюстрирует вариант вложенных комментариев. Он также исключает из вычислительного процесса участие переменной y.

```
/*
Программа, демонстрирующая
различные виды комментария в C++
*/
#include <iostream.h>
int main()
{
    int x;          // Пройденный путь
    //int y = 5; // Закомментированное объявление
    x = 0;
/*
// Закомментированное присвоение (вложенный комментарий)
        x = y;
*/
    cout << x;
    return 0;
}
```

Упражнение 1.3
Переменные различных типов данных

В представленном ниже листинге Вам предлагается проанализировать получаемый результат каждого отдельного присвоения. Следует обратить внимание на то, какие значения может принимать та или иная переменная. При этом рекомендуется руководствоваться табл. 1.2 теоретической части занятия.

```
#include <iostream.h>
int main()
{
  unsigned short x;
  char a,
  b='\n';     // Символ конца строки
  bool bVar = true;
  long z;
  a = -195; // a примет значение 256-195=61 т.е. символ '='
  x = a;     // x примет значение 61
  z = -65537;
  cout << x << b << a << b << bVar << b << z;
  x = z;  // x примет значение -z по модулю 65536, т.е. 65535
  cout << b << x;
  float f = -3.56;
  cout << b << f;
  return 0;
}
```

Упражнение 1.4
Константные выражения

В данном задании Вам предлагается исправить ошибку, возникающую при компиляции программы.

```
#include <iostream.h>
int main()
{
  const float pi = 3.1414;
  const int hex = 0x3512;
  const short oct = 0152;
  const char CRLF = '\n';
  const char *Hello = "Bond..., James Bond!";
  const long Lng = 51342L * 4;
  const float fl = pi + 1.8e-4;
  fl = 0;
  cout << Hello << CRLF << fl;
  return 0;
}
```

Упражнение 1.5
Применение перечислений

Проанализируйте предлагаемый ниже пример, иллюстрирующий использование перечислений. По умолчанию переменная Left принимает значение, равное 0, а переменные Center и Right – значения 1 и 2 соответственно. Таким образом, присвоение переменной Current величины 1 приведет к тому, что надпись "Text align program" будет выведена в центре строки экрана.

```
#include <iostream.h>
int main()
{
   enum TextAlignment {Left,Center,Right};
     char *Text = "Text  align  program";
   char *Spaces = "                    ";
   short Current = 1;
   if(Current == Left)
      cout << Text;
   else if(Current == Center)
      cout << Spaces << Text;
   else if(Current == Right)
      cout << Spaces << Spaces << Text;
   return 0;
}
```

Упражнение 1.6
Явное и неявное преобразование типов данных

Несоответствие типов данных при работе в C++ является одной из наиболее часто встречаемых ошибок на начальном этапе обучения программированию. К сожалению, компилятор не всегда может предотвратить подобные недочеты программиста. Как следствие, программа выполняет не те действия, которые от нее ожидает программист. Проанализируйте предлагаемый ниже пример, иллюстрирующий неявное преобразование типов данных, когда переменной типа short присваивается значение переменной типа long с неизбежной потерей разрядов числа:

```
#include <iostream.h>
int main()
{
   long LVal = 613216458;
   short ShVal = LVal;      // потеря значащих разрядов
   cout << ShVal;
```

```
    return 0;
}
```

Напротив, явное преобразование зачастую позволяет представить данные в удобном для программиста виде. Приведенный ниже пример показывает, как явное преобразование типов можно использовать для получения изображения символа по его порядковому номеру:

```
#include <iostream.h>
int main()
{
    for(int i=33; i<128; i++)
        cout << (char)i;        // явное преобразование
    return 0;
}
```

Раздел 2
Выражения и операторы

Как и любой язык программирования, C++ поддерживает различные конструкции для манипулирования данными. Сюда можно отнести выражения, арифметические, логические, вспомогательные операторы и операции. Предлагаемый раздел поможет Вам освоить применение подобных конструкций на практике.

Тема 2.1
Арифметические операции. Оператор присваивания

Для осуществления манипуляций с данными C++ располагает широким набором *операций*. *Операции* представляют собой некоторое действие, выполняемое над одним (унарная) или несколькими (бинарная операция) операндами, результатом которого является возвращаемое значение. К базовым арифметическим операциям можно отнести операции сложения (+), вычитания (-), умножения (*), деления (/) и взятия по модулю (%), то есть вычисления остатка от деления левого операнда на правый.

Для эффективного использования возвращаемого операциями значения предназначен *оператор присваивания* (=) и его модификации: сложение с присваиванием (+=), вычитание с присваиванием (-=), умножение с присваиванием (*=), деление с присваиванием (/=), модуль с присваиванием (%=) и ряд других, которые будут рассмотрены позже.

В приведенном ниже листинге представлены примеры некоторых операций.

```
#include <iostream.h>
int main()
{
  int a = 0, b = 4, c = 90;
```

```
    char z = '\n';
    a = b;                  // a=4
    cout << a << z;
    a = b + c + c + b;  // a=4+90+90+4=188
    cout << a << z;
    a = b - 2;              // a=2
    cout << a << z;
    a = b * 3;              // a=4*3=12
    cout << a << z;
    a = c / (b + 6);        // a=90/(4+6)=9
    cout << a << z;
    cout << a%2 << z; // 9%2=1
    a += b;                 // a=a+b=9+4=13
    a *= c - 50;            // a=13*(90-50)=520
    a -= 38;                // a=520-38=482
    cout << a << z;
    a %= 8;                 // a=482%8=2
    cout << a << z;
    return 0;
}
```

В приведенном фрагменте производится инициализация объявленных переменных начальными значениями с последующей демонстрацией работы операторов присваивания, операций сложения, вычитания, умножения, деления, скобки, вычисления остатка от деления, а также различные модификации оператора присваивания.

В результате работы программа выведет столбец из цифр:

```
4 188 2 12 9 1 482 2
```

На практике также довольно широко используется *пустой оператор*:

```
;
```

Такая форма оператора в программном коде не выполняет никаких действий, но позволяет компилятору обходить требования синтаксиса на наличие оператора в некоторых конструкциях.

Тема 2.2
Понятие выражения

Выражение в C++ представляет собой последовательность операторов, операндов и знаков пунктуации, воспринимаемую компилятором как руководство к определенному действию над

данными. Всякое выражение, за которым идет точка с запятой, образует *предложение* или *инструкцию* языка:

```
выражение;
```

На практике возможны также случаи, когда сами операнды в выражениях могут быть представлены выражениями.

Приведем примеры предложений языка:

```
x = 3 * (y + 2.48);
y = My_Func(dev, 9, i);
```

Тема 2.3
Операторы инкремента и декремента

Как и в Ассемблере, в C++ имеется эффективное средство увеличения и уменьшения значения операнда на единицу – унарные операторы *инкремента* (++) и *декремента* (--).

Унарные операторы инкремента/декремента преобразуются компилятором в машинный код однозначно и на языке Ассемблера могут выглядеть так:

```
инкремент:      INC N
декремент:      DEC N
```

где N может быть либо регистром процессора, либо содержимым ячейки памяти.

По отношению к операнду данный вид операторов может быть *префиксным* или *постфиксным*. *Префиксный оператор* применяется к операнду перед использованием полученного результата. *Постфиксный оператор* применяется к операнду после использования операнда.

Понятие префикса и постфикса имеет смысл только в выражениях с присваиванием:

```
x = y++;          // постфикс
index = --current; // префикс
count++;          // унарная операция; то же, что ++count;
abc--;            // то же, что --abc;
```

Здесь переменная y сначала присваивается переменной x, а затем увеличивается на единицу. Переменная current сначала уменьшается на единицу, после чего результат присваивается переменной index.

Тема 2.4
Оператор sizeof

Программы на языке C++, по сути, являются переносимыми. Переносимость означает, что программы, написанные для работы на компьютерах одного типа, будут (хотя и не обязательно!) работать и на компьютерах другого типа. Это свойство языка C++ сделало его с самого начала популярным, когда операционная система UNIX, реализованная на C++ для машин PDP, была перенесена с небольшими изменениями на Interdata 8/32. Но при переносе программного обеспечения с ЭВМ одного типа на другой разработчика подстерегают определенные неприятности.

Переменные разных типов данных занимают в памяти компьютера неодинаковое количество байтов (см. табл. 1.2). При этом переменные одного и того же типа могут иметь разный размер еще и в зависимости от того, на каком компьютере и в какой операционной системе они используются.

Так, чистая целочисленная переменная (тип int без спецификаторов short или long) при работе в 16-разрядной операционной системе (DOS, Windows 3.1 и т.д.) занимает всего 2 байта. В то же время под эту же переменную в 32-разрядной операционной системе (Windows 95, Windows NT и т.д.) отводится уже 4 байта.

Определить размер переменной любого типа данных (как базового, так и производного) можно с помощью оператора размера sizeof. Данный оператор может быть применен к константе, типу и переменной.

Рассматриваемая ниже программа поможет вам определить размер базовых типов данных для конкретной платформы ПК.

```
int main()
{
  cout << "Size of int type:    ";
  cout << sizeof(int) << '\n';
  cout << "Size of float type:  ";
  cout << sizeof(float) << '\n';
  cout << "Size of double type: ";
  cout << sizeof(double) << '\n';
  cout << "Size of char type:   ";
  cout << sizeof(char) << '\n';
  return 0;
}
```

В результате работы программы с помощью оператора sizeof будет вычислена, а затем выведена на экран величина каждого из перечисленных базовых типов.

Тема 2.5
Поразрядные логические операции

На практике довольно часто приходится отслеживать состояния различных программных объектов с помощью флагов. Для этой цели можно использовать булевские переменные. Однако если у вас слишком много логических признаков, удобнее в качестве флагов использовать отдельные биты переменных. Для ПК на базе платформы Intel один байт представляет собой восемь бит информации, а каждый бит может принимать значение 0 или 1. Таким образом, в одном байте может быть закодировано любое целое число от 0 до 255 включительно ($2^8 = 256$). Биты в байте считаются справа налево.

Рассмотрим, например, панель из пяти выключателей SW1 ... SW5. Можно было бы объявить в качестве флагов 5 переменных типа bool (по 1 байту на каждый объект). Каждый выключатель может принимать состояние «включено» (1 - true) или «выключено» (0 - false):

```
bool SW1, SW2, SW3, SW4, SW5;
```

Тогда, проверяя значения указанных выше переменных, можно судить о состоянии того или иного выключателя.

Принимая во внимание тот факт, что биты в байте также могут принимать значения только нуля или единицы, используя компактную запись, эту же панель можно было бы уместить всего в одном байте (см. рис. 2.1).

Для того чтобы разработчик мог свободно адресоваться к отдельным битам, используются так называемые поразрядные логические операции, приведенные в табл. 2.1.

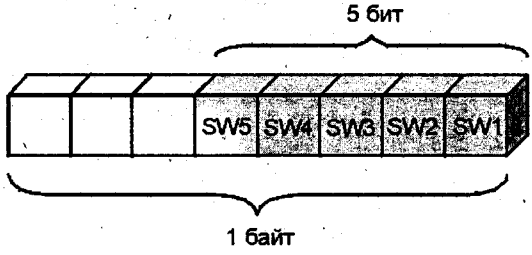


Рис. 2.1. Вариант установки значений битовых переменных в байте

Результат применения оператора «логическое И» (умножение) к двум битам равен единице только тогда, когда оба операнда установлены в единицу (см. табл. 2.2).

Таблица 2.1
Поразрядные логические операции

Операция	Описание
&	логическое И (умножение)
|	логическое ИЛИ (сложение)
^	исключающее ИЛИ
~	логическое НЕ (инверсия)

Таблица 2.2
Операция «логическое И»

A	B	C = A & B
0	0	0
0	1	0
1	0	0
1	1	1

При логическом ИЛИ (сложении) результат равен единице, если хотя бы один из участвующих в операции бит установлен в единицу (см. табл. 2.3).

Таблица 2.3
Операция «логическое ИЛИ»

A	B	C = A | B
0	0	0
0	1	1
1	0	1
1	1	1

В табл. 2.4 показано, что исключающее ИЛИ (иногда для упрощения понимания называют «исключительно ИЛИ») возвращает единицу только в том случае, если операнды не равны (первый операнд – единица, а второй – ноль или наоборот).

Таблица 2.4
Операция «исключающее ИЛИ»

A	B	C = A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Операция «логическое НЕ» (см. табл. 2.5) инвертирует биты (меняет значение на противоположное). Если исходный бит установлен, после применения к нему данной операции он сбрасывается. И наоборот, если бит сброшен в ноль, логическое НЕ установит его в единицу.

Таблица 2.5
Операция «логическое НЕ»

A	C = ~A
0	1
1	0

Вернемся к примеру с выключателями. Для всей панели отведем однобайтную переменную, например, типа char:

```
char SW = 0;
```

Присвоив первоначально переменной SW значение 0 (или в двоичном виде 00000000), мы тем самым указываем, что все выключатели находятся в выключенном состоянии.

Допустим, теперь нам надо включить третий выключатель на панели (SW3 на рис. 2.1). То есть надо сделать так, чтобы переменная SW в двоичном виде выглядела как 00000100, или в шестнадцатеричном виде 0x04. Применим для этого операцию «логическое ИЛИ»:

```
SW = SW | 0x04; // включить SW3
```

Аналогичным образом включим первый выключатель:

```
SW = SW | 0x01; // включить SW1
```

Предположим, что далее по ходу программы значение байта SW изменялось каким-то образом и перед нами стала задача выключить третий и первый выключатель, не имея информации об их текущем состоянии. Для сброса отдельных бит в байте на практике применяют операцию «логическое И».

```
SW = SW & 0xFA; // FA16 = 11111010₂
```

Ту же операцию можно было бы провести более наглядно в два приема. Так как нам требуется сбросить единичные биты в байте 00000101 (число 0x05), при условии, что они установлены, проинвертируем предварительно число 0x05 и применим логическое И:

```
SW = SW & (~0x05);
```

Теперь первый и третий биты SW, независимо от того, в каком состоянии остальные биты, будут обращены в ноль.

И в заключение изменим состояние четвертого выключателя на противоположное, независимо от его исходного значения, применив операцию «исключающее ИЛИ»:

```
SW = SW ^ 0x08; // 8₁₆ = 00001000₂
```

Следует также отметить, что на практике часто используется сокращенная форма записи для присвоения результата поразрядных логических операторов:

&= – побитовое И с присваиванием;

|= – побитовое логическое ИЛИ с присваиванием;

^= – побитовое исключающее ИЛИ с присваиванием.

Тема 2.6
Операции сдвига влево и вправо

Для осуществления сдвига последовательности бит влево и вправо применяются соответственно операции << и >>. Операнд справа от знака операции указывает, на какую величину должны быть сдвинуты биты, задавая тем самым количество бит, «выводимых» из переменной, и число нулевых бит, заполняющих переменную с другой стороны.

Например:

```
unsigned char A = 12; // A = 00001100₂
A = A << 2;           // A = 00110000₂
A = A >> 3;           // A = 00000110₂
```

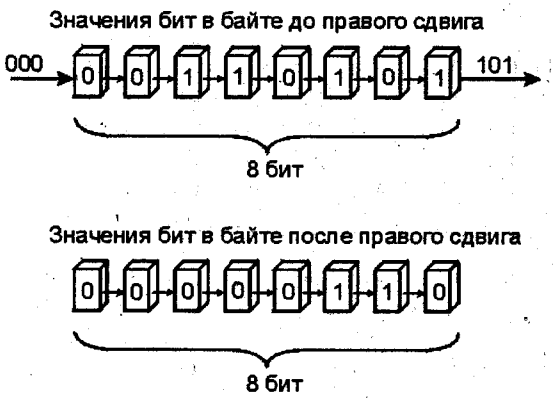


Рис. 2.2. Пример установки значений битов в байте

Следует учитывать, что при использовании правого сдвига, если самый старший бит равен единице (признак отрицательного числа у переменных со спецификатором signed), некоторые компиляторы могут не «ввести» нули слева. Во избежание подобных ситуаций рекомендуется преобразовывать операнд операции в беззнаковый тип (unsigned).

Тема 2.7
Операторы сравнения

На практике приходится регулярно сравнивать значения некоторых объектов, чтобы определить момент наступления некоторого события. Для того чтобы имелась возможность сравнивать между собой значения каких-либо переменных и констант, язык C++ предусматривает так называемые *операторы сравнения*. *Операторы сравнения* – бинарные операторы вида:

Операнд1 Оператор_сравнения Операнд2

К ним относятся следующие операторы (см. табл. 2.6):

Таблица 2.6
Операторы сравнения

Оператор	Название
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно

Здесь наблюдается аналогия с математическими операторами сравнения. В результате работы операторов сравнения возвращается логическое значение true (истина), если проверяемое условие верно, или false (ложь) в противном случае.

Тема 2.8
Операция «запятая»

Операция «запятая» связывает между собой несколько выражений таким образом, что последние рассматриваются компилятором как единое выражение.

Благодаря использованию данной операции при написании программ достигается высокая эффективность. К примеру, в операторе ветвления if, рассматриваемом ниже, можно в качестве выражения ввести:

```
if(i = CallFunc(), i > 7)
  ...
```

Тогда сначала выполнится вызов функции CallFunc() с присвоением результата переменной i, а затем произойдет сравнение значения i с числом 7.

Еще большей эффективности позволяет достичь использование операции «запятая» в операторе цикла for. Более подробно этот момент будет рассмотрен ниже.

Тема 2.9
Приоритет и порядок выполнения операций

Как и в арифметике, в C++ операции выполняются в определенном порядке. Так, например, математические выражения вычисляются слева направо, в то время как оператор присвоения выполняется справа налево.

Для того чтобы компилятор мог разобраться, какое же действие осуществлять первым при выполнении операций, последние обладают важным свойством – *приоритетом*. В первую очередь выполняются операции с наивысшим приоритетом. В таблице 2.7 приводятся используемые в C++ операции с указанием их приоритета и направления выполнения.

Таблица 2.7
Операции и приоритеты их выполнения

Оператор	Описание	Приоритет, направление	
::	Разрешение видимости	16	⇒
[ ]	Индекс массива	16	⇒
( )	Вызов функции	16	⇒
.	*Выбор члена структуры или класса*	16	⇒
->			
++	Постфиксный инкремент	15	⇐

Оператор	Описание	Приоритет, направление	
--	Постфиксный декремент	15	⇐
++	Префиксный инкремент	14	⇐
--	Префиксный декремент	14	⇐
sizeof sizeof()	Размер	14	⇐
(тип)	Преобразование типа	14	
^	Побитовое исключающее ИЛИ	14	⇐
!	Логическое НЕ	14	⇐
-	Унарный минус	14	⇐
+	Унарный плюс	14	⇐
&	Получение адреса	14	⇐
*	Разыменование	14	⇐
new new[ ]	Создание динамического объекта	14	⇐
delete delete[ ]	Удаление динамического объекта	14	⇐
casting	Приведение типа	14	
*	Умножение	13	⇒
/	Деление	13	⇒
%	Остаток от деления	13	⇒
+	Сложение	12	⇒
-	Вычитание	12	⇒
>>	Сдвиг вправо	11	⇒
<<	Сдвиг влево	11	⇒
<	Меньше	10	⇒
<=	Меньше или равно	10	⇒
>	Больше	10	⇒
>=	Больше или равно	10	⇒
==	Равно	9	⇒
!=	Не равно	9	⇒
&	Побитовое И	8	⇒

Оператор	Описание	Приоритет, направление	
^	Побитовое исключающее ИЛИ	7	⇒
|	Побитовое ИЛИ	6	⇒
&&	Логическое И	5	⇒
||	Логическое ИЛИ	4	⇒
?:	Условие	3	⇐
=	Присваивание	2	⇐
*=	Умножение с присваиванием	2	⇐
/=	Деление с присваиванием	2	⇐
%=	Модуль с присваиванием	2	⇐
+=	Сложение с присваиванием	2	⇐
-=	Вычитание с присваиванием	2	⇐
<<=	Сдвиг влево с присваиванием	2	⇐
>>=	Сдвиг вправо с присваиванием	2	⇐
&=	Побитовое И с присваиванием	2	⇐
^=	Исключающее ИЛИ с присвайванием	2	⇐
|=	Побитовое ИЛИ с присваиванием	2	⇐
throw	Генерация исключения	2	⇐
,	Запятая	1	⇒

Язык C++ предоставляет программисту возможность самостоятельно задавать порядок выполнения вычислений. С этой целью, как и в математике, операнды группируются при помощи скобок, которые могут быть вложенными друг в друга. Не существует ограничения на вложенность скобок. Приведенный ниже фрагмент иллюстрирует использование скобок для изменения порядка выполнения последовательности арифметических операторов.

```
#include <iostream.h>
int main()
{
  int x = 0, y = 0;
  int a = 3, b = 34, c = 82;
  x = a * b + c;
```

```
    y = (a * (b + c));
    cout << "x = " << x << '\n';
    cout << "y = " << y << '\n';
    return 0;
}
```

Таким образом, при вычислении значения переменной x сначала будет произведена операция умножения a на b с последующим сложением с переменной c, тогда как вычисление значения переменной y начнется со сложения переменных b и c с последующим умножением полученной суммы на величину переменной a. Результат работы программы:

```
x = 184
y = 348
```

Практикум «Выражения и операторы»

Упражнение 2.1
Арифметические операции. Составление выражений

Ниже приведен листинг программы, демонстрирующей составление выражений и использование арифметических операций. В частности, необходимо рассчитать работу, совершаемую постоянным током на участке цепи за определенное время. Для этой цели воспользуемся формулой вида:

$A = iUt$,

где t – время прохождения тока, U – напряжение на участке, i – сила тока.

Таким образом, после объявления переменных и ввода соответствующих значений производится расчет значения работы по вышеприведенной формуле.

```
#include <iostream.h>
int main()
{
    float U = 0; // напряжение на участке
    float t = 0; // время прохождения тока
    float i = 0;  // сила тока
    float A = 0; // работа постоянного тока
    cout << "Введите значение напряжения U: ";
```

```
    cin >> U;
    cout << "Введите показатель времени t: ";
    cin >> t;
    cout << "Введите значение силы тока i: ";
    cin >> i;
    A = i * U * t; // собственно вычисления
    cout << "Рассчитанная работа тока A: " << A;
    return 0;
}
```

Упражнение 2.2
Применение операторов инкремента и декремента

В настоящем упражнении Вам предлагается самостоятельно создать целочисленные переменные x и y, инициализировав их нулевыми значениями. После этого необходимо инкрементировать значение переменной x трижды и произвести вычитание с присваиванием полученного значения переменной y.

Упражнение 2.3
Определение размера переменных

В предлагаемом задании Вам предлагается составить программу, которая выведет на консоль значения размеров переменных типа bool, unsigned short и char. В качестве подсказки рекомендуется воспользоваться примером, приведенным в теоретической части темы 2.4 «Оператор sizeof».

Упражнение 2.4
Поразрядные логические операции

В данном задании Вам предлагается определить, какие биты переменной Var будут установлены, а какие - сброшены. Сверьте полученное Вами значение с результатом работы программы, листинг которой приведен ниже.

```
#include <iostream.h>
int main()
{
   char Var = 0;
   Var |= 0x0A;
   Var &= ~0x08;
```

```
    Var ^= 0x05;
    cout << (int)Var;
    return 0;
}
```

Упражнение 2.5
Правый и левый сдвиг битов

Рассматриваемые в настоящем практическом занятии операции сдвига битов в сочетании с поразрядными логическими операциями широко применяются при вычислении всевозможных контрольных сумм числовых последовательностей. Вам предлагается определить теоретически конечное значение переменной Var и подтвердить полученный результат практически.

```
#include <iostream.h>
int main()
{
    char Var = 0x9A;
    Var = Var << 2;
    Var &= ~0x08;
    Var ^= 0x05;
    Var = Var >> 1;
    cout << (int)Var;
    return 0;
}
```

Упражнение 2.6
Сравнение операндов. Операция «запятая»

Проанализируйте предлагаемый ниже листинг программы, демонстрирующей использование операции «запятая» и сравнение значений операндов. После инициализации объявленных переменных производится сравнение переменных a и b. Поскольку условие $a \le b$ удовлетворяется, переменная Result первоначально принимает значение true (1). В связи с тем, что условие $b \ge c$ не удовлетворяется, во втором случае переменная Result примет ложное значение (0).

```
#include <iostream.h>
int main()
{
    int a, b, c;
    a=0, b=3, c=7;
```

```
    bool Result;
    Result = (a <= b);
    cout << Result << '\n';
    Result = (b >= c);
    cout << Result << '\n';
    return 0;
}
```

Упражнение 2.7
Порядок выполнения операций

Проанализируйте предлагаемый ниже пример определения суммы n членов арифметической прогрессии, которая демонстрирует место приоритетов и порядка выполнения операторов. Напомним, что сумма n членов прогрессии S_n вычисляется по формуле: $S_n = \frac{2a_1 + d(n-1)}{2} n$,

где a_1 - значение первого члена прогрессии;

d - разность прогрессии.

```
#include <iostream.h>

int main()
{
    int a1, d, n;
    float Sn;
    cout << "Введите a1: ";
    cin >> a1;
    cout << "Введите d: ";
    cin >> d;
    cout << "Введите n: ";
    cin >> n;
    Sn = (2*a1 + d*(n - 1))*n / 2;
    cout << "Sn = " << Sn;
    return 0;
}
```

Раздел 3
Управление выполнением программ

Тема 3.1
Условные операторы

Все рассмотренные ранее примеры программ исполнялись в порядке следования операторов, выполняющихся исключительно по одному разу. Однако на практике возможности подобных программ весьма ограничены. Слишком малое число проблем может быть решено подобным способом – большинство задач требует от программы принятия решения в зависимости от различных ситуаций. Язык C++ обладает исчерпывающим набором конструкций, позволяющих управлять порядком выполнения отдельно взятых ветвей программы. Например, Вы можете передать управление в ту или иную часть программы в зависимости от результата проверки некоторого условия, выполнять некоторый набор операторов заданное количество раз или пока не будет выполняться какое-то условие и т.д.

Для осуществления ветвления используются так называемые *условные операторы*.

Тема 3.2
Операторы if

Оператор if производит ветвление программы в зависимости от результата проверки некоторого условия на истинность:

```
if(проверяемое_условие)
  оператор1;
оператор2;
```

Проверяемое_условие может быть любым выражением, но чаще всего оно содержит операторы сравнения.

Если проверяемое_условие принимает истинное значение (true), выполняется оператор1. В противном случае (false) выполнение программы переходит к оператору2.

В конструкциях языка C++ операторы могут быть *блочными*. Это означает, что в зависимости от принятого решения выполняется не один, а целый *блок операторов*.

Блок начинается с открывающейся фигурной скобки ({) и заканчивается закрывающейся фигурной скобкой (}). Все содержимое блока рассматривается компилятором языка как единый оператор.

Следующий пример иллюстрирует использование оператора if.

```
#include <iostream.h>
int main()
{
  int b;
  if(b > 0)
  {
  // Если условие b > 0 выполнено
    ...
     cout << "b - положительное";
    ...
  }
  if(b < 0)
  {
  // Если условие b < 0 выполнено
    ...
     cout << "b – отрицательное";
    ...
  }
  if(b==0)
  {
  // Если условие b == 0 выполнено
    ...
     cout << "b принимает нулевое значение";
    ...
  }
  return 0;
}
```

В приведенном фрагменте объявляется целочисленная переменная b, а затем следует сравнение ее значения с нулем. Если величина b больше нуля, выводится сообщение «b - положительное», а если b меньше нуля – сообщение «b – отрицательное». Если же переменная b примет нулевое значение, будет выведено сообщение «b принимает нулевое значение».

Одной из типичных ошибок при использовании оператора if является пропуск фигурных скобок для обозначения блока выполняемых операторов:

```
#include <iostream.h>
int main()
{
  int x = 0;
  int y = 8;
  int z = 0;
  if(x != 0)
  z++;
  y /= x; // Ошибка! Деление на 0
  cout << "x = ";
  return 0;
}
```

В этом примере следует проверка значения переменной x на неравенство нулю. Если значение x отлично от нуля, производится увеличение на единицу значения переменной z. Однако далее осуществляется действие вида y = y / x, независимо от того, какое значение имеет переменная x.

Если правильно определить блок исполняемых операторов, как показано ниже, вычисления выполнятся корректно:

```
if(x != 0)
{
  z++;
  y /= x;
  cout << "x = ";
}
```

В данном варианте вычисление выражения y = y / x будет производиться только для x, значение которого отлично от нуля.

Проверка на нулевое значение используется очень часто в программировании и помогает избавиться от таких нелепых ошибок, как обращение к невыделенным областям памяти и удаление объектов, которые были удалены из памяти ранее (см. раздел 5 «Указатели и ссылки»).

Тема 3.3
Операторы if-else

Оператор if с ключевым словом else имеет следующий вид:

```
if(проверяемое_условие)
   предложение1;
```

```
else
  предложение2;
предложение3;
```

Если проверяемое_условие выполняется, осуществляется переход к предложению1 с последующим переходом к предложению3. В случае когда проверяемое_условие принимает ложное значение, программа выполнит ветвь, содержащую предложение2, а затем перейдет к предложению3.

Следует отметить, что комбинация if-else позволяет значительно упростить код программы.

В качестве примера решим квадратное уравнение вида $x^2 + 2x - 3 = 0$. Для решения поставленной задачи воспользуемся методом вычисления детерминанта вида: $D = (b^2 - 4ac)$, с последующим сравнением полученного результата с нулем.

```
#include <iostream.h>
#include <math.h>
int main()
{
int a = 1;
int b = 2;
int c = -3;
float D;
float x1, x2;
D = b * b - 4 * a * c;
if(D < 0)
{
cout << "Уравнение не имеет корней";
return 0;
}
if(D == 0)
{
x1 = x2 = -b / (2 * a);
cout << "Уравнение имеет 1 корень:";
cout << " x = " << x1;
return 0;
}
else
{
x1 = (-b + sqrt(D)) / (2 * a);
x1 = (-b - sqrt(D)) / (2 * a);
cout << "Уравнение имеет 2 корня:";
cout << "\nx1 = " << x1;
cout << "\nx2 = " << x2;
}
return 0;
}
```

Типичной ошибкой программистов, переходящих на C и C++ после опыта работы на других языках программирования, является использование в условных конструкциях оператора присваивания вместо оператора сравнения (= вместо ==). Хотя язык C и позволяет производить присвоения в условных операторах, здесь следует быть особенно внимательным – в больших проектах подобные ошибки локализуются с большим трудом.

Еще одно замечание: на практике часто проверяемое условие представляет собой проверку значения некоторой целочисленной переменной. Тогда, если эта переменная принимает ненулевое значение, результатом вычисления условного выражения будет true и произойдет переход на выполнение предложения, указанного за оператором if. Поэтому в листингах программ часто встречаются записи вида:

```
if(!x)
{
// если x == 0
...
}
if(x)
{
// если x != 0
...
}
```

Например, можно было бы написать:

```
if(!D = b * b - 4 * a * c)
{
  ...
}
```

что означало бы: *присвоить переменной* D *вычисленное значение* (b^2-4ac), *и если переменная* D *приняла ненулевое значение, выполнить блочный оператор* {...}. Обычно в подобных случаях компилятор выдает предупреждение о вероятно ошибочном использовании оператора присвоения вместо сравнения.

Тема 3.4
Условный оператор ?:

Вместо операторов if-else вполне можно использовать условный оператор ?:, если входящие в него выражения являются достаточно простыми. Данная конструкция будет иметь следующий вид:

условие ? выражение1 : выражение2;

По аналогии с оператором if, данный условный оператор работает так: если условие приняло истинное значение, выполняется выражение1, а если ложное – выражение2. Обычно возвращаемое значение присваивается какой-либо переменной.

Например, решим задачу нахождения максимума из двух целочисленных переменных:

```
#include <iostream.h>
int main()
{
  int a = 10;
  int b = 20;
  int max;
  max = (a > b) ? a : b;
  cout << max;
  return 0;
}
```

Тема 3.5
Оператор switch

Еще одной альтернативой управляющей конструкции if-else может служить оператор ветвления switch. Он используется в основном в случаях, когда необходимо производить сравнение некоторой переменной с большим числом однотипных переменных или констант. Конструкция имеет следующий синтаксис:

```
switch(выражение)
{
  case константное_выражение :
  группа_операторов;
  break;
  case константное_выражение :
  группа_операторов;
  break;
  ...
  default константное_выражение :
  группа_операторов;
}
```

Конструкция switch-case представляет собой своеобразный «переключатель». Работает он следующим образом.

На первом этапе анализируется проверяемое выражение и осуществляется переход к той ветви программы, для которой его значение совпадает с указанным константным_выражением. Далее

следует выполнение оператора или группы_операторов до тех пор, пока не встретится ключевое слово break (происходит выход из тела оператора switch-case) или не будет достигнут конец блока конструкции (соответствующая закрывающаяся фигурная скобка }).

Если значения выражения и константных_выражений, указанных в case, не совпадут ни в одном из случаев, выполнится ветвь программы, определенная с помощью ключевого слова default. Вообще говоря, ключевые слова default и break не являются обязательными и зачастую программисты умышленно их опускают. Подобный прием может в некоторых случаях упростить код программы, однако его следует применять очень аккуратно. Приведенный ниже фрагмент иллюстрирует работу конструкции switch-case. Объявленная символьная переменная Answer служит для приема из входного потока ответа пользователя на вопрос, продолжить ли работу с программой.

```
#include <iostream.h>
int main()
{
  char Answer = '';
  cout << "Продолжить работу? ";
  cin >> Answer;
  switch(Answer)
  {
    case 'y':
    case 'Y':
    case 'д':
    case 'Д':
      cout << "Продолжим...\n";
      break;
    default:
      cout << "Завершение...\n";
      return 0;
  }
  // Продолжение работы
  // Здесь следует содержательная часть программы
  ...
  return 0;
}
```

Как видно из примера, пользователю достаточно выбрать любую из букв y, Y, д, Д, чтобы продолжить выполнение программы, либо нажать любую другую клавишу для завершения.

Если же по ошибке пропустить оператор break, на экран будет выведено сразу два сообщения без выполнения содержательной части программы:

```
Продолжим...
Завершение...
```

Хотя оператор switch и допускает вложения в себя аналогичных операторов, стоит избегать подобных конструкций, так как в противном случае код становится визуально плохо воспринимаемым.

Наиболее эффективно применение связки операторов switch-case с использованием в программе перечислений. В приведенном ниже фрагменте объявляется перечисление цветов гаммы Rainbow и соответствующая этому типу переменная Color. Далее осуществляется сравнение значения переменной Color с заданными значениями цветов.

```
#include <iostream.h>
int main()
{
  enum Rainbow {Red, Orange, Yellow, Green,
                Blue, Cyan, Magenta};
  Rainbow Color;
...
  switch(Color)
  {
  case Red:
  case Orange:
  case Yellow:
    cout << "Выбрана теплая гамма\n";
    break;
  case Green:
  case Blue:
  case Cyan:
  case Magenta:
    cout << "Выбрана холодная гамма\n";
    break;
  default:
    cout << "Радуга не имеет такого цвета!\n";
  }
  return 0;
}
```

Если на момент проверки переменная Color примет значение, соответствующее красному (Red), оранжевому (Orange) или желтому (Yellow) цвету, будет выведено сообщение "Выбрана теплая гамма". Если проверяемая переменная будет соответствовать зеленому (Green), синему (Blue), голубому (Cyan) или фиолетовому (Magenta) цвету, будет отображено сообщение "Выбрана холодная гамма". Если же проверяемое значение переменной Color не сов-

падет ни с одним из вышеперечисленных, будет выведена строка "Радуга не имеет такого цвета!".

Тема 3.6
Операторы цикла

Следующим мощным механизмом управления ходом последовательности выполнения программы является использование циклов.

Цикл задает многократное выполнение одного и того же куска кода программы (итерации). Он имеет точку вхождения, проверочное условие и (необязательно) точку выхода. Цикл, не имеющий точки выхода, называется бесконечным. Для бесконечного цикла проверочное условие всегда принимает истинное значение.

Проверка условия может осуществляться перед выполнением (циклы for, while) или после окончания (do-while) тела цикла.

Циклы могут быть вложенными друг в друга произвольным образом.

Тема 3.7
Циклы for

Синтаксис цикла for имеет вид:

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

Этот оператор работает следующим образом.

Сначала выполняется выражение1, если оно присутствует в конструкции. Затем вычисляется величина выражения2 (если оно указано) и, если полученный результат принял истинное значение, выполняется тело цикла (оператор_или_блок_операторов). В противном случае выполнение цикла прекращается и осуществляется переход к оператору, следующему непосредственно за телом цикла.

После выполнения тела цикла вычисляется выражение3, если оно имеется в конструкции, и осуществляется переход к пункту вычисления величины выражения2.

Выражение1 чаще всего служит в качестве инициализации какой-нибудь переменной, выполняющей роль счетчика итераций.

Выражение2 используется как проверочное условие и на практике чаще всего содержит выражения с операторами срав-

нения. По умолчанию величина выражения2 принимает истинное значение.

Выражение3 обычно служит для приращения значения счетчика циклов либо содержит выражение, влияющее каким бы то ни было образом на проверочное условие.

Все три выражения не обязательно должны присутствовать в конструкции, однако синтаксис не допускает пропуска символа точка с запятой (;). Поэтому простейший пример бесконечного цикла for (выполняется постоянно до принудительного завершения программы извне) выглядит следующим образом:

```
for( ; ; )
  cout << "Бесконечный цикл...";
```

Если в цикле должны синхронно изменяться значения нескольких переменных, которые зависят от переменной цикла, вычисление их значений можно поместить в оператор for, воспользовавшись оператором «запятая».

Типичная ошибка программирования циклов for – изменение значения счетчика как в конструкции (выражение3), так и в теле цикла. Это может приводить к таким негативным последствиям, как «выпадение» или повтор итераций.

Рассмотрим несколько примеров.

Для начала просуммируем набор из десяти целых чисел, начиная с 10.

```
#include <iostream.h>
int main()
{
  int Sum = 0;
  for(int i=10; i<20; i++)
    Sum += i;
  cout << "Сумма составит: " << Sum;
  return 0;
}
```

Как видно из примера, при инициализации был объявлен счетчик i, с начальным значением 10, увеличивающийся с каждой итерацией на единицу. Тело цикла в данном случае состоит из одного-единственного оператора, добавляющего к результирующей величине Sum значение счетчика i в данной итерации. Используя свойство оператора «запятая» объединять операторы в выражения, вышеприведенный пример можно модифицировать

таким образом, что тело цикла будет выполняться внутри выражения3:

```
#include <iostream.h>
int main()
{
  int Sum = 0;
  for(int i=10; i<20; i++, Sum += i) ;
  cout << "Сумма составит: " << Sum;
  return 0;
}
```

Обратите внимание на то, что в качестве тела цикла используется пустой оператор (;). Его применение в данном случае обязательно, так как компилятор требует, чтобы тело цикла было непустым.

Следующий фрагмент иллюстрирует применение оператора «запятая» в цикле for:

```
for(int i=10, j=2; i<20; i++, j=i+17)
{
  // Здесь можно использовать вычисленное значение
  // переменной j, т.к. она не оказывает влияния на
  // проверочное условие цикла
  ...
}
```

Таким образом, первоначально инициализируются переменные i и j, а затем при каждой из 10 итераций будет вычисляться новое значение переменной j.

Теперь введем в программу задержку по времени на 500 циклов с использованием оператора for. В каждом новом цикле будет уменьшаться на единицу значение целочисленной переменной delay и выполняться пустой оператор до тех пор, пока переменная delay не примет нулевое значение.

```
#include <iostream.h>
int main()
{
int delay;
// Начало вычислений
  ...
// Задержка 500 циклов
  for(delay=500; delay>0; delay--)
    ; // Пустой оператор, ничего делать не надо!
      // Продолжение вычислений
  ...
```

```
    return 0;
}
```

Здесь следует отметить, что применение подобных задержек в программе не совсем корректно, т.к. ПК обладают разной производительностью и циклы будут вычисляться с различной скоростью.

Пример вычисления факториала демонстрирует использование в качестве тела цикла блочного оператора, хотя приведенный текст не является самым рациональным способом вычисления факториала (в дальнейшем будет рассмотрена более эффективная реализация).

```
#include <iostream.h>
int main()
{
  int a = 0;
  unsigned long fact = 1;
  cout << "Введите целое число: ";
  cin >> a;
  if((a >= 0) && (a < 33))
  {
    for(int i=1; i<=a; i++)
    {
      if(a != 0)
        fact *= i;
      else
        fact = 1;
    }
    cout << fact << '\n';
    fact = 1;
  }
  return 0;
}
```

В примере пользователю предлагается ввести целое число от 0 до 33. В организованном далее цикле от единицы до введенного числа производится собственно вычисление факториала.

Тема 3.8
Циклы while

Оператор цикла while выполняет оператор или блок до тех пор, пока проверочное условие (выражение) остается истинным. Он имеет следующий синтаксис:

```
while(выражение)
оператор_или_блок
```

Если выражение представляет собой константу с истинным значением, тело цикла будет выполняться всегда и, следовательно, мы имеем дело с бесконечным оператором. Цикл также окажется бесконечным, когда условие, определенное в выражении изначально, истинно и нигде далее в теле цикла не изменяется. Если же проверочное условие возвращает ложное значение, осуществится выход из цикла и тело оператора while будет пропущено.

Довольно часто в качестве выражения используется оператор присваивания. Так как при этом возвращается некоторое число, в операторе while фактически производится сравнение полученного значения с нулем (следует напомнить, что ноль – эквивалент ложного значения) с дальнейшим принятием решения о выходе из цикла либо о его продолжении.

Как и для оператора for, если в цикле должны синхронно изменяться несколько переменных, которые зависят от переменной цикла, вычисление их значений можно поместить в проверочное выражение оператора while, воспользовавшись оператором «запятая».

Приведенный ниже пример показывает использование оператора цикла while для представления десятичного числа в двоичном виде.

```
#include <iostream.h>
int main()
{
  int counter = 4;
  short dec;
  while(1)   // Бесконечный цикл
  {
    cout << "Введите десятичное число от 0 до 15: ";
    cin >> dec;
    cout << "Двойчное представление числа " << dec << ": ";
    while(counter)
    {
      if(dec & 8)
        cout << 1;
      else
        cout << 0;
      dec = dec << 1; // Сдвиг влево на 1 бит
      counter--;      // Изменяем счетчик итераций
    }
    cout << '\n';
```

```
    counter = 4;
  }
  return 0;
}
```

В приведенном примере вводится десятичное число dec и далее в цикле while осуществляется проверка четвертого бита (число от 0 до 15 может быть закодировано четырьмя битами). Если бит установлен (равен единице), на экран выводится единица, а если он находится в сброшенном состоянии – выводится ноль.

Тема 3.9
Циклы do-while

В отличие от оператора while, цикл do-while сначала выполняет тело (оператор или блок), а затем уже осуществляет проверку выражения на истинность. Такая конструкция гарантирует, что тело цикла будет обязательно выполнено хотя бы один раз.

Синтаксис оператора имеет вид:

```
do
  оператор_или_блок_операторов;
while(выражение)
```

Одно из часто используемых применений данного оператора – запрос к пользователю на продолжение выполнения программы:

```
#include <iostream.h>
int main()
{
  char answer;
  do
  {
    // Тело программы
    ...
    cout << "Продолжать выполнение?";
    cin >> answer;
  }
  while(answer != 'N')
  return 0;
}
```

Таким образом, тело программы будет повторяться до тех пор, пока пользователь на вопрос «Продолжать выполнение?» не ответит символом N. При этом в качестве переменной, используемой для хранения этого значения, выступает answer.

Тема 3.10
Оператор break

Вполне вероятно, у вас уже назревает вопрос, что же делать, если нужно предусмотреть выход из оператора цикла в нескольких местах, не дожидаясь выполнения оставшейся части кода. Здесь на помощь программисту приходит оператор break. Данный оператор, как и в случае switch-case, передает управление вне тела конструкции и может встречаться в теле цикла сколько угодно раз.

В приведенном ниже примере пользователю предлагается ввести число, которое в дальнейшем тестируется на попадание в один из трех диапазонов. На самом деле, если введенное число больше трех, происходит выход из тела цикла while и запрос на ввод числа повторяется.

```
#include <iostream.h>
int main()
{
  int answer = 0;
  do
  {
    while(answer)
    {
      if(answer <= 3)
        cout<<"Меньше 3-х\n";
      break; // Выход из цикла while
// В этот фрагмент кода никогда не попадем
      if(answer > '3' && answer < '7')
        cout<<"От 3-х до 7-ми\n";
      break;
      if(answer >= '7')
        cout<<"Больше 7-ми\n";
    }
// В этот фрагмент кода попадаем по break
    cout << "Repeat?\n";
    cin >> answer;
  }
  while(answer != 0);
  return 0;
}
```

На практике оператор break часто используют для выхода из бесконечного цикла.

```
for( ; ; )
```

```
{
// Условие 1:
  if(...)
  {
    ...
    break;
  }
// Условие 2
  if(...)
  {
    ...
    break;
  }
}
```

В данном примере, если выполнится любой из условных операторов if, будет достигнут и соответствующий оператор break, который выведет управление из тела конструкции бесконечного цикла for.

Тема 3.11
Оператор continue

Так же как и ключевое слово break, оператор continue прерывает выполнение тела цикла, но в отличие от первого, он предписывает программе перейти на следующую итерацию цикла.

В качестве примера использования оператора continue предлагается программа нахождения простых чисел (делящихся на 1 и на самих себя).

```
#include <iostream.h>
int main()
{
  bool dev = false;
  for(int i=2; i<50; i++)
  {
    for(int j=2; j<i; j++)
    {
      if(i%j)
        continue;
      else
      {
        dev = true;
        break;
      }
    }
    if(!dev)
```

```
        cout << i << " ";
      dev = false;
    }
    return 0;
}
```

Программа организована в виде двух вложенных циклов таким образом, что осуществляется перебор и проверка остатка от деления пары чисел, первое из которых изменяется от 2 до 50, а второе – от 2 до значения первого числа. Если остаток от деления ненулевой, осуществляется продолжение внутреннего цикла по оператору continue. В случае если остаток от деления составил 0, выполняется выход из внутреннего цикла с установкой признака деления в логической переменной dev. По окончании внутреннего цикла производится анализ логической переменной dev и вывод простого числа.

Тема 3.12
Оператор goto и метки

Метка представляет собой идентификатор с расположенным за ним символом двоеточия (:). Метками помечают какой-либо оператор, на который в дальнейшем должен быть осуществлен безусловный переход.

Безусловная передача управления на метку производится при помощи оператора goto. Оператор goto может осуществлять переход (адресоваться) к меткам, обязательно расположенным в одном с ним теле функции.

Синтаксис оператора goto:

```
goto метка;
```

Данный оператор – очень мощное и небезопасное средство управления поведением программы. Использовать его нужно крайне осторожно, так как, например, «прыжок» внутрь цикла (обход кодов инициализации) может привести к трудно локализуемым ошибкам.

Рассмотрим пример применения оператора goto.

```
#include <iostream.h>
int main()
{
  // Инициализация переменных
  ...
```

```
    if(...)
    {
    // Если условие выполнено, производим какие-то действия
    // и переходим к оператору по метке label
      ...
      goto label;
    }
    ...
  label:
    ...
    return 0;
  }
```

Как только выполнение программы достигнет оператора goto, управление будет передано оператору, следующему за меткой label.

Вообще говоря, использование структурного и объектно-ориентированного подходов к программированию позволяет полностью отказаться от применения операторов безусловного перехода. Однако на практике часто бывают случаи, когда goto значительно упрощает код программы. В особенной степени это утверждение касается вложенных конструкций switch-case и if-else.

В качестве примера вернемся к модели светофора, дополнив ее соответствующим образом:

```
#include <iostream.h>
int main()
{
  enum lighter {RED, REDYELLOW, YELLOW, GREEN};
  lighter current = YELLOW;
  lighter previous = GREEN;
  lighter next = RED;
start:
  switch(current)
  {
    case RED:
      cout << "СТОП!!!\n";
      if(previous == YELLOW)
      {
        next = REDYELLOW;
        break;
      }
      else
        goto lab1;
    case REDYELLOW:
      if(previous == YELLOW)
        goto lab1;
      if(previous == GREEN)
```

```
            goto lab1;
          cout << "Приготовиться!\n";
          next = GREEN;
          break;
        case YELLOW:
          if(previous == YELLOW)
            goto lab1;
          if(previous == RED)
            goto lab1;
          cout << "Остановись!\n";
          next = RED;
          previous = GREEN;
          break;
        case GREEN:
          if(previous == RED)
            goto lab1;
          if(previous == YELLOW)
            goto lab1;
          previous = REDYELLOW;
          next = YELLOW;
          cout << "Вперед!\n";
          break;
      }
      previous = current;
      current = next;
      goto start;
    lab1:
      cout << "Светофор неисправен...\n";
      current = previous = next = YELLOW;
      goto start;
      return 0;
    }
```

В приведенной программе объявляются три переменные для хранения информации о текущем (current), предыдущем (previous) и следующем (next) состоянии светофора. В операторе switch производится анализ текущего состояния и на основании полученного результата осуществляется вывод соответствующего сообщения. В случае сбоя работы светофора (рассчитанные значения переменных next и previous не совпадают с реальными данными), осуществляется переход к метке lab1 с последующим переводом светофора в режим «Неуправляемый перекресток». После проверки выполнение программы продолжается с метки start.

Практикум
«Управление выполнением программ»

Упражнение 3.1
Использование условного оператора if

Проанализируйте пример, демонстрирующий применение условного оператора if в теле программы C++.

Пусть необходимо ввести некоторое вещественное число и определить, на каком интервале лежит тестируемое значение, является ли оно отрицательным, положительным или равным нулю.

```
#include <iostream.h>
int main()
{
   float var;
   cin >> var; // Вводимое значение
   if(var < 0)
   {
      cout << "Отрицательное";
   }
   if(var == 0)
   {
      cout << "Нуль";
   }
   if(var > 0)
   {
      cout << "Положительное";
   }
   return 0;
}
```

Упражнение 3.2
Применение оператора ветвления if-else

В настоящей практической работе Вам предлагается проработать вопрос составления оператора ветвления if-else. Пример включает использование генератора случайных чисел для задания случайного целочисленного значения переменной DialogExecution в диапазоне (0;1).

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

```
int main()
{
   const short IDOK = 0;
   const short NBC_START = 220;
   const short NBC_END = 225;
   int DialogExecution, MessageState;
   randomize(); // инициализация генератора случайных чисел
   DialogExecution = rand() % 1; // 0 или 1
      if(DialogExecution == IDOK)
      {
               MessageState = NBC_START;
               cout << "Служба запущена";
      }
      else
      {
               MessageState = NBC_END;
               cout << "Служба остановлена";
      }
   return 0;
}
```

Упражнение 3.3

Практическое использование оператора ?:

В этой части практического задания Вам предлагается на основе предыдущего практического упражнения самостоятельно составить программу, в которой оператор if-else будет заменен условным оператором вида ?:.

Упражнение 3.4

Ветвление с помощью оператора switch

В данном задании умышленно допущены неточности. Вам предлагается обнаружить и устранить ошибки в приведенном ниже листинге с тем, чтобы в программе корректно обрабатывались все перечисленные операции с числом 2 над операндом x.

```
#include <iostream.h>
const int ADD = 0;
const int SUBTRACT = 1;
const int MULTIPLY = 3;
const int DEVIDE = 4;
const int SQUARE = 5;
int main()
{
   int x = 8;
```

```
    int Operation = ADD;
    switch( Operation )
    {
       case ADD:
          x += 2;
       case SUBTRACT:
          x -= 2;
       case MULTIPLY:
          x *= 2;
       case DEVIDE:
          x /= 2;
       case SQUARE:
          x = x * x;
    }
    return 0;
}
```

Упражнение 3.5

Оператор цикла с известным числом итераций for

В предлагаемом практическом занятии рассматривается пример подсчета единичных битов в некоторой переменной типа long. В приведенном листинге переменная Counter исполняет роль счетчика битов, установленных в «1», а переменная Var – собственно анализируемое число.

В цикле for, начиная с нулевого бита, осуществляется проверка, установлен ли крайний правый бит в «1» (самый младший бит переменной Var), и если это так, значение счетчика инкрементируется. Следующим шагом значение анализируемой переменной сдвигается вправо на 1 бит для последующей проверки.

Поскольку заранее известно, сколько бит занимает переменная Var, а именно: sizeof(Var) байт * 8 бит; число итераций также заранее известно и, следовательно, в нашем случае целесообразно применить цикл for.

```
#include <iostream.h>
int main()
{
   int Counter = 0;
   long Var = 0;
   cout << "Введите значение типа long: "; cin >> Var;
   cout << "Число" << Var << " содержит ";
   for(int i=0; i<sizeof(Var)*8; i++)
   {
      if(Var & 0x01)
```

```
        Counter++;
      Var = Var >> 1;
   }
   cout << Counter << " единичных битов";
   return 0;
}
```

Упражнение 3.6
Циклы с предусловием while

В данной лабораторной работе Вам предлагается самостоятельно модифицировать пример предыдущего практического задания, заменив цикл for оператором цикла while.

Упражнение 3.7
Использование циклов с постусловием do-while

В предлагаемой лабораторной работе Вам предлагается обнаружить и устранить ошибку компиляции программы, представленной следующим листингом:

```
#include <iostream.h>
int main()
{
   int x = 300;
   do
   {
      cout << x << '\n';
      x /= 2;
   }while(x>0)
   return 0;
```

Упражнение 3.8
Оператор окончания цикла break

Оператор break относится к операторам перехода, осуществляя безусловную передачу управления в некоторое место программы. Так же как и в случае с оператором ветвления switch, в операторах цикла break прерывает выполняющийся оператор. На практике очень часто оператор break используется для выхода из бесконечного цикла. При этом обычно ему предшествует проверочное условие. Проанализируйте предлагаемый ниже пример, иллюстрирующий такую ситуацию.

```
#include <iostream.h>
int main()
{
   float x = 3;
   float y = 2;
   while(true)
   {
      x += y;
      x /= 2;
      cout << x << '\n';
      if(x > 10000)
         break;
      x *= 10;
   }
   cout << x << '\n';
   return 0;
}
```

Упражнение 3.9
Переход к новой итерации с помощью continue

Так же как оператор break, оператор continue относится к операторам перехода. Его отличие состоит лишь в том, что break осуществляет передачу управления на первый оператор, следующий за оператором цикла (или switch-оператора), а continue прекращает текущую итерацию, осуществляя переход к проверочному выражению цикла.

Проанализируйте предлагаемый ниже пример, где в операторе for осуществляется 1000 итераций, в каждой из которых происходит проверка на четность номера итерации. Если итерация нечетная, осуществляется приращение счетчика counter, в противном случае выполняется переход к следующей итерации по continue.

```
#include <iostream.h>
int main()
{
   int counter = 0;
   for(int i=0; i<1000; i++)
   {
      if(i%2)
         continue;
      counter++;
   }
   cout << counter << '\n';
```

```
    return 0;
}
```

Упражнение 3.10
Применение меток и оператора безусловного перехода

Общеизвестно, что метод трапеций является одним из наиболее простых способов вычисления значения определенного интеграла вида $I = \int_a^b f(x)dx$. При использовании данного метода подынтегральная функция $f(x)$ заменяется ломаной, а сам интеграл – суммой площадей трапеций. Тогда отрезки, соответствующие значениям функции $f(x)$ в точках x_i и x_{i+1}, являются основаниями трапеции, а отрезок ломаной и его проекция на ось x – ее боковыми сторонами.

Если ввести обозначение вида $y_i = f(x_i)$, то обобщенную формулу вычисления интеграла методом трапеции можно записать в виде:

$$I \approx \frac{b-a}{n}\left(\frac{y_0}{2} + y_1 + y_2 + \ldots + y_{n-1} + \frac{y_n}{2}\right),$$

где n – число разбиений отрезка [a;b].

Проанализируйте предлагаемый ниже пример вычисления интеграла функции $y = \sqrt{2x-1}$ на отрезке [1;50]:

```
#include <iostream.h>
#include <math.h>
int main()
{
   float a=1, b=50, e=1;
   float n=20;
   float s1=0;
   float h; float x, s;
   label1: h=(b-a)/n; x=a, s=0;
   for(int i=1; i<=n-1; i++)
   {
      x=x+h;
      s=s+sqrt(2*x-1);
   }
   s=s+(sqrt(2*a-1) + sqrt(2*b-1))/2;
   s=s*h;
   if((fabs(s - s1)) < e) goto label2;
```

```
    n=n*2;
    s1=s;
    goto label1;
    label2: cout << s;
    return 0;
}
```

Реализация подынтегральной функции $y=\sqrt{2x-1}$ осуществляется с помощью использования функции квадратного корня sqrt() из модуля math.h.

В качестве практики попробуйте модифицировать вышеприведенный пример, отказавшись от оператора goto и используя только операторы циклов for, while, do-while, а также безусловных переходов break и continue.

Раздел 4
Функции

Разработка программного обеспечения на практике является довольно непростым процессом. Программисту требуется учесть все тонкости и нюансы как всего программного комплекса в целом, так и отдельных его частей. Системный подход к программированию основывается на том, что поставленная перед разработчиком задача предварительно разбивается на пару-тройку менее крупных вопросов, которые, в свою очередь, делятся еще на несколько менее сложных задач, и так до тех пор, пока самые мелкие задачи не будут решены с помощью стандартных процедур. Таким образом осуществляется так называемая функциональная декомпозиция.

Ключевым элементом данной модели для решения конкретной задачи в C++ выступает функция. Функцию можно представить как подпрограмму или некую процедуру, несущую законченную смысловую нагрузку.

Тема 4.1
Параметры и аргументы функций

Каждая функция, которую предполагается использовать в программе, должна быть в ней объявлена. Обычно объявления функций размещают в заголовочных файлах, которые затем подключаются к исходному тексту программы с помощью директивы #include. Объявление функции описывает ее прототип (иногда говорят, сигнатура). Прототип функции объявляется следующим образом:

```
возвр_тип FuncName(список объявляемых параметров);
```

Здесь возвр_тип – возвращаемый функцией тип данных. Если возвращаемый тип данных не указан, то по умолчанию компилятор считает, что возвращаемый функцией тип есть int. Список

объявляемых параметров задает тип и имя каждого параметра функции, разделенные запятыми. Допускается опускать имя параметра. Список объявляемых параметров функции может быть пустым. Приведем примеры прототипов функций:

```
int     swap(int, int);
double  max (double  par1, double par2);
void    func();
```

Прототип функции может быть пропущен, если определение функции следует до первого ее вызова из любой другой функции. Последний вариант считается плохим стилем программирования, так как бывают случаи перекрестных вызовов, то есть когда две или несколько функций вызывают друг друга, в результате чего невозможно вызвать одну функцию без предварительного определения другой.

Определение функции состоит из ее заголовка и собственно тела, заключенного в фигурные скобки и несущего смысловую нагрузку. Если функция возвращает значение, отличное от типа void, в теле функции обязательно должен присутствовать оператор return с параметром того же типа, что и возвращаемое значение. В случае если возвращаемое значение не будет использоваться в дальнейшем в программе (void), оператор return следует без параметра или вообще может быть опущен, тогда возврат из функции осуществляется по достижении последней закрывающейся фигурной скобки тела функции.

Для того чтобы функция выполнила определенные действия, она должна быть вызвана в программе. При обращении к функции она выполняет поставленную задачу, а по окончании работы возвращает в качестве результата некоторое значение.

Вызов функции представляет собой указание идентификатора функции (ее имени), за которым в круглых скобках следует список аргументов, разделенных запятыми:

```
имя_функции ( аргумент_1, аргумент_2 ..., аргумент_N);
```

Каждый аргумент функции представляет собой переменную, выражение или константу, передаваемые в тело функции для дальнейшего использования в вычислительном процессе. Список аргументов функции может быть пустым.

Функция может вызывать другие функции (одну или несколько), а те, в свою очередь, производить вызов третьих и т.д. Кроме того, функция может вызывать сама себя. Это явление в программировании называется рекурсией. Подробнее рекурсия будет рассмотрена ниже.

Как уже было отмечено ранее, любая программа на C++ обязательно включает в себя главную функцию main(). Именно с этой функции начинается выполнение программы.

На рис. 4.1 показан схематический порядок вызова функций.

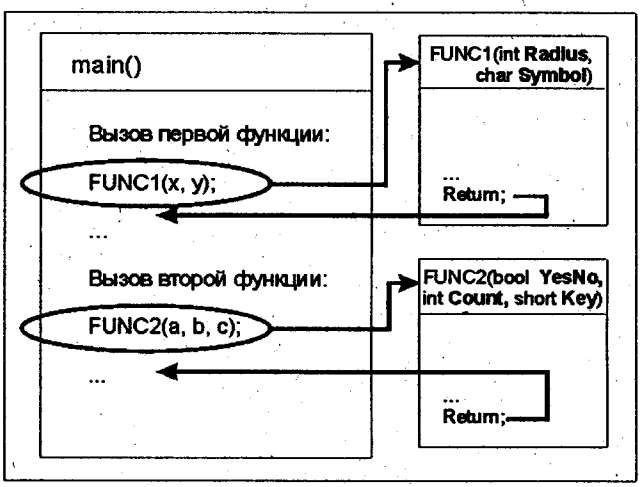


Рис. 4.1. Вызов функций

Программа начинает выполняться с функции main() до вызова функции FUNC1(x, y). С этого момента управление программой передается в функцию FUNC1(x, y), причем в качестве значения переменной Radius данная функция использует величину переменной x, а в качестве переменной Symbol передается значение y (рис. 4.2 иллюстрирует передачу параметров в функции). Далее до оператора return выполняется тело функции FUNC1(x, y), после чего управление возвращается в тело функции main(), а именно следующему за вызовом FUNC1(x, y) оператору. После этого продолжается выполнение функции main() до вызова функции FUNC2(a, b, c). При вызове этой функции переменная a передает значение логической переменной YesNo, переменная b – целочисленной переменной Count, а переменная c – короткому целому Key.

Функция main() часто не имеет аргументов, однако если требуется при вызове программы передать ей какие-нибудь параметры, синтаксис функции main() меняется:

```
int main(argc, argv)
```

Здесь первый аргумент, argc, указывает количество передаваемых параметров, а второй, argv, является указателем на мас-

сив символьных строк, содержащих эти аргументы. Массивы и указатели будут рассмотрены позже.

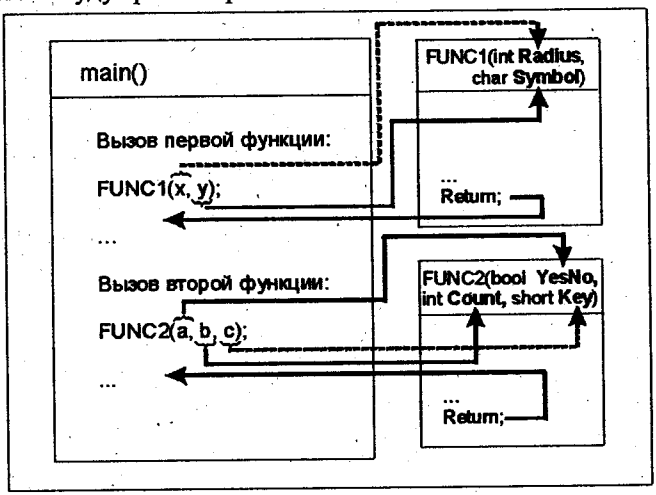


Рис. 4.2. Передача параметров в функции

Рассмотрим несколько примеров объявления, вызова и определения функций.

```
// Объявление функций:
int MyFunction(int Number, float Point);
char InputSymbol();
void SetBit(short Num);
void EmptySample(int, char);
// Вызов функций:
Result = MyFunction(varbl, 3.14);
symb = InputSymbol();
SetBit(3);
EmptySample(2, smbl);
// Определение функций:
int MyFunction(int Number, float Point)
{
  int my_x;
  ...
  return my_x; // my_x – типа int
};
char InputSymbol()
{
  char symbol;
  cin >> symbol;
  return symbol;
};
```

```
void SetBit(short number)
{
  GlobalBit = GlobalBit | number;
};
void EmptySample(int x, char ch)
{
};
```

Для того чтобы было более понятно место функции в программе, детально рассмотрим пример вычисления квадрата числа с использованием функции.

В заголовочном файле header.h разместим прототип функции MySquare():

```
// header.h
long MySquare(int);
```

Тогда главный модуль программы будет подключать заголовочный файл, содержать описание и вызов функции MySquare() из функции main().

```
#include "header.h"
int main()
{
  int Variable = 5;
  cout << MySquare(Variable);
  return 0;}
long MySquare(int x)
{
  return x * x;
}
```

Тот же пример может выглядеть несколько иначе, если вместо подключения заголовочного файла поместить прототип функции MySquare() прямо в файл исходного текста программы:

```
// Прототип функции:
long MySquare(int);
int main()
{
  int Variable = 5;
  cout << MySquare(Variable);
  return 0;
}
long MySquare(int x)
{
  return x * x;
}
```

Результат работы программы не изменится – на печать будет выведено число 25.

Тема 4.2
Аргументы по умолчанию

C++ допускает при вызове функций опускать некоторые ее параметры. Достигается это указанием в прототипе функции значений аргументов по умолчанию. Например, функция, прототип которой приведен ниже, может при вызове иметь различный вид в зависимости от ситуации.

```
// Прототип функции:
void ShowInt (int i, bool Flag = true, char symbol = '\n');

// Вызовы функции ShowInt:
ShowInt(1, false, 'a');
ShowInt(2, false);
ShowInt(3);
```

В первом случае все три аргумента заданы явно, поэтому работа функции осуществляется в обычном режиме. Во втором вызове в функцию передается два параметра из трех, причем вместо последнего аргумента подставляется значение по умолчанию, а именно символ '\n'. Третий вариант обращения к функции сообщает только один целочисленный параметр, а в качестве остальных аргументов используются значения по умолчанию: логическая переменная со значением true и символьная переменная со значением '\n'.

Для используемых параметров по умолчанию существует обязательное правило: все параметры справа от аргумента по умолчанию должны также иметь значение по умолчанию. Так, в приведенном выше прототипе нельзя было бы, указав значение параметра по умолчанию для целочисленной переменной i, пропустить определение любого из остальных аргументов по умолчанию.

Рассмотрим пример, в котором осуществляется вывод знакового числа двойной точности с указанием количества значащих символов. Другими словами, определим функцию, принимающую в качестве одного из параметров число выводимых знаков. Для решения поставленной задачи можно воспользоваться функцией возведения числа в степень pow() и функцией взятия модуля от длинного числа с плавающей точкой fabsl(), прототипы которых содержатся в заголовочном файле math.h:

```
#include <iostream.h>
#include <math.h>
void Out(double Numb, double Sig=1, bool Flg=true);
int main()
{
  double Mpi = -3.141592654;
  Out(Mpi, 4, false);
  Out(Mpi, 2);
  Out(Mpi);
  return 0;
}
void Out(double numb, double sig, bool flg)
{
  if(!flg)
    numb = fabsl(numb);
  numb = (int) (numb * pow(10, sig));
  numb = numb / pow(10,sig);
  cout << numb << '\n';
}
```

В теле программы производится вызов одной и той же функции Out() с различным числом параметров для вывода значения переменной двойной точности Mpi. В результате работы программы на печать будут выведены следующие значения:

```
 3,1415
-3,14
-3,1
```

Величина, указываемая в аргументах по умолчанию, может быть не только константным выражением – она может быть глобальной переменной или значением, возвращаемым некоторой функцией.

Тема 4.3
Области видимости. Локальные и глобальные переменные

Переменные могут быть объявлены как внутри тела какой-либо функции, так и за пределами любой из них.

Переменные, объявленные внутри тела функции, называются локальными. Такие переменные размещаются в стеке программы и действуют только внутри той функции, в которой объявлены. Как только управление возвращается вызывающей функции, память, отводимая под локальные переменные, освобождается.

Каждая переменная характеризуется областью действия, областью видимости и временем жизни.

Под областью действия переменной понимают область программы, в которой переменная доступна для использования.

С этим понятием тесно связано понятие области видимости переменной. Если переменная выходит из области действия, она становится невидимой. С другой стороны, переменная может находиться в области действия, но быть невидимой. Переменная находится в *области видимости*, если к ней *можно* получить доступ (с помощью операции разрешения видимости, в том случае, если она непосредственно не видима).

Временем жизни переменной называется интервал выполнения программы, в течение которого она существует.

Локальные переменные имеют своей областью видимости функцию или блок, в которых они объявлены. В то же время область действия локальной переменной может исключать внутренний блок, если в нем объявлена переменная с тем же именем. Время жизни локальной переменной определяется временем выполнения блока или функции, в которой она объявлена.

Это означает, например, что в разных функциях могут использоваться переменные с одинаковыми именами совершенно независимо друг от друга.

В рассматриваемом ниже примере переменные с именем x определены сразу в двух функциях – в main() и в Sum(), что, однако, не мешает компилятору различать их между собой:

```
#include <iostream.h>
// Прототип функции:
int Sum(int a, int b);
int main()
{
  // Локальные переменные:
  int x = 2;
  int y = 4;
  cout << Sum(x, y);
  return 0;
}
int Sum(int a, int b)
{
  // Локальная переменная x
  // видна только в теле функции Sum()
  int x = a + b;
return x;
}
```

В программе осуществляется вычисление суммы двух целочисленных переменных посредством вызова функции Sum().

Глобальные переменные, как указывалось ранее, объявляются вне тела какой-либо из функций и действуют на протяжении выполнения всей программы. Такие переменные доступны в любой из функций программы, которая описана после объявления глобальной переменной. Отсюда следует вывод, что имена локальных и глобальных переменных не должны совпадать. Если глобальная переменная не проинициализирована явным образом, она инициализируется значением 0.

Область действия глобальной переменной совпадает с областью видимости и простирается от точки ее описания до конца файла, в котором она объявлена. Время жизни глобальной переменной – постоянное, то есть совпадает с временем выполнения программы.

Вообще говоря, на практике программисты стараются избегать использования глобальных переменных и применяют их только в случае крайней необходимости, так как содержимое таких переменных может быть изменено внутри тела любой функции, что чревато серьезными ошибками при работе программы.

Рассмотрим пример, поясняющий вышесказанное:

```
#include <iostream.h>
// Объявляем глобальную переменную Test:
int Test = 200;
void PrintTest(void);
int main()
{
// Объявляем локальную переменную Test:
  int Test = 10;
  // Вызов функции печати глобальной переменной:
  PrintTest();
  cout << "Локальная: " << Test << '\n';
  return 0;
}
void PrintTest(void)
{
  cout << "Глобальная: " << Test << '\n';
}
```

Первоначально объявляется глобальная переменная Test, которой присваивается значение 200. Далее объявляется локальная переменная с тем же именем Test, но со значением 10. Вызов

функции PrintTest() из main() фактически осуществляет временный выход из тела главной функции. При этом все локальные переменные становятся недоступны и PrintTest() выводит на печать глобальную переменную Test. После этого управление программой возвращается в функцию main(), где конструкцией cout<< выводится на печать локальная переменная Test. Результат работы программы выглядит следующим образом:

```
Глобальная: 200
Локальная: 10
```

В C++ допускается объявлять локальную переменную не только в начале функции, а вообще в любом месте программы. Если объявление происходит внутри какого-либо блока, переменная с таким же именем, объявленная вне тела блока, «прячется». Видоизменим предыдущий пример с тем, чтобы продемонстрировать процесс сокрытия локальной переменной:

```
#include <iostream.h>
// Объявляем глобальную переменную Test:
int Test = 200;
void PrintTest(void);
int main()
{
// Объявляем локальную переменную Test:
  int Test = 10;
  // Вызов функции печати глобальной переменной:
  PrintTest();
  cout << "Локальная: " << Test << '\n';
  // Добавляем новый блок с еще одной
  // локальной переменной Test:
  {
    int Test = 5;
    cout << " Локальная: " <<Test << '\n';
  }
  // Возвращаемся к локальной Test вне блока:
  cout << " Локальная: " <<Test << '\n';
  return 0;
}
void PrintTest(void)
{
  cout << "Глобальная: " << Test << '\n';
}
```

Результат модифицированной программы будет выглядеть следующим образом:

```
Глобальная: 200
Локальная: 10
Локальная: 5
Локальная: 10
```

Тема 4.4
Операция ::

Как было показано выше, объявление локальной переменной скрывает глобальную переменную с таким же именем. Таким образом, все обращения к имени глобальной переменной в пределах области действия локального объявления вызывают обращение к локальной переменной. Однако C++ позволяет обращаться к глобальной переменной из любого места программы с помощью использования операции разрешения области видимости. Для этого перед именем переменной ставится префикс в виде двойного двоеточия (::):

```
#include <iostream.h>
// Объявление глобальной переменной
int Turn = 15;
int main()
{
  // Объявление локальной переменной
  int Turn = 70;
  // Вывод локального значения:
  cout << Turn << '\n';
  // Вывод глобального значения:
  cout << ::Turn << '\n';
  return 0;
}
```

В результате в две строки будет выведено два значения: 15 и 70.

Из рассмотренного примера видно, что были объявлены глобальная и локальная переменные с именем Turn, которые позже были выведены на печать.

Тема 4.5
Классы памяти

В разделе 1 уже упоминалось о таком важном свойстве переменной, как время жизни. Существует четыре модификатора переменных, определяющих область видимости и время действия переменных. Все они приводятся в таблице 4.1.

Таблица 4.1

Модификаторы переменных

Модификатор	Применение	Область действия	Время жизни
auto	локальное	Блок	временное
register	локальное	Блок	временное
extern	глобальное	Блок	временное
static	локальное	Блок	постоянное
	глобальное	Файл	
volatile	глобальное	Файл	постоянное

Ниже будет рассмотрен более подробно каждый из приведенных классов памяти.

Тема 4.6
Автоматические переменные

Модификатор auto используется при описании локальных переменных. Поскольку для локальных переменных данный модификатор используется по умолчанию, на практике его чаще всего опускают.

```
#include <iostream.h>
int main()
{
  auto int MyVar = 2;  // то же, что int MyVar = 2;
  cout << MyVar;
  return 0;
}
```

Модификатор auto применяется только к локальным переменным, которые видны исключительно в блоке, в котором они объявлены. При выходе из блока такие переменные уничтожаются автоматически.

Тема 4.7
Регистровые переменные

Модификатор register предписывает компилятору попытаться разместить указанную переменную в регистрах процессора. Если *такая попытка оканчивается неудачно, переменная ведет себя как*

локальная переменная типа auto. Размещение переменных в регистрах оптимизирует программный код по скорости, так как процессор оперирует с переменными, находящимися в регистрах, гораздо быстрее, чем с памятью. Но в связи с тем, что число регистров процессора ограничено, количество таких переменных может быть очень небольшим.

```
#include <iostream.h>
int main()
{
  register int REG;
  ...
  return 0;
}
```

Модификатор register применяют только к локальным переменным. Попытка употребления данного модификатора (так же как и модификатора auto) к глобальным переменным вызовет сообщение компилятора об ошибке. Переменная существует только в пределах блока, содержащего ее объявление.

Тема 4.8
Внешние переменные и функции

Если программа состоит из нескольких модулей, некоторые переменные могут использоваться для передачи значений из одного файла в другой. При этом некоторая переменная объявляется глобальной в одном модуле, а в других файлах, в которых она должна быть видима, производится ее объявление с использованием модификатора extern. Если объявление внешней переменной производится в блоке, она является локальной.

В отличие от рассмотренных ранее, этот модификатор сообщает, что первоначальное объявление переменной производится в каком-то другом файле. Рассмотрим пример использования внешней переменной:

```
// файл myheader.h
void ChangeFlag(void);
// файл myfunction.cpp
extern bool Flag;
void ChangeFlag(void)
{
  Flag = !Flag;
}
```

```
#include <iostream.h>
#include "myheader.h"
extern bool Flag;

int main()
{
  ChangeFlag();
  if(Flag)
    cout << "Сейчас TRUE\n";
  else
    cout << " Сейчас FALSE\n";
  return 0;
}
```

Сначала в файле myheader.h объявляется функция ChangeFlag(). Далее в файле myfunction.cpp следует объявление глобальной логической переменной Flag и определяется реализация тела функции ChangeFlag(), и, наконец, в главном модуле подключается заголовочный файл myheader.h и переменная Flag описывается как внешняя (extern). Поскольку описание функции ChangeFlag() включается в главный модуль директивой #include "myheader.h", данная функция доступна в теле функции main().

Тема 4.9
Статические переменные

Статические переменные во многом похожи на глобальные переменные. Для описания статических переменных используется модификатор static. Если такая переменная объявлена глобально, то она инициализируется при запуске программы, а ее область видимости совпадает с областью действия и простирается от точки объявления до конца файла. Если же статическая переменная объявлена внутри функции или блока, то она инициализируется при первом входе в соответствующую функцию или блок. Значение переменной сохраняется от одного вызова функции до другого. Таким образом, статические переменные можно использовать для хранения значений переменных на протяжении времени работы программы. Статические переменные не могут быть объявлены в других файлах как внешние.

Если статическая переменная не проинициализирована явным образом, то, как и глобальная переменная, она инициализируется значением 0.

В качестве примера рассмотрим реализацию счетчика вызовов некоторой функции:

```
#include <iostream.h>
int Count(void);
int main()
{
  int result;
  for(int i=0; i<30; i++)
    result = Count();
  cout << result;
  return 0;
}
int Count(void)
{
  static short counter = 0;
  ...
  counter++;
  return counter;
}
```

Здесь главная функция в цикле (30 раз подряд) вызывает функцию Count(), которая содержит статическую переменную counter. Как видно из примера, начальная инициализация этой переменной нулем выполнится только один раз, при первом вхождении в тело функции. Поскольку значение переменной сохраняется между вызовами функции, на печать будет выведено число 30.

Тема 4.10
Переменные класса volatile

В тех случаях, когда необходимо предусмотреть возможность модификации переменной периферийным устройством или другой программой, используют модификатор volatile. В связи с этим компилятор не пытается оптимизировать программу путем размещения переменной в регистрах.

Пример объявления таких переменных приведен ниже:

```
volatile short sTest;
volatile const int vciTest;
```

Как видно из примера, переменная vciTest с модификатором volatile в то же время может быть объявлена как константа. В этом случае ее значение не сможет меняться в разрабаты-

ваемой программе, но может модифицироваться в зависимости от внешних факторов.

Тема 4.11
Новый стиль заголовков

Исторически так сложилось, что в языке C++ при подключении заголовочных файлов использовался тот же синтаксис, что и в языке C для совместимости с разработанным на тот момент программным обеспечением. Однако при стандартизации языка этот стиль был изменен, и теперь вместо заголовочных файлов (как это было в C) указываются некоторые стандартные идентификаторы, по которым компилятор сам находит необходимые файлы. Предопределенные идентификаторы представляют собой имя заголовка в угловых скобках без указания расширения (.h). Ниже приводится пример включения заголовков в стиле C++:

```
#include <iostream>
#include <stdlib>
#include <new>
```

Помимо этого, для включения в программу библиотек функций языка C в соответствии с новым стандартом заголовок преобразуется следующим образом: отбрасывается расширение .h и к имени заголовка добавляется префикс c. Таким образом, например, заголовок <string.h> заменяется заголовком <cstring>. Если же используемый компилятор не поддерживает объявления заголовков в новом стиле, можно по-прежнему использовать заголовки в стиле языка C, хотя это и не рекомендуется стандартом C++.

Тема 4.12
Пространства имен

Определения функций и переменных в заголовочных файлах неразрывно связаны с понятием *пространства имен*. Это понятие появилось сравнительно недавно. До введения понятия пространства все объявления идентификаторов и констант, сделанные в заголовочном файле, помещались компилятором в глобальное пространство имен. Такое положение вещей приводило к возникновению массы конфликтов, связанных с использованием различными объектами одинаковых имен. Чаще всего недоразуме-

ния возникали, когда в одной программе использовались библиотеки, разработанные различными производителями. Введение понятия пространства имен позволило значительно снизить количество подобных конфликтов имен. Когда в программу включается заголовок нового стиля, его содержимое помещается не в глобальное пространство имен, а в пространство имен std. Если в программе требуется определить некоторые идентификаторы, которые, как Вы подозреваете, могут переопределить уже имеющиеся, просто заведите свое собственное, новое пространство имен. Это достигается путем использования ключевого слова namespace:

```
namespace имя_пространства_имен
{
    // объявления
    ...
}
```

Таким образом, объявления внутри нового пространства имен будут находиться только внутри видимости определенного имени_пространства_имен, предотвращая тем самым возникновение конфликтов. В качестве примера создадим следующее пространство имен:

```
namespace NewNameSpace
{
  int x, y, z;
  void SomeFunction(char smb);
}
```

Для того чтобы указать компилятору, что следует использовать имена из конкретного именного пространства (в данном случае из NewNameSpace), можно воспользоваться операцией разрешения видимости:

```
NewNameSpace::x = 5;
```

Однако, если в программе обращения к собственному пространству имен производятся довольно часто, такой синтаксис вызывает определенные неудобства. В качестве альтернативы можно воспользоваться инструкцией using, синтаксис которой имеет две формы:

```
using namespace имя_пространства_имен;
```

или

```
using имя_пространства_имен::идентификатор;
```

При использовании первой формы компилятору сообщается, что в дальнейшем необходимо использовать идентификаторы из указанного именного пространства вплоть до того момента, пока не встретится следующая инструкция using. Например, указав в теле программы

```
using namespace NewNameSpace;
```

можно напрямую работать с соответствующими идентификаторами:

```
x=0; y=z=4;
SomeFunction('A');
```

На практике часто после включения в программу заголовков явно указывается использование идентификаторов стандартного пространства имен:

```
using namespace std;
```

Вторая форма записи предписывает компилятору использовать указанное пространство имен лишь для конкретного идентификатора. Таким образом, определив

```
using namespace std;
using NewNameSpace::z;
```

можно использовать идентификаторы стандартной библиотеки C++ и целочисленную переменную z из пространства имен NewNameSpace без использования операции разрешения видимости:

```
z=12;
```

Следует понимать, что указание нового пространства имен инструкцией using namespace отменяет видимость стандартного пространства std, поэтому для получения доступа к соответствующим идентификаторам из std потребуется каждый раз использовать операцию разрешения видимости std::.

Пространства имен не могут быть объявлены внутри тела какой-либо функции, однако могут объявляться внутри других пространств. При этом для доступа к идентификатору внутреннего пространства необходимо указать имена всех вышестоящих именованных пространств. Например, объявлено следующее пространство имен:

```
namespace Highest
{
  namespace Middle
```

```
    {
      namespace Lowest
      {
        int nAttr;
      }
    }
}
```

Использование объявленной переменной nAttr будет выглядеть:

```
Highest::Middle::Lowest::nAttr = 0;
```

Допускается объявление нескольких именованных пространств с одним и тем же именем, что позволяет разделить его на несколько файлов. Несмотря на это, содержимое всех частей будет храниться в одном и том же пространстве имен.

Чтобы объявления переменных и функций в некотором пространстве имен были более упорядоченными, рекомендуется в пределах описания пространства имен объявлять только прототипы функций, помещая определение тела функции отдельно. При этом следует явно указывать, к какому пространству имен относится функция:

```
namespace Nspace
{
  char c;
  int i;
  void Func1(char Flag);
}
void Nspace::Func1(char Flag)
{
  // тело функции
  ...
}
```

Кроме вышесказанного, допускается объявление неименованных пространств имен. В этом случае просто опускается имя пространства после ключевого слова namespace. Например:

```
namespace
{
  char cByte;
  long lValue;
}
```

Обращение к объявленным элементам производится по их имени, без какого-либо префикса. Неименованные пространства

имен могут быть использованы только в том файле, в котором они объявлены.

Стандарт языка C++ предусматривает определение *псевдонимов* пространства имен, которые ссылаются на конкретное пространство имен. Чаще всего псевдонимы используются для упрощения работы с длинными именами пространств. Следующий пример иллюстрирует создание более короткого псевдонима и его использование для доступа к переменной.

```
namespace A_Very_Long_Name_Of_NameSpace
{
    float y;
}
A_Very_Long_Name_Of_NameSpace::y = 0.0;
namespace
   Neo = A_Very_Long_Name_Of_NameSpace;
Neo::y = 13.4;
```

Тема 4.13
Встраиваемые (inline-) функции

В результате работы компилятора каждая функция представляется в виде машинного кода. Если в программе вызов функции встречается несколько раз, в местах таких обращений генерируются коды вызова уже реализованного экземпляра функции. Однако выполнение вызовов требует некоторой затраты времени. Таким образом, если тело функции небольшого размера и обращение к ней в программе происходит довольно часто, на практике можно указать компилятору вместо вызовов функции в соответствующих местах генерировать все ее тело. Осуществляется это с помощью ключевого слова inline. Тем самым увеличивается производительность реализованного кода, хотя, конечно, размер программы может значительно возрасти. Компиляторы различных фирм накладывают свои ограничения на использование встраиваемых функций, поэтому перед использованием inline-функций необходимо обратиться к руководству компилятора.

Ключевое слово inline должно предшествовать первому вызову встраиваемой функции (например, содержаться в ее прототипе).

```
#include <iostream.h>
// Прототип встраиваемой функции:
```

```
inline int Sum(int, int);
int main()
{
  int A=2, B=6, C=3;
  char eol = '\n';
// Вызовы встраиваемой функции,
// генерируют все тело функции
  cout << Sum(A, B) << eol;
  cout << Sum(B, C) << eol;
  cout << Sum(A, C) << eol;
  return 0;
}
int Sum(int x, int y)
{
  return x + y;
}
```

В приведенном примере в каждом месте вызова функции Sum() будет сгенерирован код тела всей функции.

Тема 4.14
Рекурсивные функции

Как уже упоминалось ранее, функция может вызывать сама себя. При этом говорят, что возник *рекурсивный* вызов. Рекурсия бывает:

- простой – если функция в теле содержит вызов самой себя;
- косвенной – если функция вызывает другую функцию, а та в свою очередь вызывает первую.

При выполнении рекурсии программа сохраняет в стеке значения всех локальных переменных функции и ее аргументов, с тем чтобы в дальнейшем по возвращении из рекурсивного вызова восстановить их сохраненные значения. Рис. 4.3 иллюстрирует поведение рекурсивной функции.

В связи с вышеизложенным, применять рекурсию следует с осторожностью, так как ее использование для функций, содержащих большое количество переменных или слишком большое число рекурсивных вызовов, может вызвать переполнение стека. Следует также помнить, что при использовании рекурсивного вызова разработчик обязан предусмотреть механизм возврата в вызывающую процедуру, чтобы не произошло образования бесконечного цикла.

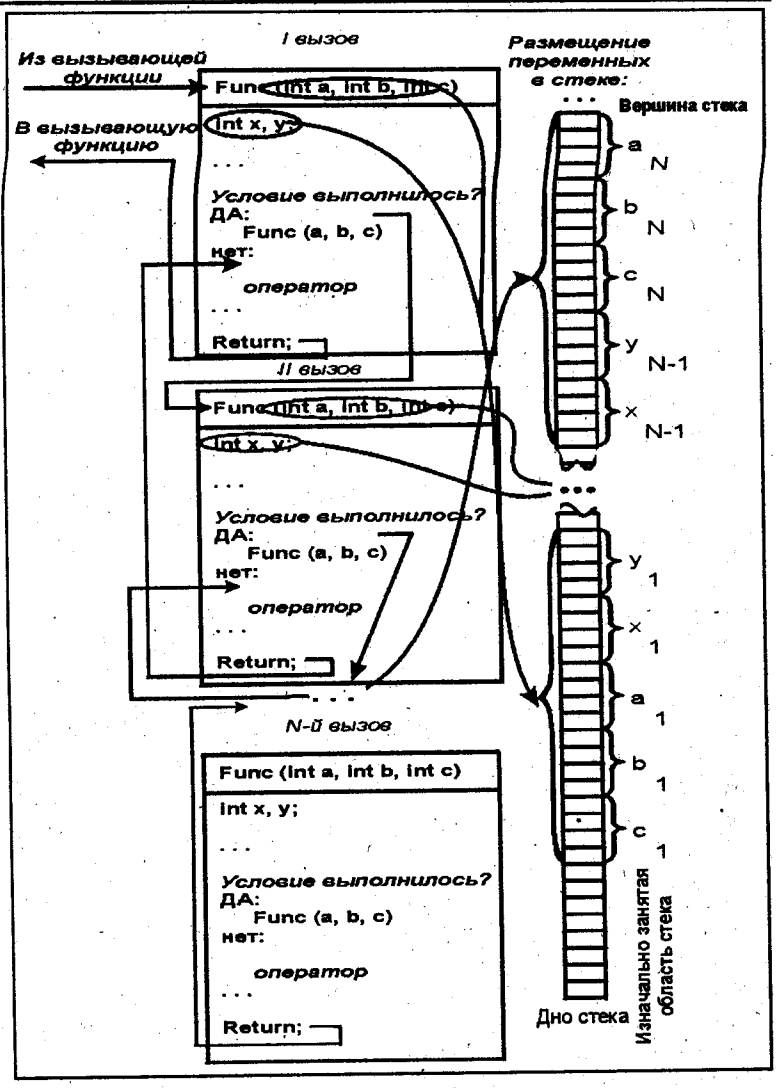


Рис. 4.3. Рекурсивный вызов

Некоторые задачи на практике могут быть проще и нагляднее решены именно с использованием рекурсивных функций. Например, решение тривиальной задачи нахождения факториала без обращения к рекурсии могло бы выглядеть следующим образом:

```
#include <iostream.h>
int main()
{
  int count = 1;
  long int result =1;
  while(count && count <31)
  {
    cout << "Введите целое число: ";
    cin >> count;
    for(int i=count; i>1; i--)
    {
      result *= i;
    }
    cout << result << '\n';
    result = 1;
  }
  return 0;
}
```

Ту же задачу можно решить более элегантно, применив рекурсию:

```
#include <iostream.h>
long int fact(long);
int main()
{
  int count = 1;
  while(count && count <31)
  {
    cout << "Введите целое число: ";
    cin >> count;
    cout << fact(count) << '\n';
  }
  return 0;
}

long int fact(long x)
{
  if(x==0 || x==1)
    return 1;
  return x * fact(x-1);
}
```

Как видно, тело функции main() во втором примере максимально упростилось и занимается фактически вводом значения и выводом результата, в то время как все вычисления возложены на единственную содержательную строку в рекурсивной функции fact().

Тема 4.15
Математические функции

Прототипы стандартных математических функций определены в заголовочном файле math.h. Рассмотрим некоторые из них, наиболее часто употребляемые в работе.

Ранее уже упоминалась функция pow(), позволяющая возводить число в степень. Синтаксис данной функции выглядит следующим образом:

```
double pow(double x, double y);
```

Таким образом, компилятору сообщается, что необходимо число двойной точности x возвести в степень числа двойной точности y.

К данной категории также относятся логарифмические функции и функция извлечения корня числа:

```
double log(double); // натуральные
float logf( float );
long double logl(long double);
double log10(double); // десятичные
float log10f(float);
long double log10l(long double);
double sqrt(double); // корень числа
float sqrtf(float);
long double sqrtl(long double);
```

Другая большая группа – функции получения абсолютной величины числа:

```
int abs(int);       // целые
double fabs(double); // двойной точности
long labs(long);     // длинные
float fabsf(float);  // с плавающей точкой
long double fabsl(long double);  // длинные двойной точности
```

Эти функции воспринимают в качестве параметра аргумент некоторого типа (свой для каждой из функций) и возвращают его беззнаковую форму.

Для вычисления остатка от деления числа x на y используется функция fmod(), которая имеет следующий синтаксис:

```
double fmod(double x, double y);
```

Стандартная библиотека располагает широким набором тригонометрических функций и их модификаций для различных типов аргументов:

```
// Арккосинус:
double acos(double);
float  acosf(float);
// Арксинус:
double asin(double);
float  asinf(float);
// Арктангенс:
double atan(double);
float  atanf(float);
// Арктангенс отношения y/x:
double atan2(double x, double y);
float  atan2f(float, float);
// Косинус:
double cos(double);
float  cosf(float);
// Гиперболический косинус:
double cosh(double);
float  coshf(float);
// Синус:
double sin(double);
float  sinf(float);
// Гиперболический синус:
double sinh(double);
float  sinhf(float);
// Тангенс:
double tan(double);
float  tanf(float);
// Гиперболический тангенс:
double tanh(double);
float  tanhf(float);
```

Следует отметить, что углы тригонометрических функций указываются в радианах. Ниже приводится пример, осуществляющий перевод градусов в радианы и вывод значения синуса для введенного в градусах числа.

```
#include <iostream.h>
#include <math.h>
int main()
{
  double Angle;
  double PI = 3.14159;
  cout << "Введите угол в градусах: ";
  cin >> Angle;
  cout << "Значение синуса: ";
```

```
    cout << sin(Angle * PI / 180) << '\n';
    return 0;
}
```

К сожалению, в C++ нет готовой реализации функции возведения аргумента в квадрат. Эту функцию можно реализовать, например, следующим образом:

```
inline double sqr(double x)
{return pow(x, 2);}
```

Заметим, что для работы данной функции необходимо задействовать заголовочный файл math.h, содержащий прототип функции pow().

Тема 4.16
Функции округления

Зачастую требуется воспользоваться округленным значением той или иной переменной. C++ предлагает набор функций для решения этой задачи. В зависимости от конкретной ситуации может понадобиться функция, округляющая значение аргумента в большую или меньшую сторону. Рассмотрим наиболее часто используемые варианты вызовов.

Для округления числа в меньшую сторону используется функция floor() и ее разновидности для различных типов аргументов и возвращаемых параметров. Данная функция имеет следующий синтаксис:

```
double floor(double x);
long double floorl(long double x);
```

Округление в большую сторону производится с помощью функции ceil():

```
double ceil(double x);
long double ceil(long double x);
```

Однако в реальности проблема выбора в какую же сторону производить округление, возлагается на разрабатываемую программу. Ниже предлагается два варианта решения этой задачи.

Вариант I:

```
inline double Round(double x)
{
  return floor(x + .5);
```

```
}
```

Вариант II:

```
double round(double num)
{
  double frac;
  double val;
  frac = modf(num, &val);
  if(frac < 0.5) num = val;
  else num = val + 1.0;
  return num;
}
```

Для работы обоих вариантов функции необходимо задействовать заголовочный файл math.h, содержащий прототип функций floor() и modf().

Практикум «Функции»

Упражнение 4.1 Понятие и синтаксис функции

Проанализируйте предлагаемый ниже пример приближенного вычисления определенного интеграла методом Симпсона. Суть его сводится к следующему.

Подынтегральная функция заменяется отрезками парабол, а затем вычисляется сумма площадей полученных криволинейных трапеций. В общем случае

$$I = \int_a^b f(x)dx \approx \frac{h}{3}[f(a) + 4f(a+h) + 2f(a+2h) +$$

$$+ \ldots + 4f(b-h) + f(b)]$$

В приведенном ниже листинге демонстрируется использование функций для вычислений по вышеприведенной формуле. В качестве подынтегральной функции выбрана функция вида $y = \frac{1}{x^2}$, однако использование понятия «функция» в теле программы позволяет изменить подынтегральную функцию на любую другую (с тем же числом и типами аргументов) в одной-единственной строке листинга.

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

```
double f(double x);
int main()
{
   double a, b;
   int n = 20; // число разбиений отрезка
   double e;
   cout << "Введите пределы интегрирования:\n";
   cout << "a = "; cin >> a;
   cout << "b = "; cin >> b;
   cout << "Введите точность вычислений: e = ";
   cin >> e;
   double s1 = 0, h, s;
   do
   {
      h = (b-a) / n;
      s = f(a) + f(b);
      for(float i=1; i<=n/2-1; i++)
      {
         s = s + 4 * f(a + (2 * i - 1) * h) + 2 * f(a + 2 * i * h);
      }
      s = s + 4 * f(b - h);
      s = s * h / 3;
      if(fabs(s - s1) < e)
      {
         cout << "Значение интеграла: " << s;
         return 0;
      }
      s1 = s;
      n = n * 2;
   }
   while(true);
}
double f(double x)
{
   return 1/(x*x);
}
```

Упражнение 4.2
Использование аргументов по умолчанию

В данном упражнении Вам предлагается самостоятельно составить текст программы, использующей вызов функции с тремя целочисленными параметрами, один из которых задан по умолчанию.

Упражнение 4.3
inline-функции

Напишите программу, в которой будет производиться вызов inline-функции, осуществляющей сдвиг влево на один бит переданного ей аргумента и возвращающей полученное новое значение.

Упражнение 4.4
Применение рекурсии

Проанализируйте предлагаемый ниже пример применения рекурсивных функций, реализующий функцию возведения в степень. Данная функция должна получать в качестве аргументов вещественное число x, возводимое в некоторую степень, и, собственно, целочисленный показатель степени exp.

```
#include <iostream.h>
double  power(double x, int exp)
{
   if (exp <= 0)
      return(1);
   else
      return(x * power(x, exp-1));
}
int main()
{
   double x;
   int exp;
   cin >> x; cin >> exp;
   cout << power(x, exp);
   return 0;
}
```

Упражнение 4.5
Практическое использование математических функций

Данное практическое задание состоит двух частей, в которых Вам предлагается самостоятельно написать программу, реализующую те или иные действия с использованием математических функций.

1. Напишите программу, в которой используется вызов inline-функции расчета логарифма некоторого числа по некоторому основанию, если известно, что

$$\log_b x = \frac{\lg x}{\lg b} = \frac{\ln x}{\ln b},$$

где:
x – некоторое число;
b – основание;
lg – десятичный логарифм числа;
ln – натуральный логарифм числа.

2. Составьте программу вычисления тангенса некоторого угла с использованием функций $y = \sin(x)$ и $y = \cos(x)$, если известно, что $\tan x = \frac{\sin x}{\cos x}$. Листинг программы должен включать функцию перевода введенного пользователем значения градусов в радианы.

Упражнение 4.6
Локальные и глобальные переменные

Проанализируйте предлагаемый ниже пример использования локальных и глобальных переменных. Глобальные константные объекты программы Global, eol, tab и целочисленная переменная Var доступны в любой части рассматриваемого программного модуля, поскольку они объявлены вне пределов какой-либо функции. В отличие от них, вещественная переменная Var сохраняет свое значение лишь в функции main(), где она объявлена, а целочисленная переменная i, являясь локальной, еще и перекрывается одноименной переменной, объявленной в цикле for.

```
#include <iostream.h>
const int Global = 365;
int Var = 15;
const char eol = '\n';
const char tab = '\t';

int main()
{
   float Var = 32;
   cout << Var << eol;
   int i = 34;
   cout << i << eol;
```

```
    for(int i=0; i<19; i++)
    {
       cout << i << tab << Global << eol;
    }
    cout << Var << eol;
    return 0;
}
```

Упражнение 4.7
Разрешение области видимости

Проанализируйте пример, иллюстрирующий место операции разрешения области видимости в программе C++.

В модуле объявляется два объекта с именем Letter: глобальная константа и локальная переменная. Поскольку глобальная константа перекрывается объявленной позже локальной переменной, в цикле for осуществляется присвоение значений переменной и ошибок компиляции не возникает. Чтобы получить доступ к глобальному объекту Letter, применена операция разрешения области видимости(::).

```
#include <iostream.h>
const char Letter = '1';
int main()
{
   char Letter;
   for(int i='A'; i<'Z'; i++)
   {
      Letter = (char)i;
      cout << Letter;
   }
   cout << '\n' << ::Letter;
   return 0;
}
```

Упражнение 4.8
Модификаторы auto и register

Найдите и устраните ошибки в приведенном ниже листинге:

```
#include <iostream.h>
auto int Misstake;
int main()
```

```
{
   register int i;
   auto bool Enabled;
   char Automatic;
   return 0;
}
```

Упражнение 4.9
Работа с внешними объектами

В данном упражнении Вам необходимо будет создать файл проекта и добавить в него два прилагаемых модуля, листинг которых представлен ниже.

```
// Extern1.cpp
#include <iostream.h>
extern int Sum(int, int);

int main()
{
   int x = 12, y = 5, z = 500;
   int res = Sum(Sum(x, y), Sum(y, z));
   cout << res;
   return 0;
}
// Extern2.cpp
int Sum(int a, int b)
{
   return a + b;
}
```

Как видно из программы, в модуле Extern1.cpp объявлена внешняя функция extern int Sum(int, int), которая на самом деле реализована во втором модуле(Extern2.cpp).

Упражнение 4.10
Использование переменных static и volatile

Проанализируйте предлагаемый ниже пример, в котором описывается функция, содержащая статический элемент counter и возвращающая его значение. Фактически, функция просто подсчитывает количество раз, сколько она была вызвана.

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

```
int main()
{
   int res = 0;
   for(int i=0; i<20; i++)
      Count();
   res = Count();
   cout << res;
   return 0;
}
int Count(void)
{
   static counter = 0;
   return ++counter;
}
```

Рассмотрим пример использования ключевого слова volatile. Предположим, имеется некоторая функция timer(), некоторым образом связанная с аппаратным прерыванием таймера. Для реализации функции задержки на заданное число «тиков» (переменная interval) необходимо указать компилятору, что глобальная переменная tics, участвующая в проверочном условии цикла while, может изменяться внешними источниками, такими как фоновые подпрограммы, посредством прерывания или порта ввода/вывода.

```
volatile int ticks;
void timer( )
{
  ticks++;
}
void wait(int interval)
{
  ticks = 0;
  while(ticks < interval);  // Ожидание
}
```

Раздел 5
Указатели и ссылки

Тема 5.1
Понятие указателя

Любой объект программы, будь то переменная базового или же производного типа, занимает в памяти определенную область. Местоположение объекта в памяти определяется его *адресом*. Как уже говорилось, при объявлении переменной для нее резервируется место в памяти, размер которого зависит от типа данной переменной, а для доступа к содержимому объекта служит его имя (идентификатор). При обращении к переменной определяется ее фактическое местоположение в памяти, и все действия ведутся уже с этой конкретной областью. Для выяснения адреса конкретной переменной служит унарная операция взятия адреса. При этом перед именем переменной ставится знак амперсанда (&). Приведенный ниже пример программы выведет на печать сначала значение переменных Var1 и Var2, а затем их адреса:

```
#include <iostream.h>
#define  endl  '\n'
int main()
{
 unsigned int Var1 = 40000;
 unsigned int Var2 = 300;
 cout << "Значение1: " << Var1 << endl;
 cout << "Адрес    : " << &Var1 << endl;
 cout << "Значение2: " << Var2 << endl;
 cout << "Адрес    : " << &Var2 << endl;
 return 0;
}
```

В результате будет выведено:

```
Значение1: 40000
Адрес    : 0x0068fe00
```

```
Значение2: 300
Адрес    : 0x0068fdfc
```

Как видно из примера, адреса локальных переменных, размещаемых в стеке, следуют в обратном порядке (стек растет в направлении младших адресов). Результат может отличаться даже при повторном запуске программы, так как невозможно предугадать, по какому адресу начнут размещаться переменные. Важно другое: разница в адресах первой и второй переменной всегда будет одинакова и при четырехбайтном представлении типа int составит 4 байта (см. рис. 5.1).

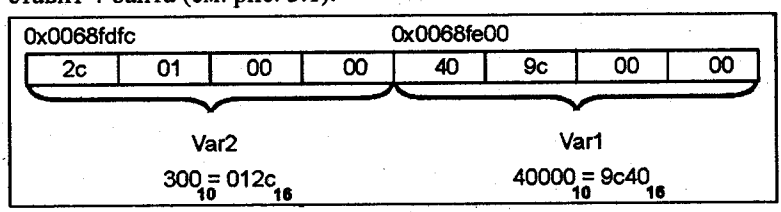


Рис. 5.1. Переменные в памяти

Мощным средством разработчика программного обеспечения на C++ является возможность осуществления непосредственного доступа к памяти. Для этой цели предусматривается специальный тип переменных – *указатели*.

Указатель (pointer) представляет собой переменную, значение которой является адресом ячейки памяти. Указатель может ссылаться на переменную (базового или производного типа) или функцию. Наибольшая эффективность применения указателей в разработке приложений достигается при использовании их с *массивами* и *символьными строками*.

Объявление указателя имеет следующий синтаксис:

```
тип_объекта* идентификатор;
```

Здесь тип_объекта определяет тип данных, на которые ссылается указатель с именем идентификатор. Символ «звездочка» (*) сообщает компилятору, что объявленная переменная является указателем, и независимо от того, сколько памяти требуется отвести под сам объект, для указателя резервируется два или четыре байта в зависимости от используемой модели памяти.

Поскольку указатель представляет собой ссылку на некоторую область памяти, ему может быть присвоен только адрес некоторой переменной (или функции), а не само ее значение (см. рис. 5.2). В случае некорректного присвоения компилятор выдаст соответствующее сообщение об ошибке.

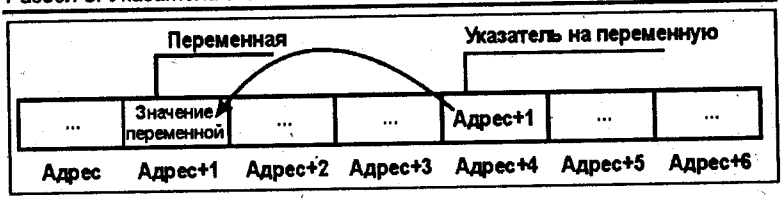


Рис. 5.2. Смысл указателя

Рассмотрим пример объявления и инициализации указателя.

```
char Symbol = 'Y';
char *pSymbol = &Symbol;
long Capital = 304L;
long* pLong;
pLong = &Capital;
```

В приведенном фрагменте объявляется символьная переменная Symbol и инициализируется значением 'Y', затем определяется указатель на символьный тип данных pSymbol, значение которого назначается равным адресу переменной Symbol. Вслед за этим объявляется переменная Capital типа long и указатель на этот же тип pLong, после чего производится инициализация указателя адресом переменной Capital.

Тема 5.2
Разыменование указателей

Как уже отмечалось выше, указатели помогают осуществлять непосредственный доступ к памяти. Для того чтобы получить (прочитать) *значение*, записанное в некоторой области, на которую ссылается указатель, используют операцию косвенного обращения, или *разыменования* (*). При этом используется имя указателя со звездочкой перед ним:

```
long double Num = 10;
long double Flag;
long double *ptr = &Num;
Flag = *ptr;
cout << Flag;
```

В приведенном фрагменте объявляются две переменные двойной точности Num и Flag и указатель (ptr) на тип long double, проинициализированный адресом переменной Num. После этого посредством косвенного доступа к переменной Flag присваивается значение, хранящееся по адресу, указанному в ptr, то есть факти-

чески значение переменной Num, что и подтверждает вывод на печать.

На практике довольно широко применяется так называемый *пустой указатель* (типа void), который может указывать на объект любого типа. Для получения доступа к объекту, на который ссылается указатель void, его необходимо предварительно привести к тому же типу, что и тип самого объекта.

Рассмотрим пример, иллюстрирующий использование пустого указателя.

```
#include <iostream.h>
int main()
{
  char Let = 'T';
  int nNum = 9;
  void *ptr;

  ptr = &Let;
  *(char*)ptr = 'L';
  ptr = &nNum;
  *(int*)ptr = 43;

  cout << Let << '\n';
  cout << nNum;

  return 0;
}
```

Сначала создаются два разнотипных объекта Let и nNum и пустой указатель ptr, который инициализируется адресом символьной переменной. Далее ptr разыменовывается, приводится к типу char* и посредством косвенной адресации модифицируется значение символа Let. Аналогичным образом указатель инициализируется значением адреса переменной nNum, приводится к целочисленному типу, после чего становится возможным изменение содержимого переменной nNum.

Тема 5.3
Арифметика указателей

К указателям (кроме указателей на переменные типа void) могут применяться арифметические операции. Для изменения пустого указателя он должен быть предварительно приведен к какому-либо типу (не void). Рассмотрим подробнее суть арифметических операций для указателей.

Компилятор, зная тип указателя, вычисляет размер переменной этого же типа, после чего модифицирует адрес, содержащийся в указателе в соответствии с заданной арифметической операцией, но с учетом вычисленного для данного типа размера. Это означает, что если объявлен указатель типа double, занимающего в памяти 8 байт, операция, например, инкремента указателя увеличит значение адреса не на один, а на восемь байт:

```
#include <iostream.h>
int main()
{
  double Var;
  double* ptr = &Var;
  cout << ptr << '\n';
  ptr++;
  cout << ptr;
  return 0;
}
```

В результате будет выведено, например:

```
0x0068fdfc
0x0068fe04
```

Указатели можно вычитать друг из друга, тем самым определяя количество элементов (того же типа, на который указывают указатели), расположенных между ними. Этот прием бывает полезным при операциях с массивами и символьными строками, которые будут рассмотрены позже.

Существует возможность использования при объявлении указателей ключевого слова const. При этом следует принимать во внимание, что применение данного спецификатора трактуется компилятором следующим образом:

```
// Указатель на константу типа char,
// разыменованное значение неизменно
const char* myKey;
// Константный указатель типа int,
// ВСЕГДА указывает на один и тот же адрес
int* const Cell;
```

Попытка модификации содержимого указателя на константу, как и применение любой арифметической операции к константному указателю, приведет к сообщению об ошибке.

В табл. 5.1 приводится перечень допустимых арифметических операций над указателями.

Таблица 5.1

Арифметические операции над указателями

Наименование операции	Пример
Сравнение на равенство	p1 == p2
Сравнение на неравенство	p1 != p2
Сравнение на меньше	p1 < p2
Сравнение на меньше или равно	p1 <= p2
Сравнение на больше	p1 > p2
Сравнение на больше или равно	p1 >= p2
Вычисление числа элементов между указателями	p2 - p1
Вычисление указателя, отстоящего от заданного на определенное в n число элементов	p1 + n p1 – n

Тема 5.4
Применение к указателям оператора sizeof

Как и к любой переменной или типу данных, к указателям можно применять операцию определения размера sizeof. Выше отмечалось, что размер указателя может принимать одно из двух значений: два или четыре байта, что позволяет указателю адресовать $2^{(2*8)}$ = 65 Кбайт или $2^{(4*8)}$ = 4 Гбайта памяти соответственно. На размер указателя (2 или 4 байта) влияет выбранная модель памяти и ряд других причин, которые будут рассмотрены в разделах, посвященных моделям памяти и модификаторам.

К указателям можно применять не только оператор sizeof, но и одноименную функцию, что и демонстрирует следующий пример.

```
#include <iostream.h>
int main()
{
  char endl = '\n';
  unsigned long ulCone = 546213;
  bool IsTrue = false;
  unsigned long* pUL = &ulCone;
  bool* pBool = &IsTrue;
  cout << sizeof pUL << endl;
  cout << sizeof(pUL) << endl;
```

```
    cout << endl;
    cout << sizeof pBool << endl;
    cout << sizeof(pBool) << endl;

    return 0;
}
```

В начале программы объявляются целочисленная и логическая переменные, а также соответствующие указатели, после чего выводится информация о размере указателей.

Тема 5.5
Указатели на указатели

Указатели могут сами ссылаться на другие указатели. При этом в ячейках памяти, на которые ссылается указатель, содержится не значение, а адрес какого-либо объекта. Сам объект может также являться указателем и т.д. На рис. 5.3 представлен вариант размещения в памяти указателя на указатель, который, в свою очередь, ссылается на однобайтный тип данных (например, char или bool).

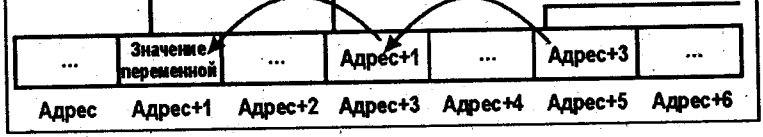


Рис. 5.3. Размещение указателя на указатель в памяти

Здесь по Адресу+1 хранится значение некоторой переменной, на которую указывает обычный указатель, расположенный по Адресу+3 (хранит значение Адрес+1), на который, в свою очередь, ссылается указатель, расположенный по Адресу+5 (содержит в качестве значения Адрес+3).

Синтаксис указателя на указатель выглядит следующим образом:

```
// Объявление указателя на указатель
int **pPtrInt;
// ppSymbol – указатель на указатель,
// который сам является указателем
char ***ppSymbol;
```

При объявлении указатель на указатель может инициализироваться адресом объекта:

```
char    *pSymb = &myChar;
char    **p_ptr = &pSymb;
char    ***ppptr = &p_ptr;
```

Число символов «звездочка» (*) при объявлении говорит о «порядке» указателя. Чтобы получить доступ к значению, такой указатель должен быть разыменован соответствующее количество раз (по числу символов «*»).

В приведенном ниже примере создается указатель второго порядка pPDX, содержащий адрес указателя pDX, который, в свою очередь, ссылается на переменную двойной точности.

```
#include <iostream.h>
int main()
{
  double dX = 903.456;
  double *pDX = &dX;
  double **pPDX = &pDX;
  cout << **pPDX;
  return 0;
}
```

В результате будет выведено значение переменной dX.

Тема 5.6
Указатели на функции

В C++ указатели могут ссылаться на функции. Имя функции само по себе представляет константный указатель на эту функцию, то есть содержит адрес входа в нее. Однако можно задать свой собственный указатель на данную функцию:

```
тип (*имя_указателя)(список_типа_аргументов)
```

Например,

```
bool (*MyFuncPtr)(char, long);
```

объявляет указатель MyFuncPtr, ссылающийся на функцию, возвращающую логическое значение и принимающую в качестве параметров одну символьную и одну целую длинную переменную.

Перед самым первым вызовом функции через указатель она обязательно должна быть инициализирована именем самой функции. При этом вызов через указатель осуществляется так, будто имя указателя является просто именем вызываемой функ-

ции. То есть после имени указателя следует список аргументов, ожидаемых функцией. Так, для приведенного выше указателя на функцию MyFuncPtr вызов может выглядеть, например, следующим образом:

```
bool bVar;
char Symbol = 'x';
long lNum = 0L;
bVar = MyFuncPtr(Symbol, lNum);
```

Рассмотрим пример использования указателя на функцию. Пусть задано какое-либо целое число, характеризующее месяц в году. Требуется определить, к какому сезону года относится данный месяц.

```
#include <iostream.h>
bool IsSpring(int);
bool IsSummer(int);
bool IsAutumn(int);
bool IsWinter(int);
int main()
{
  int Month = 4;
  bool (*pFunc)(int) = IsWinter;
  if(pFunc(Month))
    cout << "Зима";
  pFunc = IsAutumn;
  if(pFunc(Month))
    cout << "Осень";
  pFunc = IsSummer;
  if(pFunc(Month))
    cout << "Лето";
  pFunc = IsSpring;
  if(pFunc(Month))
    cout << "Весна";
  return 0;
}
bool IsSpring(int x)
{
     return (x>2 && x<6);
}
bool IsSummer(int x)
{
     return (x>5 && x<9);
}
bool IsAutumn(int x)
{
```

```
    return (x>8 && x<12);
}
bool IsWinter(int x)
{
    return (x>11 && x<3);
}
```

В приведенной программе создается указатель (pFunc) на функцию (принимающую один целочисленный аргумент и возвращающую логическое значение), инициализированный адресом функции IsWinter. Далее через указатель поочередно вызываются функции, определяющие, попадает ли принятый аргумент в заданный диапазон. Возвращаемые функциями логические значения определяют, выводить ли на экран сообщение.

Указатели на функции используются часто в качестве аргументов других функций. Таким образом создаются универсальные функции, значительно упрощающие текст сложной программы (например, численное решение уравнений, дифференцирование, интегрирование). Некоторые библиотечные функции в качестве параметра также принимают указатели на функции.

Тема 5.7
Ссылки

Ссылка – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается. Иными словами, он может использоваться просто как другое имя, или псевдоним объекта. При объявлении ссылки перед ее именем ставится знак амперсанда, а сама она должна быть тут же проинициализирована именем того объекта, на который ссылается:

```
тип &имя_ссылки = имя_переменной;
```

Тип объекта, на который указывается ссылка, может быть любым. Объявление неинициализированной ссылки вызовет сообщение компилятора об ошибке (кроме ситуации, когда ссылка объявляется как extern). Рассмотрим пример объявления ссылок:

```
char Letter_A = 'A';
char &ref = Letter_A;
```

Здесь объявляется и инициализируется символьная переменная Letter_A и ссылка на нее ref.

Любое изменение значения ссылки повлечет за собой изменение того объекта, на который данная ссылка указывает:

```
int i=0;
int &ref = i;
ref += 10;    // то же, что i += 10;
```

После выполнения приведенного фрагмента значение обеих переменных i и ref будет равно 10.

Использование ссылок не связано с дополнительными затратами памяти.

Следует отметить, что ссылки нельзя переназначать – инициализировав ссылку однажды адресом некоторой переменной, любое действие со ссылкой сказывается на самом объекте. Попытка переназначить имеющуюся ссылку какой-либо другой переменной приведет к присвоению оригиналу объекта значения второй переменной:

```
char letA = 'A';
char &refA = letA;
char B_Letter = 'B';
refA = B_Letter; // то есть letA = B_Letter
```

Кроме того, следует учесть, что ссылаться можно только на сам объект. Нельзя объявить ссылку на тип объекта. Ниже приводится корректный вариант объявления.

```
bool Flag = true;
bool &ref = Flag; // а не &ref = bool
```

Еще одно ограничение, налагаемое на ссылки, заключается в том, что они не могут указывать на нулевой объект (принимающий значение NULL). Таким образом, если есть вероятность того, что объект в результате работы приложения станет нулевым, от ссылки следует отказаться в пользу применения указателя.

Тема 5.8
Передача параметров по ссылке и по значению

Параметры в функцию могут передаваться одним из следующих способов:

- по значению;
- по ссылке.

При передаче аргументов *по значению* компилятор создает временную копию объекта, который должен быть передан, и размещает ее в области стековой памяти, предназначенной для хранения локальных объектов. Вызываемая функция оперирует именно с этой копией, не оказывая влияния на оригинал объекта.

Прототипы функций, принимающих аргументы по значению, предусматривают в качестве параметров указание типа объекта, а не его адреса. Например, функция

```
int GetMax(int, int);
```

принимает два целочисленных аргумента по значению.

Если же необходимо, чтобы функция модифицировала оригинал объекта, используется передача параметров *по ссылке*. При этом в функцию передается не сам объект, а только его адрес. Таким образом, все модификации в теле функции переданных ей по ссылке аргументов воздействуют на объект. Принимая во внимание тот факт, что функция может возвращать лишь единственное значение, использование передачи адреса объекта оказывается весьма эффективным способом работы с большим числом аргументов, которые сохраняют модифицированные функцией значения при выходе из ее тела. Кроме того, так как передается адрес, а не сам объект, существенно экономится стековая память.

В C++ передача по ссылке может осуществляться двумя способами:

- используя непосредственно ссылки;
- с помощью указателей.

Синтаксис передачи с использованием ссылок подразумевает применение в качестве аргумента ссылки на тип объекта. Например, функция

```
double Glue(long& var1, int& var2);
```

получает две ссылки на переменные типа long и int. При передаче в функцию параметра-ссылки компилятор автоматически передает в функцию адрес переменной, указанной в качестве аргумента. Ставить знак амперсанда перед аргументом в вызове функции не нужно. Например, для предыдущей функции вызов с передачей параметров по ссылке выглядит следующим образом:

```
Glue(var1, var2);
```

Пример прототипа функции при передаче параметров через указатель приведен ниже:

```
void SetNumber(int*, long*);
```

Кроме того, функции могут возвращать не только значение некоторой переменной, но и указатель или ссылку на него. Например, функции, прототип которых:

```
*int Count(int);
&int Increase();
```

возвращают указатель и ссылку соответственно на целочисленную переменную типа int. Следует иметь в виду, что возвращение ссылки или указателя из функции может привести к проблемам, если переменная, на которую делается ссылка, вышла из области видимости. Например,

```
int& func()
{
  int x;
  return x;
}
```

В этом случае попытка возвратить ссылку на локальную переменную x приведет к ошибке, которая, к сожалению, выяснится только в ходе выполнения программы.

Эффективность передачи адреса объекта вместо самой переменной ощутима и в скорости работы, особенно если используются большие объекты, в частности *массивы* (будут рассмотрены позже).

Если требуется передать в функцию объект, занимающий достаточно большой объем памяти, без модификации этого объекта в дальнейшем, используется вызов с ключевым словом const. Примером может служить функция передачи константного указателя.

Данный тип

```
const int* FName(int* const Number)
```

принимает и возвращает указатель на константный объект типа int. Любая попытка модифицировать такой объект в пределах тела вызываемой функции вызовет сообщение компилятора об ошибке. Рассмотрим пример, иллюстрирующий использование константных указателей.

```
#include <iostream.h>
int* const call(int* const);
int main()
{
  int X = 13;
  int* pX = &X;
  call(pX);
  return 0;
}
int* const call(int* const x)
{
  cout << *x;
```

```
  //*x++; // нельзя модифицировать объект!
  return x;
}
```

Вместо приведенного выше синтаксиса константного указателя в качестве альтернативы при передаче параметров можно использовать константные ссылки, например:

```
const int& FName(const int& Number)
```

имеющие тот же смысл, что и константные указатели.

```
#include <iostream.h>
const int& call(const int& x)
{
  cout << x;
  //x++; // нельзя модифицировать объект!
  return x;
}
int main()
{
  int X = 13;
  int& rX = X;
  call(rX);
  return 0;
}
```

Тема 5.9
Использование указателей и ссылок с ключевым словом const

Некоторые конструкции языка C++ являются источником путаницы. Одной из таких конструкций является использование ключевого слова const с указателями и ссылками. Следующие примеры помогут вам прояснить ситуацию.

```
// Объявление данных
int number;
const int count=0;
// Указатель является константой
int* const n1=&number;
// Указатель указывает на константу
// (указываемое значение есть const)
const int* n2=&count;
// И указатель, и указываемое значение
// являются константами
```

```
const int* const n3=&count;

// Указатели на строки, строка является константной
const char* str1="text";

// Указатель на строку является константой
char* const str2="text";

// Указатель и сама строка – константы
const char* const str3="text";

/* Массивы указателей на символы */
// Символы являются константами
const char* text1[]={"line1","line2","line3"};

// Указатели являются константами
char* const text2[]={"line1","line2","line3"};

// Указатели и символы являются константами
const char* const text3={"st1","st2","st3"};
```

Практикум «Указатели и ссылки»

Упражнение 5.1
Пример работы с указателями

Проанализируйте предлагаемый ниже пример, в котором объявляются две целочисленные переменные и по одному указателю на каждую из них. Далее значения переменных модифицируются посредством разыменования соответствующих указателей, и полученное значение выводится на экран.

```
#include <iostream.h>

int main()
{
   int i = 123;
   short s = 52;
   int *pi;
   short *ps;
   pi = &i;
   ps = &s;
   *pi += 5;
   *pi += *ps;
   cout << *pi;

   return 0;
}
```

Упражнение 5.2
Сравнение указателей

Проанализируйте пример, в котором объявляются две целочисленные переменные и указатели на них. Адреса переменных сравниваются посредством указателей, и выводится информация об их месторасположении.

```
#include <iostream.h>
int main()
{
   int i = 123;
   int s = 52;
   int *pi;
   int *ps;
   pi = &i;
   ps = &s;
   if(pi < ps)
      cout << "Значение " << pi << " лежит выше в стеке, "\
      "чем значение " << ps;
   else if(ps < pi)
      cout << "Значение " << pi << " лежит ниже в стеке, "\
      "чем значение " << ps;
   return 0;
}
```

Модифицируйте данный пример таким образом, чтобы значения переменных сравнивались в функции, выводящей результат на экран.

Упражнение 5.3
Размер указателя

Составьте текст программы, в которой выводятся значения размера указателя на все известные Вам базовые типы данных. Обоснуйте полученные результаты.

Упражнение 5.4
Работа с указателями на функции

Напишите функцию, которая будет подсчитывать число сброшенных битов в однобайтовом передаваемом ей аргументе. В главной функции программы создайте указатель на получен-

ную функцию, через который осуществите вызов для трех различных значений.

Упражнение 5.5
Передача параметров по ссылке и указателю

Проанализируйте предлагаемый ниже пример. Пусть необходимо в одной функции производить сложение и перемножение двух аргументов и возвращать полученные значения. Для этого составим функцию MyFunc(int& x, int& y, int& sum, int& mult). Тогда при ее вызове вместо значений в функцию будут передаваться лишь адреса объектов для вычисления. Результат выполнения арифметических операций (значения суммы и произведения) присвоим аргументам sum и mult соответственно.

```
#include <iostream.h>
void MyFunc(int& x, int& y, int& sum, int& mult)
{
   sum = x + y;
   mult = x * y;
}
int main()
{
   int A, B;
   int Sum, Mult;
   cout << "A = "; cin >> A;
   cout << "B = "; cin >> B;
   MyFunc(A, B, Sum, Mult);
   cout << "A + B = " << Sum << '\n';
   cout << "A * B = " << Mult;
   return 0;
}
```

Преобразуйте приведенный пример так, чтобы вместо передачи параметров по ссылке происходила передача данных в функцию по указателю.

Раздел 6
Модификаторы

Существует ограниченное число задач, решение которых требует минимального доступа к системным ресурсам вычислительного средства. На практике чаще всего приходится учитывать особенности архитектуры компьютера с тем, чтобы разрабатываемое приложение было максимально гибким. Основные ограничения на использование системных средств ЭВМ накладывает организация памяти.

Память компьютеров на базе процессоров семейства Intel 80x86 представляется в виде *сегментов* или частей от 0 байт до 64 Кбайт.

Как известно, в состав процессора входят *регистры* – быстро доступные ячейки собственной памяти. Для различных арифметических операций используются *регистры общего назначения:* AH, AL, BH, BL, CH, CL, DH и DL, которые в случае необходимости могут группироваться в сдвоенные: AX (из AH и AL), BX (из BH и BL), CX (из CH и CL) и DX (из DH и DL).

Доступ к ячейке *оперативной памяти* организован через обращение к значению сегмента и указание смещения внутри данного сегмента. Процессор имеет четыре *сегментных регистра*, посредством которых осуществляется манипулирование памятью:

- CS (сегмент кода) предназначен для обеспечения адресации к сегменту, содержащему исполняемые программные коды;
- DS (сегмент данных) используется для адресации глобальных и статических переменных;
- ES (дополнительный сегментный регистр), как и DS, содержит сегментное значение для доступа к глобальным и статическим переменным; является вспомогательным регистром;
- SS (сегмент стека) служит для адресации к сегментному значению стека программы и, как следствие, к локальным переменным.

При адресации памяти в DOS процессор преобразует значение сегментного регистра и смещения в линейный адрес. В случае работы в операционной системе Windows сегментный регистр

служит в качестве селектора дескрипторной таблицы, содержащей действительный адрес.

К счастью, компиляторы ведущих фирм предусматривают так называемые расширения языка, помогающие обходить те или иные ограничения системы. В этом разделе мы познакомимся с основными из них.

Тема 6.1
Модели памяти

Доступ к переменным или функциям в программе также осуществляется через сегментные регистры. В регистр DC заносится базовое значение, после чего процессору тем или иным способом передается значение смещения. Поскольку размер сегмента не должен превышать 64 Кбайт, очень вероятно, что в реальности коды или данные будут располагаться в различных сегментах. Таким образом, каждый раз при обращении к переменным, расположенным в различных сегментах, программе необходимо будет загружать в регистр DC новое значение сегмента. Совершенно очевидно, что для обращения к разным сегментам требуется гораздо больше времени, так как каждый раз приходится загружать новое базовое значение. В лучшем случае, все данные или все коды программы будут умещаться в один единственный сегмент и для доступа к любой переменной или функции будет достаточно загрузить смещение лишь один раз. Однако в данном случае размер программы был бы ограничен величиной одного сегмента, то есть 64 Кбайтами. В настоящее время такое ограничение неприемлемо за редким исключением, да чаще всего и не нужно.

Для того чтобы имелась возможность выбора оптимальной конфигурации конечного программного продукта, C++ предлагает к использованию различные модели памяти. Модели памяти служат для управления выделением ресурсов при выполнении разработанного приложения. Существует шесть моделей памяти: Tiny, Small, Medium, Compact, Large и Huge.

DOS приложения могут использовать любую из перечисленных моделей, в то время как программы, разработанные для функционирования в 16-разрядной ОС Windows (например, Windows 3.11), только одну из четырех: Small, Medium, Compact или Large. В сущности, выбор одной из них задается определенным ключом компилятора, однако может меняться для той или иной функции или переменной опционально в программе (по мере надобности).

Помимо определения требований, предъявляемых разрабатываемой программой к объему памяти, выбор определенной модели влияет на размер указателей, динамической памяти, а для DOS программ – еще и на размер области стека. В связи с вышеизложенным указатели бывают ближними и дальними.

Ближний указатель содержит только значение смещения, в то время как значение сегмента адреса находится в сегментном регистре. Этот тип указателей используется по умолчанию в моделях с ближней динамической памятью Tiny, Small и Medium. При этом сегментная часть адреса данных размещается в регистре DS. Программный код размером до 64 Кбайт применяет модель с короткими указателями кода (Tiny, Small или Compact), размещая сегментную часть адреса в регистре CS. Таким образом, функция или переменная будет доступна только в том сегменте кода, в котором она откомпилирована.

Дальние указатели содержат как сегментную часть адреса, так и смещение. Этот тип указателей используется по умолчанию в моделях памяти Compact, Large и Huge – для данных, и в Medium, Large и Huge – для кодов программы.

В модели Tiny в регистры CS, DS, SS и ES заносится одинаковое значение сегментной части адреса. Код программы, статические, динамические данные и стек в общей сложности могут занимать всего лишь 64 Кбайта. По умолчанию переменная типа указатель в такой модели памяти занимает два байта (ближний указатель) и содержит смещение внутри данного сегмента памяти. Очень небольшое число задач может решаться при использовании такой модели – ее применяют при существенном недостатке памяти.

Модель Small предусматривает код программы размером до 64 Кбайт (один сегмент) и еще один сегмент под статические, динамические данные и стек. Указатели в такой модели также занимают два байта и содержат смещение внутри используемого сегмента. Эта модель применяется для решения небольших и средних задач.

В модели Medium код разрабатываемого приложения может занимать до 1 Мбайта (1024 Кбайт), в то время как данные и стек размещаются в пределах одного сегмента. Таким образом, в коде программы используются дальние указатели, занимающие четыре байта. Адресация данных подразумевает использование ближних указателей (по два байта). Эту модель следует использовать при разработке больших приложений, оперирующих с небольшим количеством данных.

Модель Compact отводит под код программы один сегмент (до 64 Кбайт) и по одному сегменту под статические данные и стек. Для адресации в коде программы задействуют ближние указатели, в то время как адресация данных осуществляется посредством дальних указателей. Компактная модель применяется при разработке небольших и средних приложений, требующих большого объема статических данных.

В модели Large код программы занимает до 1 Мбайта, а статические данные со стеком – по одному сегменту до 64 Кбайт. В модели используются дальние четырехбайтные указатели и для адресации кода и для адресации данных. Модель Large часто используют на практике при решении больших задач.

Модель Huge допускает превышение статическими данными объема 64 Кбайт. В остальном эта модель аналогична модели Large и использует четырехбайтные дальние указатели. Модели памяти 16-разрядных приложений представлены в табл. 6.1.

Таблица 6.1

Модели памяти 16-разрядных приложений

Модель	Код	Данные	Стек	Указатель по умолчанию
DOS				
Tiny	код + данные + стек + динамическая память до 64 Кбайт			near
Small	64 Кбайт	данные + стек + динамическая память до 64 Кбайт		near
Medium	1 Мбайт	данные + стек + динамическая память до 64 Кбайт		near
Compact	64 Кбайт	64 Кбайт	64 Кбайт	far
Large	1 Мбайт	64 Кбайт	64 Кбайт	far
Huge	1 Мбайт	> 64 Кбайт	64 Кбайт	far
16-bit Windows EXE				
Small	64 Кбайт	данные + стек + динамическая память до 64 Кбайт		near
Medium	1 Мбайт	данные + стек + динамическая память до 64 Кбайт		near

Модель	Код	Данные	Стек	Указатель по умолчанию
Compact	64 Кбайт	данные + стек до 64 Кбайт		far
Large	1 Мбайт	данные + стек до 64 Кбайт		far
16-bit Windows DLL				
Small	64 Кбайт	64 Кбайт	—	far
Medium	1 Мбайт	64 Кбайт	—	far
Compact	64 Кбайт	64 Кбайт	—	far
Large	1 Мбайт	64 Кбайт	—	far
32-bit Windows				
Flat	2 Гбайт	1 Мбайт	1 Мбайт	far

Для изменения длины указателей, применяемой по умолчанию, используются модификаторы:

near – для генерирования компилятором ближних вызовов;
far – для генерирования дальних вызовов.

Следует понимать, что ближний указатель всегда может быть преобразован компилятором в дальний, однако дальний указатель не может быть преобразован в ближний. Синтаксис спецификаторов вызова имеет следующий вид:

```
тип near *имя_переменной;
тип far *имя_переменной;
```

что читается как «объявить указатель *имя_переменной типа 'тип' как ближний / дальний». Если же поместить модификатор дальнего или ближнего вызова между символом звездочка (*) и именем переменной, компилятор поймет такую запись в ином контексте и в результате объявит указатель на переменную в том же (near) или обязательно в другом (far) сегменте (то есть модификатор в этом случае влияет на месторасположение указателя, а не на его размер).

Необходимо отметить, что модификаторы в библиотеках компиляторов различных фирм могут в синтаксисе включать ведущие символы подчеркивания (один или два). Так, модификаторы

```
near
_near
__near
```

подразумевают одно и то же. В дальнейшем изложении данного материала нет жесткой привязки к какому-либо типу написания, однако следует иметь этот факт в виду и руководствоваться при разработке приложений технической документацией к конкретному компилятору. Кроме того, модификаторы с одним ведущим символом подчеркивания зачастую используются компилятором для внутренних нужд, и пользоваться ими не рекомендуется.

Ниже приводится несколько вариантов применения модификатора:

```
// Символьная переменная в сегменте по умолчанию:
char A = 'a';

// Символьная переменная в дальнем сегменте данных:
char far B = 'b';

// Указатель на символьную переменную,
// размещается в сегменте по умолчанию:
char* pA = &A;

// Дальний указатель на символьную переменную A,
// размещается в сегменте данных по умолчанию:
char far* pfA = &A;

// Обычный указатель на символьную переменную,
// размещаемый в дальнем сегменте данных:
char* far pB1 = &B;

// Дальний указатель на символьную переменную,
// размещаемый в дальнем сегменте данных:
char far* far pB2 = &B;
```

Тема 6.2
Модификатор huge

Если проводятся некоторые арифметические операции над указателями, они изменяют только значение смещения, оставляя значение сегмента неизменным. В этом случае для указателей, расположенных, например, в конце сегмента, возможна ситуация, когда при увеличении указателя вместо обращения к значению другого сегмента произойдет переход через границу последнего и будет произведено обращение к нулевому смещению того же самого сегмента. То же утверждение верно для указателей, расположенных в начале сегмента. Для того чтобы компилятор сгенерировал код, позволяющий указателям выходить за текущий сегмент (*нормализация указателя*), применяется модификатор huge. В этом случае синтаксис объявления указателя имеет следующий вид:

```
тип huge *имя_указателя
```

Приведенный ниже пример демонстрирует использование модификатора huge.

```
#include <stdio.h>
int main()
{
  long huge *p = NULL;
  for(int i=0; i<10; i++, p++)
    printf("p содержит адрес %Fp\n", p);
  return 0;
}
```

В рассмотренной программе подключается заголовочный файл stdio.h, содержащий прототип функции форматированного вывода данных printf() – здесь с ее помощью производится отображение указателя в виде СЕГМЕНТ:СМЕЩЕНИЕ. Далее определяется указатель типа long huge и в цикле производится инкремент и вывод 10 значений адресов дальних указателей. Из результатов работы программы видно, что осуществляется нормализация указателя, и величина смещения не превышает значения 15, в то время как значение сегмента изменяется.

Во время выполнения приложения неизбежны вызовы каких-либо функций. Параметры функций и переменные, объявленные внутри, являются локальными объектами и располагаются внутри стека. Кроме того, в стеке хранится информация (например, адрес возврата, состояние программы до вызова функции, выполняющейся в текущий момент), необходимая программе при возврате из вызываемых функций.

Глобальные же объекты хранятся в сегменте данных, а так как размер данных для всех моделей памяти, кроме huge, не может превышать одного сегмента, количество глобальных переменных в программе ограничено. Если компилятор при сборке приложения указывает, что используется слишком много глобальных объектов, часть из них можно описать как дальние (far). Если же необходимо определить переменную, о которой изначально известно, что ее размер выходит за пределы 64 Кбайт, используют модификатор huge.

Так, например, глобальный массив большого размера можно описать следующим образом:

```
long huge ARR[30000];
```

Таким образом, с помощью спецификаторов far и huge можно определять переменные, если изначально известно (еще при проектировании), что размер приложения будет выходить за рамки отведенных 64 Кбайт.

Тема 6.3
Модификаторы функций

В зависимости от того, какая модель памяти применяется, функции и указатели на функцию являются ближними или дальними. Как и в случае с указателями на переменные, размер указателя на функцию можно изменить с помощью модификаторов near и far. Так, например, функции, обращение к которым осуществляется операционной системой или драйвером какого-либо устройства, всегда должны быть объявлены как дальние (far) и, если используется модель памяти с короткими указателями для кода (Tiny, Small или Compact), такие функции должны специфицироваться с помощью модификатора far.

Рассмотрим пример использования модификаторов функций.

```
#include <iostream.h>
void __near PrintNear(char*);
void __far PrintFar(char*);
void (__near* nPrint)(char*) = PrintNear;
void (__far* fPrint)(char*) = PrintFar;

int main()
{
  nPrint("NEAR");
  fPrint("FAR");

  return 0;
}
void __near PrintNear(char* nMess)
{
  cout << nMess << '\n';
  cout << "Вызов ближней функции\n";
}
void __far PrintFar(char* fMess)
{
  cout << fMess << '\n';
  cout << "Вызов дальней функции\n";
}
```

После объявления функций PrintNear() и PrintFar() следует определение указателей на эти функции короткого nPrint и длинного

fPrint соответственно. Тело функции main() содержит вызовы ближней и дальней функций через их указатели.

Тема 6.4
Модификаторы cdecl и pascal

Как уже упоминалось ранее, при вызове некоторой функции ее параметры обычно помещаются в стек, а при возврате из функции параметры из стека удаляются. Помещение и удаление параметров из стека осуществляется в соответствии с определенными правилами. Существует два варианта помещения аргументов в стек: слева направо (соглашение о вызове языка Pascal) и справа налево (соглашение о вызове языка C). По умолчанию компилятор использует соглашение, указанное в параметрах (при вызове из командной строки или в опциях интегрированной среды разработки приложения), однако программист всегда может явно указать для каждой функции, какое из соглашений вызова необходимо применять. Это осуществляется с помощью модификаторов cdecl и pascal.

Оба модификатора соответствующим образом влияют на внутреннее имя функции при декорировании имен (будет рассматриваться ниже), тем самым сообщая компилятору использовать то или иное соглашение о вызове.

Модификатор cdecl указывает компилятору на то, что параметры вызываемой функции должны помещаться в стек в порядке, обратном следованию при вызове, то есть справа налево (первым передается аргумент, стоящий последним в списке параметров, затем предыдущий и т.д.). При этом не делается никаких предположений об ответственности функции за очистку стека. Внутреннее имя функции эквивалентно объявляемому имени с добавлением символа подчеркивания и использованием соглашения языка C о различении верхнего и нижнего регистров:

```
MyFunction – имя из прототипа функции
_MyFunction – внутреннее имя функции
```

Модификатор pascal, наоборот, требует прямой передачи параметров в стек, слева направо. Кроме того, данный спецификатор сигнализирует, что именно вызываемая функция ответственна за очистку стека. Данное соглашение используют на практике, если функция вызывается много раз из разных мест. Однако функции с переменным числом параметров не могут использо-

вать этого соглашения. Внутреннее имя при использовании данного вида соглашения совпадает с именем прототипа, но с преобразованием его в верхний регистр:

```
MyFunction – имя из прототипа функции
MYFUNCTION – внутреннее имя функции
```

Если по умолчанию используется соглашение языка Pascal, следует об этом помнить, так как имя главной функции main() будет преобразовано не в _main, а в MAIN, что может вызвать сообщение компилятора об ошибке, обойти которую поможет использование модификатора cdecl:

```
int __cdecl main()
{
  ...
  return 0;
}
```

Иногда необходимо ускорить вызов функции. Этого можно достичь путем передачи параметров функции не через стек, а непосредственно через регистры общего назначения, указав перед именем функции спецификатор fastcall. Поскольку число регистров ограничено, следует применять быстрый вызов только для функций, число параметров которых не больше трех. При этом тип аргументов должен соответствовать типу char, int или long либо быть коротким указателем. Внутреннее имя функции совпадает с именем ее прототипа с добавлением символа @:

```
MyFunction – имя из прототипа функции
@MyFunction – внутреннее имя функции
```

C++ предлагает еще и использование стандартного соглашения о вызове, которое может применяться только в 32-разрядных приложениях. Это своеобразный конгломерат соглашений языков C и Pascal, который специфицируется с помощью модификатора stdcall. При стандартном соглашении о вызове параметры помещаются в стек справа налево, как в случае с cdecl, однако вызванная функция сама отвечает за очистку стека. Внутреннее имя функции полностью совпадает с именем, указанным в прототипе:

```
MyFunction – имя из прототипа функции
MyFunction – внутреннее имя функции
```

Другим способом ускорения работы той или иной части функции является использование встроенного ассемблерного кода.

Ключевое слово asm определяет следующий за ним однострочный (до конца строки) или блочный (многострочный, заключенный в фигурные скобки) ассемблерный оператор, который имеет возможность взаимодействия с другими переменными, используемыми в программе. Ниже рассмотрен пример, позволящий двум переменным быстро обменяться своим содержимым:

```
#include <iostream.h>
int main()
{
  int var1 = 55;
  int var2 = 99;
  cout << "До обмена:\n";
  cout << var1 << '\n';
  cout << var2 << '\n';
asm
  {
    push var1
    push var2
    pop  var1
    pop  var2
  }
  cout << "После обмена:\n";
  cout << var1 << '\n';
  cout << var2;
  return 0;
}
```

В примере сначала объявляются и инициализируются две целочисленные переменные, а затем выводится их содержимое. Далее в блоке ассемблерной вставки первая, а затем и вторая переменная помещаются в стек, после чего в качестве первой переменной извлекается значение второй переменной (помещена в стек последней), а после этого из стека извлекается значение первой переменной с присвоением var2. Рис. 6.1 иллюстрирует обмен значений переменных посредством стека.

Следует, однако, помнить, что операции с ассемблерными вставками небезопасны: если в них используется изменение содержимого регистров общего назначения, это может неожиданным образом повлиять на ход выполнения всей программы. Для использования тех или иных регистров правильно будет при входе в ассемблерную вставку предварительно сохранить, например, в стеке, значения изменяемых регистров, а на выходе восстано-

вить исходное значение. Ниже рассмотрен тот же пример с использованием регистра общего назначения AX:

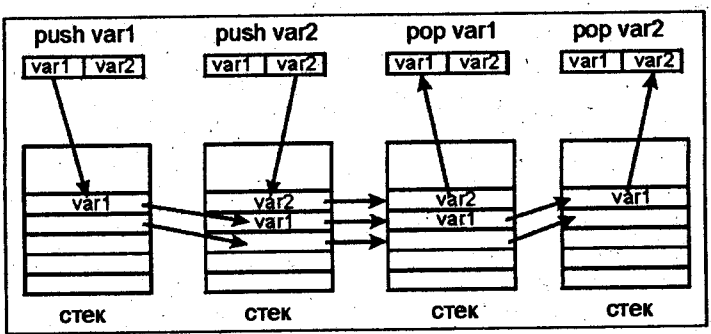


Рис. 6.1. Обмен значений переменных через стек

```
#include <iostream.h>
int main()
{
  int var1 = 55;
  int var2 = 99;
  cout << "До обмена:\n";
  cout << var1 << '\n';
  cout << var2 << '\n';
asm
  {
    // сохраняем в стеке первоначальное значение регистра AX
    push ax
    // осуществляем обмен значениями переменных
    mov ax, var1
    push ax
    mov ax, var2
    mov var1, ax
    pop var2
    // восстанавливаем первоначальное значение регистра AX
    pop ax
  }
  cout << "После обмена:\n";
  cout << var1 << '\n';
  cout << var2;
  return 0;
}
```

C++ также предусматривает непосредственные операции над регистрами и флагами процессора без использования ассемблерной вставки. Для этого зарезервированы следующие ключевые слова:

_AX	_AH	_AL	_BX	_BH	_BL
_CX	_CH	_CL	_DX	_DH	_DL
_SI	_DI	_BP	_SP	_ES	_SS
_CS	_DS	_FLAGS			

Кроме этого, если в опциях компилятора разрешено использование инструкций процессора Intel 80x386, в программе также можно использовать дополнительные ключевые слова:

_EAX	_ECX	_ESI	_ESP
_EBX	_EDX	_EDI	_EBP

Ниже приведен пример, демонстрирующий использование псевдорегистров.

```
#include <iostream.h>
int main()
{
  int var1 = 55;
  cout << _AX << '\n';
  cout << '\n';
  _AX = var1;
  cout << _AX << '\n';
  return 0;
}
```

Первоначально значение регистра AX неоднозначно, поэтому первый оператор вывода покажет какое-то неопределенное значение, далее идет присвоение регистру значения переменной var1 и вывод этого значения на экран. Следует учесть, что псевдорегистры нужно использовать с осторожностью, так как многие операции могут менять значения регистров еще до того, как в программе следует использование последних. Так, например, если в вышеприведенный пример после присвоения

```
_AX = var1;
```

добавить строку.

```
cout << "Привет!";
```

значение регистра будет изменено и на экране второй оператор вывода отобразит неопределенное значение.

Внутри блока ассемблерной вставки можно помещать метки и вызовы других функций программы. При этом перед вызовом функции необходимо самостоятельно размещать аргументы в стеке, а после вызова может потребоваться его (стека) очистка.

Следующий пример иллюстрирует вызов внешней функции из ассемблерной вставки (для модели памяти Large).

```
#include <iostream.h>
void ShowSum(int, int);
int main()
{
  int a = 2;
  int b = 3;
  asm
  {
    // Помещение параметров в стек
    push a
    push b
    // Вызов дальней функции (модель Large!)
    call far ptr ShowSum
    // Пример ближнего вызова: call near ptr ShowSum
    // Очистка стека
    pop cx
  }
  return 0;
}
void ShowSum(int x, int y)
{
  cout << (x + y);
}
```

Здесь в ассемблерной вставке следует помещение аргументов функции в стек, а также вызывается функция, выводящая сумму полученных аргументов.

В качестве иллюстрации использования ассемблерных вставок с метками рассмотрим пример, осуществляющий вывод на экран символов от A до Z:

```
#include <iostream.h>
int main()
{
  char letter = 'A';
  char stop = 'Z';
start:
  asm
  {
    mov ah, letter
    mov al, stop
    cmp ah, al
    jle print
    jmp exit
  }
print:
  cout << letter;
  letter++;
```

```
    cout << " ";
    goto start;
exit:
    return 0;
}
```

В этом примере символьной переменной letter присваивается значение символа 'A', переменной stop – значение символа 'Z'. Далее, в регистр ah заносится значение переменной letter, а в регистр al – значение переменной stop, после чего команда cmp ah, al сравнивает значения регистров между собой. Если ah <= al, осуществляется переход на метку print и переменная letter выводится на печать. Вслед за этим значение переменной letter увеличивается, и цикл повторяется с метки start. Если же значение ah больше значения al, следует выход из программы.

Практикум «Модификаторы»

Упражнение 6.1
Модификаторы cdecl и pascal

Составьте текст программы, в которой используется вызов двух функций с модификаторами cdecl и pascal.

Упражнение 6.2
Использование пространства имен

Проанализируйте предлагаемый ниже пример, иллюстрирующий использование пространства имен.

Пусть в некотором пространстве имен Pro имеется частица с зарядом, равным единице, а в пространстве имен Anti – частица с зарядом –1. Вычислим сумму зарядов на границе обоих пространств (пространство имен Hyper):

```
#include <iostream.h>
namespace Pro
{
   int Particle = 1;
}
```

```
namespace Anti
{
   int Particle = -1;
}
namespace Hyper
{
   int Charge = 123;
}
int main()
{
   using namespace Pro;
   Hyper::Charge = Particle;
   Hyper::Charge += Anti::Particle;
   cout << Hyper::Charge;
   return 0;
}
```

Раздел 7
Массивы

Тема 7.1
Понятие массива

В повседневной жизни постоянно приходится сталкиваться с однотипными объектами. Как и многие другие языки высокого уровня, C++ предоставляет программисту возможность работы с наборами однотипных данных – *массивами*. Отдельная единица таких данных, входящих в массив, называется *элементом массива*. В качестве элементов массива могут выступать данные любого типа (один тип данных для каждого массива), а также указатели на однотипные данные. Массивы бывают *одномерными* и *многомерными*.

Поскольку все элементы массива имеют один тип, они также обладают одинаковым размером. Использованию массива в программе предшествует его объявление, резервирующее под массив определенное количество памяти. При этом указывается тип элементов массива, имя массива и его размер в квадратных скобках. Размер сообщает компилятору, какое количество элементов будет размещено в массиве. Например:

```
int Array[20];
```

зарезервирует в памяти место для размещения двадцати целочисленных элементов.

Элементы массива в памяти располагаются непосредственно один за другим. На рис. 7.1 показано расположение одномерного массива двухбайтных элементов (типа short) в памяти.

Обращение к элементам массива может осуществляться одним из двух способов:

- по номеру элемента в массиве (через его индекс);
- по указателю.

При обращении через индекс за именем массива в квадратных скобках указывается номер элемента, к которому требуется вы-

полнить доступ. Следует помнить, что в C++ элементы массива нумеруются начиная с 0. Первый элемент массива имеет индекс 0, второй – индекс 1 и т.д. Таким образом, запись типа:

```
x = Array[13];
y = Array[19];
```

выполнит присвоение переменной x значения 14-го элемента, а переменной y – значение 20-го элемента массива.

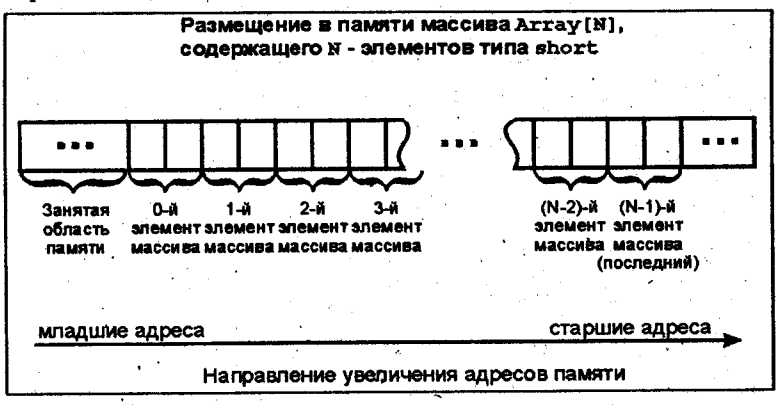


Рис. 7.1. Одномерный массив в памяти

Доступ к элементам массива через указатели заключается в следующем. Имя объявляемого массива ассоциируется компилятором с адресом его самого первого элемента (с индексом 0). Таким образом, можно присвоить указателю адрес нулевого элемента, используя имя массива:

```
char ArrayOfChar[] = {'W','O','R','L','D'};
char* pArr = ArrayOfChar;
```

Разыменовывая указатель pArr, можно получить доступ к содержимому ArrayOfChar[0]:

```
char Letter = *pArr;
```

Поскольку в C++ указатели и массивы тесно взаимосвязаны, увеличивая или уменьшая значение указателя на массив, программист получает возможность доступа ко всем элементам массива путем соответствующей модификации указателя:

```
// pArr указывает на ArrayOfChar[0] ('W')
pArr += 3;
// pArr указывает на ArrayOfChar[3] ('L')
pArr++;
// pArr указывает на ArrayOfChar[4] ('D')
char Letter = *pArr; // Letter = 'D';
```

Таким образом, после проведенных арифметических операций указатель pArr будет ссылаться на элемент массива с индексом 4. К этому же элементу можно обратиться иным способом:

```
Letter = *(ArrayOfChar + 4);
// Эквивалент Letter = ArrayOfChar[4];
```

Присвоение значений одного массива значениям другого массива вида Array[] = Another[] или Array = Another недопустимо, так как компилятор не может самостоятельно скопировать все значения одного массива в значения другого. Для этого программисту необходимо предпринимать определенные действия (при доступе к элементам по индексу чаще всего используется циклическое присвоение). Объявление вида

```
char (*Array)[10];
```

определяет указатель Array на массив из 10 символов (char). Если же опустить скобки, компилятор поймет запись как объявление массива из 10 указателей на тип char.

Рассмотрим пример использования массива.

```
#include <iostream.h>
int main()
{
  // Объявление целочисленного массива,
  // содержащего 5 элементов:
  short Number[5];
  char endl = '\n';
  // Заполнение всех элементов массива в цикле
  for(int i=0; i<5; i++)
    Number[i] = i;
  // Вывод содержимого с 3-го по 5-й элемента
  for(int i=2; i<5; i++)
    cout << Number[i] << endl;
  return 0;
}
```

Тема 7.2
Инициализация массивов

Инициализацию массивов, содержащих элементы базовых типов, можно производить при их объявлении. При этом непосредственно после объявления необходимо за знаком равенства (=) перечислить значения элементов в фигурных скобках через запятую (,) по порядку их следования в массиве. Например, выражение:

```
int Temp[12] = {2, 4, 7, 11, 12, 12, 13, 12, 10, 8, 5, 1};
```

проинициализирует массив температур Temp соответствующими значениями. Так, элемент Temp[0] получит значение 2, элемент Temp[1] – значение 4 и т.д. до элемента Temp[11] с присвоением ему значения 1.

Если в списке инициализации значений указано меньше, чем объявлено в размере массива, имеет место частичная инициализация. В этом случае иногда после последнего значения в инициализирующем выражении для наглядности ставят запятую:

```
int Temp[12] = {2, 4, 7, };
```

Таким образом, элементы Temp[0], Temp[1] и Temp[2] будут проинициализированы, в то время как оставшиеся элементы инициализации не получат.

При объявлении одномерного массива с одновременной его инициализацией разрешается опускать значение размера, обычно указываемое в квадратных скобках. При этом компилятор самостоятельно подсчитает количество элементов в списке инициализации и выделит под них необходимую область памяти:

```
// Выделение в памяти места для хранения
// шести объектов типа int (24 байта для 32-рзрядной системы)
int Even[] = {0, 2, 4, 6, 8, 10};
```

Если далее в программе потребуется определить, сколько элементов имеется в массиве, можно воспользоваться следующим выражением:

```
int Size = sizeof(Even) / sizeof(Even[0]);
```

Здесь выражение sizeof(Even) определяет общий размер, занимаемый массивом Even в памяти (в байтах), а выражение sizeof(Even[0]) возвращает размер (тоже в байтах) одного элемента массива.

В многомодульном проекте инициализация массива при объявлении производится лишь в одном из модулей. В других модулях получить доступ к элементам массива можно с помощью ключевого слова extern:

```
// Модуль first.cpp
...
char Hello[] = {'H','e','l','l','o'};
...
// Модуль second.cpp
extern Hello[];
```

```
...
// Модуль n.cpp
extern Hello[5];
...
```

Попытка повторной инициализации вызовет сообщение компилятора об ошибке.

Тема 7.3
Многомерные массивы

Многомерный массив (см. рис. 7.2 и рис. 7.3) размерности N можно представить как одномерный массив из массивов размерности (N-1). Таким образом, например, трехмерный массив – это массив, каждый элемент которого представляет двумерную матрицу. При этом мерность массива определяется числом парных квадратных скобок при его объявлении.

Примеры объявления многомерных массивов:

```
// Двумерный массив 6 x 9 элементов:
char Matrix2D[6][9];
// Трехмерный:
unsigned long Arr3D[4][2][8];
// Массив 7-й степени мерности:
my_type Heaven[22][16][7][47][345][91][3];
```

Выражение Array[idx][idy], представляющее двумерный массив, переводится компилятором в эквивалентное выражение:

```
*(*(Array+idx)+idy)
```

Многомерные массивы инициализируются в порядке наискорейшего изменения самого правого индекса (задом наперед): сначала происходит присвоение начальных значений всем элементам последнего индекса, затем предыдущего и т.д. до самого начала:

```
int Mass[3][2][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24};
```

В этом случае элементы массива будут получать следующие значения:

```
Mass[0][0][0] = 1;
Mass[0][0][1] = 2;
Mass[0][0][2] = 3;
...
Mass[0][1][0] = 5;
Mass[0][1][1] = 6;
...
```

```
Mass[1][0][0] = 9;
Mass[1][0][1] = 10;
...
Mass[1][1][0] = 13;
Mass[1][1][1] = 14;
...
Mass[2][0][0] = 17;
...Mass[2][1][0] = 21;
...
```

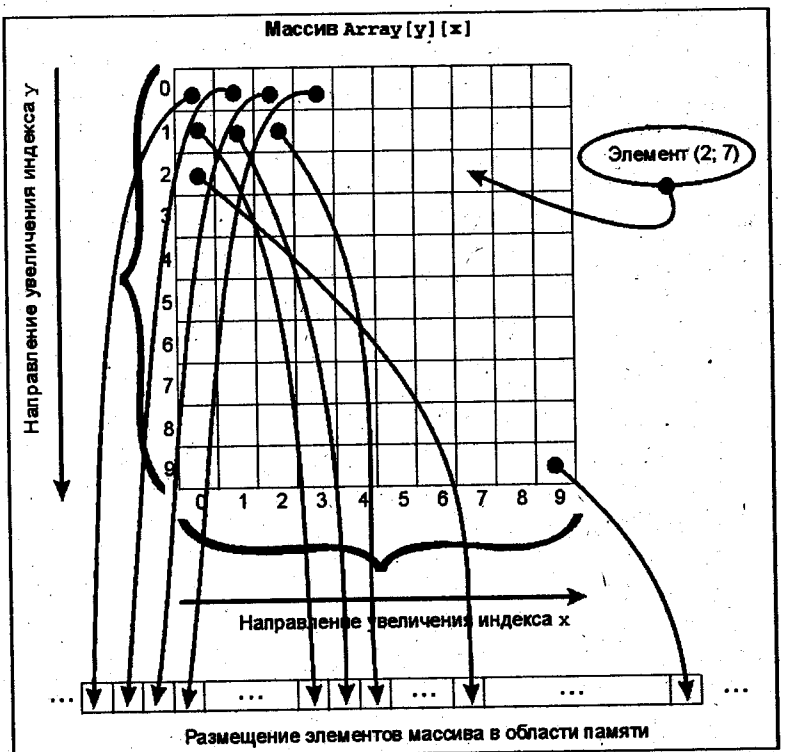


Рис. 7.2. Двумерный массив в памяти

Чтобы не запутаться, для наглядности можно группировать данные с помощью промежуточных фигурных скобок:

```
int Mass[3][2][4] = {{1,2,3,4},{5,6,7,8}},
                    {{9,10,11,12},{13,14,15,16}},
                    {{17,18,19,20},{21,22,23,24}};
```

Для многомерных массивов при инициализации разрешается опускать только величину первой размерности:

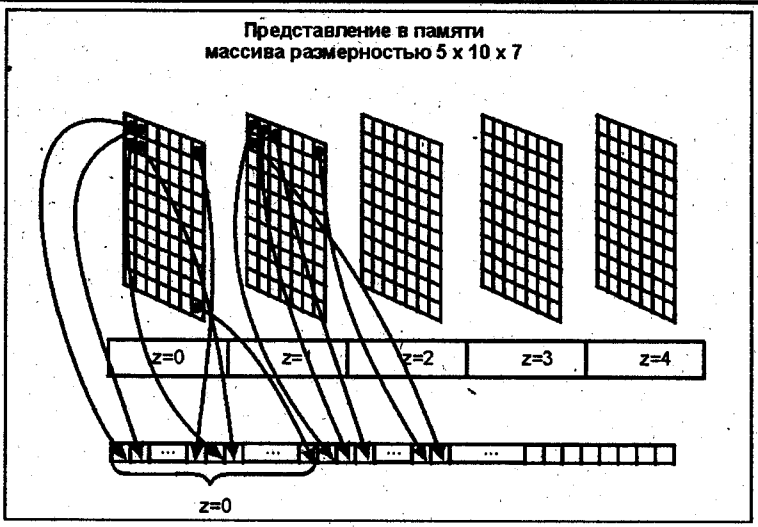


Рис. 7.3. Трехмерный массив в памяти

```
#include <iostream>
int main()
{
  char x[][3] = {{9,8,7},{6,5,4},{3,2,1}};
  for(int i=0; i<3; i++)
  {
    for(int j=0; j<3; j++)
      cout << (int)x[i][j] << ' ';
    cout << '\n';
  }
  return 0;
}
```

В начале работы программы в памяти резервируется место для девяти однобайтных элементов (тип char) с заполнением массива в порядке следования байт инициализации:

$x_{00} = 9$; $x_{01} = 8$; $x_{02} = 7$; $x_{10} = 6$; $x_{11} = 5$;
$x_{12} = 4$; $x_{20} = 3$; $x_{21} = 2$; $x_{22} = 1$.

Далее с помощью вложенного цикла осуществляется вывод значений массива: внешний цикл for перебирает строки исходной матрицы, в то время как внутренний цикл выводит значения массива по столбцам. В результате на экране будет отображено:

```
9 8 7
6 5 4
3 2 1
```

Доступ к элементам многомерного массива через указатели осуществляется немного сложнее. Поскольку, например, двумерный массив Matrix[x][y] может быть представлен как одномерный (Matrix[x]), каждый элемент которого также является одномерным массивом (Matrix[y]), указатель на двумерный массив pMtrx, ссылаясь на элемент массива Matrix[x][y], по сути указывает на массив Matrix[y] в массиве Matrix[x]. Таким образом, для доступа к содержимому ячейки указатель pMtrx придется разыменовывать дважды.

```
#include <iostream.h>
int main()
{
  char ArrayOfChar[3][2] = {'W','O','R','L','D','!'};
  char* pArr = (char*)ArrayOfChar;
  pArr += 3;
  char Letter = *pArr;
  cout << Letter;
  return 0;
}
```

В приведенном примере объявляется массив символов размерностью 3 x 2 и указатель pArr на него (фактически – указатель на ArrayOfChar[0][0]). В строке

```
char* pArr = (char*)ArrayOfChar;
```

идентификатор ArrayOfChar уже является указателем на элемент с индексом 0, однако, поскольку массив двумерный, требуется его повторное разыменование.

Увеличение pArr на 3 приводит к тому, что он указывает на элемент массива, значение которого – символ 'L' (элемент ArrayOfChar[1][1]). Далее осуществляется вывод содержимого ячейки массива, на которую указывает pArr.

Тема 7.4
Динамическое выделение массивов

В программе каждая переменная может размещаться в одном из трех мест: в области данных программы, в стеке или в свободной памяти (так называемая куча).

Каждой переменной в программе память может отводиться либо статически, то есть в момент загрузки программы, либо динамически – в процессе выполнения программы. Все рассматривае-

мые до сих пор переменные и массивы объявлялись статически и, следовательно, хранили значения всех своих элементов в стековой памяти или области данных программы. Если количество элементов массива невелико, такое размещение оправдано. Однако довольно часто возникают случаи, когда в стековой памяти, содержащей локальные переменные и вспомогательную информацию (например, точки возврата из вложенных функций), недостаточно места для размещения всех элементов большого массива. Ситуация еще более усугубляется, если массивов большого размера должно быть много. Здесь на помощь приходит возможность использования для хранения данных динамической памяти.

Тема 7.5
Функции malloc, calloc, free и операторы new и delete

Чтобы в дальнейшем можно было разместить в памяти некоторый динамический объект, для него необходимо предварительно выделить в памяти соответствующее место. По окончании работы с объектом выделенную для его хранения память требуется освободить.

Выделение динамической памяти под объект осуществляется при помощи следующих выражений:

```
malloc;
calloc;
new;
```

Освобождение выделенных ресурсов памяти производится выражениями:

```
free;
delete;
```

Функция malloc подключается в одном из заголовочных файлов stdlib.h или alloc.h и имеет синтаксис:

```
void *malloc(size_t size);
```

Данная функция призвана выделить в памяти блок размером size байт из кучи. В больших моделях памяти (Compact, Large, Huge) для кучи доступно все пространство памяти от вершины программного стека. В моделях Tiny, Small и Medium это пространство сокращено из-за наличия небольшой системной области, позволяющей стеку расти.

В случае успешного резервирования блока памяти функция malloc возвращает указатель на только что выделенный блок. При неудачном результате операции с памятью функция возвращает NULL – зарезервированный идентификатор со значением '\0' (абсолютное значение 0x00). При этом содержимое блока остается неизменным. Если в качестве аргумента функции указано значение 0, функция возвращает NULL.

Как видно из синтаксиса, данная функция возвращает указатель типа void, однако, поскольку на практике чаще всего приходится выделять память для объектов конкретного типа, необходимо привести тип полученного пустого указателя к требуемому. Например, выделение памяти под три объекта типа int и определение указателя на начало выделенного блока можно произвести следующим образом:

```
int *pInt = (int*)malloc(3*sizeof(int));
```

В отличие от malloc, функция calloc, кроме выделения области памяти под массив объектов, еще производит инициализацию элементов массива нулевыми значениями. Функция имеет следующий синтаксис:

```
void *calloc( size_t num, size_t size );
```

Аргумент num указывает, сколько элементов будет храниться в массиве, а параметр size сообщает размер каждого элемента в байтах. Приведенный здесь тип size_t (объявлен в заголовочном файле stddef.h) является синонимом типа unsigned char, возвращаемого оператором sizeof.

Функция освобождения памяти free в качестве единственного аргумента принимает указатель на удаляемый блок объектов (*block) и имеет синтаксис:

```
void free(void *block);
```

Данная функция подключается в заголовочном файле stdlib.h или alloc.h. Тип объектов удаляемого блока может быть произвольным, на что указывает ключевое слово void.

Рассмотрим пример выделения динамической памяти для десяти объектов типа int, заполнение выделенной области числами от нуля до девяти и освобождение ресурса памяти.

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

int main(void)
```

```
{
  int* pRegion;
  // Выделение области памяти
  // и проверка успешной работы функции
  if((pRegion =
    (int*)malloc(10*sizeof(int))) == NULL)
  {
    cout << "Недостаточно памяти\n";
  // Завершить программу, если не
  // хватает памяти для выделения
    return 1;
  }
  // Заполнение выделенного блока памяти
  for(int i=0; i<10; i++)
    *(pRegion+i) = i;
  // Вывод значений заполненного блока
  for(int i=0; i<10; i++)
  {
    cout << (*pRegion) << '\n';
    pRegion++;
  }
  // Освобождение блока памяти
  free(pRegion);
  return 0;
}
```

Пример динамического выделения памяти под двумерный массив Array[10][20] переменных типа int выглядит следующим образом:

```
Array = (int*)malloc(10*20*sizeof(int));
```

что эквивалентно

```
Array = (int*)malloc(200*sizeof(int));
```

Как видите, изменяется только количество отводимой под элементы массива памяти. В данном случае будет выделено 10 x 20 = 200 элементов типа int.

В отличие от функций работы с динамической памятью malloc, calloc и free, заимствованных в C++ из стандарта ANSI C для совместимости, новые операторы гибкого распределения памяти new и delete обладают дополнительными возможностями, перечисленными ниже.

Как отмечалось выше, функции malloc и calloc возвращают пустой указатель, который в дальнейшем требуется приводить к

заданному типу. Оператор new возвращает указатель на тип, для которого выделялась память, и дополнительных преобразований уже не требуется.

Операция new предполагает возможность использования совместно с библиотечной функцией set_new_handler, позволяющей пользователю определить свою собственную процедуру обработки ошибки при выделении памяти.

Операторы new и delete могут быть *перегружены* с тем, чтобы они, например, могли принимать дополнительные параметры или выполняли специфические действия с учетом конкретной ситуации работы с памятью. Подробнее перегрузка операторов описывается в разделе 15.

Операторы new и delete имеют две формы:

- управление динамическим размещением в памяти единичного объекта;
- динамическое размещение массива объектов.

Синтаксис при работе с единичными объектами следующий:

```
тип_объекта *имя = new тип_объекта;
delete имя;
```

При управлении жизненным циклом массива объектов синтаксис обоих операторов имеет вид:

```
тип_объекта *имя = new тип_объекта[число];
delete[] имя;
```

Здесь число в операторе new[] характеризует количество объектов типа тип_объекта, для которых производится выделение области памяти. В случае успешного резервирования памяти переменная-указатель имя ссылается на начало выделенной области. При удалении массива его размер указывать не нужно.

Форма оператора delete должна обязательно соответствовать форме оператора new для данного объекта: если выделение памяти проводилось для единичного объекта (new), освобождение памяти также должно осуществляться для единичного объекта (delete). В случае применения оператора new[] (для массива объектов) в конечном итоге высвобождение памяти должно быть произведено с использованием оператора delete[].

Поясним все вышесказанное на примере.

```
#include <iostream.h>
#include <new.h>
void newHandler()
```

```
{
  cout << "\nНедостаточно памяти!\n";
}
int main()
{
  int* ptr;
  set_new_handler(newHandler);
  ptr = new int[100];
  for(int i=0; i<100; i++)
    *(ptr+i) = 100-i;
  for(int i=0; i<100; i++)
    cout << *(ptr+i) << " ";
  delete[] ptr;
  long double* lptr;
  if(lptr = new long double[999999])
  {
    cout << "\nГотово!";
    delete[] lptr;
  }
  else
    cout << "\nНеудача...";
  return 0;
}
```

В начале представленной программы подключается уже знакомый заголовочный файл iostream.h для использования операций ввода/вывода, а также файл new.h для определения обработчика ошибок set_new_handler. Функция newHandler(), реализованная в начале программы, является собственно обработчиком ошибок выделения памяти. Она подключается в строке

```
set_new_handler(newHandler);
```

заменяя тем самым стандартную процедуру обработки. Теперь всякий раз, когда программе будет необходимо выделить некоторую область динамической памяти, но в силу аппаратных или программных ограничений осуществить это будет невозможно, на экране будет появляться надпись:

```
Недостаточно памяти!
```

В главной функции программы следует объявление целочисленного указателя ptr, которому тут же оператором new присваивается адрес выделенного блока динамической памяти для 100 значений типа int. Далее область памяти заполняется в цикле числами от 100 до 1, а в следующем цикле осуществляется

вывод ее содержимого, после чего выделенный блок динамической памяти освобождается (оператор delete[]). Следующий шаг – попытка выделения памяти для большого массива типа long double. Если результат операции new[] не нулевой, указателю lptr будет присвоено значение адреса зарезервированной памяти с выводом надписи «Готово!» и последующим высвобождением области. В случае неудачного выделения выводится сообщение «Неудача...».

Тема 7.6
Массивы в качестве параметров функций

В тело функций в качестве аргументов можно передавать значения, хранящиеся в массивах. При вызове функции параметр типа массива преобразовывается компилятором в указатель на тип массива. Например, если аргумент-массив имеет тип unsigned long, при вызове он будет преобразован в unsigned long*. Таким образом, изменение в функции значения любого элемента массива, являющегося аргументом, обязательно повлияет и на оригинал. Массивы отличаются от других типов тем, что их нельзя передавать по значению – внутрь тела функции попадает только адрес массива.

Синтаксис вызова функции при этом может быть следующим:

```
FunctionName(ArrayName);
```

Тогда прототип функции включает указание в качестве параметра типа передаваемого массива и следующих за ним прямоугольных скобок. Например:

```
void FunctionName(char[]);
```

Другой вариант синтаксиса передачи массива в функцию – когда прототип функции содержит символ операции взятия адреса после указания типа аргумента:

```
char FunctionName(char&);
```

При этом синтаксис вызова функции принимает следующий вид:

```
FunctionName(*ArrayName);
```

Ниже приводится пример, иллюстрирующий оба варианта передачи массива в качестве параметра функции.

```
#include <iostream.h>
// Прототипы функций:
```

```
void Out1(int[]);
void Out2(int&);
int main()
{
  int Array[]={10,8,6,4,2,0};
  // Вызов первой функции:
  Out1(Array);
  cout << '\n';
  // Вызов второй функции:
  Out2(*Array);
  cout << '\n';
  return 0;
}
// Реализация обеих функций:
void Out1(int arr[])
{
  for(int i=0; i<sizeof(arr); i++)
    cout << arr[i] << " ";
}
void Out2(int& arr)
{
  for(int i=0; i<sizeof(arr); i++)
    cout << *(&(arr)+i) << " ";
}
```

В рассмотренном примере объявляется массив Array[], содержащий шесть целочисленных значений, и осуществляется его передача в функции Out1() и Out(2). Обе функции выполняют одно и то же действие – выводят содержимое массива-аргумента на печать – и отличаются только интерфейсом.

Использование в качестве параметра функции многомерного массива затруднено, поэтому на практике чаще всего осуществляется передача массива указателей, что значительно упрощает синтаксис.

Практикум
«Массивы»

Упражнение 7.1
Операции с массивами

Проанализируйте предлагаемый ниже пример, иллюстрирующий применение массивов в теле программы на C++.

Пусть необходимо проиграть мелодию песенки «В лесу родилась елочка». Создадим вещественный массив, содержащий ноты Notes[]. Каждый элемент массива представляет собой частоту определенной ноты, а размерность массива в нашем случае определяется автоматически, поскольку он инициализируется при объявлении. В главной функции программы в цикле перебираются все элементы массива и передаются в качестве первого параметра функции Beep(), которая осуществляет вывод звука на системный динамик. Второй аргумент функции задает длительность звучания каждой ноты в миллисекундах.

```
#include <iostream.h>
#include <windows.h>
const float Notes[]={520, 880, 880, 780, 880, 700, 520, 520};
int main()
{
   for(int i=0; i<sizeof(Notes)/sizeof(Notes[0]); i++)
   {
      Beep(Notes[i], 200);
   }
   return 0;
}
```

Упражнение 7.2
Работа с многомерными массивами

Рассмотрим работу с многомерными массивами на примере транспонирования квадратной матрицы.

Целочисленная матрица Arr заполняется в цикле последовательностью чисел от 0 до 5. Затем, опять-таки в цикле, матрица TrArr заполняется элементами из массива Arr таким образом, что TrArr представляет собой транспонированный массив Arr. После этого результирующая матрица выводится на консоль.

```
#include <iostream.h>
const int COL = 5;
const int ROW = COL;
int main()
{
   int Arr[COL][ROW];
   int TrArr[ROW][COL];
   int idx = 1;
   for(int col=0; col<COL; col++)
      for(int row=0; row<ROW; row++)
```

```
        {
            Arr[col][row] = idx++;
        }
    for(int col=0; col<COL; col++)
        for(int row=0; row<ROW; row++)
        {
            TrArr[row][col] = Arr[col][row];
        }
    for(int col=0; col<COL; col++)
    {
        for(int row=0; row<ROW; row++)
        {
            cout << TrArr[col][row] << '\t';
        }
        cout << '\n';
    }
    return 0;
}
```

Модифицируйте данный пример таким образом, чтобы элементы массива использовались не посредством индексов, а с помощью указателей.

Упражнение 7.3
Динамические массивы

Проанализируйте предлагаемый ниже пример, в котором создается два целочисленных указателя: Array и p. Затем динамически выделяется область памяти под 10 элементов. В цикле массив заполняется последовательными значениями, которые в дальнейшем выводятся на консоль.

```
#include <iostream.h>
int main()
{
    int *Array;
    int *p;
    Array = new int[10];
    p = Array;
    for(int i=0; i<10; i++)
        *p++ = i;
    p = Array;
    for(int i=0; i<10; i++)
        cout << *p++ << endl;
    delete[] Array;
}
```

Упражнение 7.4
Массивы и функции

В данном упражнении Вам предлагается составить текст программы, где в некоторую функцию Func() передается адрес целочисленного массива из 10 элементов. Произведите в функции Func() суммирование элементов массива и верните результат в главную функцию. Проверьте полученный результат путем вывода на консоль.

Раздел 8
Строки и операции с ними

Тема 8.1
Массивы символов в C++

В стандарт C++ включена поддержка нескольких наборов символов. Традиционный 8-битовый набор символов называется «узкими» символами. Кроме того, включена поддержка 16-битовых символов, которые называются «широкими». Для каждого из этих наборов символов в библиотеке имеется своя совокупность функций. Подробнее об этом будет рассказано в Приложении.

Как и в ANSI C, для представления символьных строк в C++ не существует специального строкового типа. Вместо этого строки в C++ представляются как массивы элементов типа char, заканчивающиеся *терминатором строки* – символом с нулевым значением (\0). Строки, заканчивающиеся нуль-терминатором, часто называют ASCIIZ-строками. Символьные строки состоят из набора символьных констант, заключенного в двойные кавычки:

```
"Это строка символов..."
```

В табл. 8.1 приведен набор констант, применяющихся в C++ в качестве символов.

Таблица 8.1
Классификация символов

Название	Символы
прописная буква	От 'A' до 'Z', от 'А' до 'Я'
строчная буква	От 'a' до 'z', от 'а' до 'я'
цифра	От '0' до '9'

Название	Символы
пустое место	Горизонтальная табуляция '\9', перевод строки (код ASCII 10), вертикальная табуляция (код ASCII 11), перевод формы (код ASCII 12), возврат каретки (код ASCII 13)
символы пунктуации	!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
управляющий символ	Все символы с кодами от 0 до 1Fh включительно и символ с кодом 7Fh
пробел	Символ пробела (код ASCII 32)
шестнадцатеричная цифра	От '0' до '9', от 'A' до 'F', от 'a' до 'f'

При объявлении строкового массива необходимо принимать во внимание наличие терминатора в конце строки, отводя тем самым под строку на один байт больше:

```
// Объявление строки размером 10 символов,
// включая терминатор. Реальный размер
// строки: 9 символов + 1 терминатор
char buffer[10];
```

Строковый массив может при объявлении инициализироваться начальным значением. При этом компилятор автоматически вычисляет размер будущей строки и добавляет в конец нуль-терминатор:

```
// Объявление и инициализация строки:
char Wednesday[] = "Среда";

// что равносильно:
char Wednesday[] = {'С','р','е','д','а','\0'};
```

В качестве оператора ввода при работе со строками вместо оператора записи в поток >> лучше использовать функцию getline(), так как потоковый оператор ввода игнорирует вводимые пробелы, а кроме того, может продолжить ввод элементов за пределами массива, если под строку отводится меньше места, чем вводится символов. Функция getline() принимает два параметра: первый аргумент указывает на строку, в которую осуществляется ввод, а второй – число символов, подлежащих вводу.

Рассмотрим пример объявления символьных строк и использования функции ввода getline().

```
#include <iostream.h>
int main()
{
  char myString[4];
  cout << "Введите строку: " << '\n';
  cin.getline(myString, 4);
  cout << "Вы ввели: " << myString;
  return 0;
}
```

Объявленная в начале программы строка myString может принять только три значащих символа и будет завершена нультерминатором. Все последующие символы, вводимые в этот строковый массив, будут отброшены.

Как видно из примера, при использовании функции getline() вторым параметром следует указывать число, меньше или равное размеру вводимой символьной строки.

Большинство функций работы со строками содержится в библиотеке string.h. Основные функции работы с символьными массивами сведены в табл. 8.2.

Таблица 8.2

Функции работы со строками символов

Наименование	Краткое описание
strcpy	Копирует строку2 в строку1
strcat	Присоединяет строку2 в конец строки1
strchr	Возвращает позицию первого вхождения символа в строку
strcmp	Сравнивает строку1 со строкой2, различая прописные и строчные буквы
strcmpi	См. stricmp
strcpy	Копирует строку2 в строку1
strcspn	Возвращает позицию первого вхождения символа из заданного набора символов
strdup	Распределяет память и делает копию строки

Наименование	Краткое описание
strerror	Возвращает указатель на строку текста сообщения об ошибке по заданному номеру системной ошибки
_strerror	Возвращает указатель на строку, образованную объединением произвольной строки и сообщения об ошибке в библиотечной функции
stricmp	Сравнивает строку1 со строкой2, не различая прописные и строчные буквы
strlen	Возвращает длину строки в байтах, не учитывая нулевой терминатор
strlwr	Преобразует все символы строки в строчные буквы
strncat	Присоединяет заданное число символов строки2 в конец строки1
strncmp	Сравнивает заданное число символов двух строк, различая прописные и строчные буквы
strncmpi	См. strnicmp
strncpy	Копирует заданное число символов строки2 в строку1
strnicmp	Сравнивает заданное число символов двух строк, не различая прописные и строчные буквы
strnset	Помещает заданный символ в заданное число позиций строки
strpbrk	Отыскивает место первого вхождения любого символа из заданного набора
strrchr	Отыскивает последнее вхождение символа в строке
strrev	Реверс строки
strset	Помещает символ во все позиции строки

Наименование	Краткое описание
strspn	Возвращает позицию в строке первого символа, который не принадлежит заданному набору символов
strstr	Отыскивает место первого вхождения строки2 в строку1
strtok	Возвращает указатель на лексему, ограниченную заданным разделителем
strupr	Преобразует все буквы строки в прописные
isalnum(c)	Истина, если символ с является буквой или цифрой
isalpha(c)	Истина, если символ с является буквой
isascii(c)	Истина, если код символа с <= 127
iscntrl(c)	Истина, если с – управляющий символ
isdigit(c)	Истина, если с – символ десятичной цифры
isgraph(c)	Истина, если с – печатаемый символ (код от 33 до 126)
islower(c)	Истина, если с – строчная буква
isprint(c)	Истина, если с – печатаемый символ (код от 33 до 126) или пробел
ispunct(c)	Истина, если с – символ пунктуации
isspace(c)	Истина, если с – символ пустого места или пробела
isupper(c)	Истина, если с – прописная буква
isxdigit(c)	Истина, если с – символ шестнадцатеричной цифры
toascii(c)	Возвращает код с или 128, если код с больше 127

Наименование	Краткое описание
_tolower(c)	Преобразует символ прописной буквы в символ строчной. Используется, если точно известно, что с – прописная буква. Возвращает код строчной буквы
tolower(c)	Преобразует символ прописной буквы в символ строчной, не изменяя все остальные символы. В отличие от _tolower() сначала проверяет, является ли с прописной буквой. Не являющиеся прописными буквами символы не преобразуются. Возвращает код строчной буквы
_toupper(c)	Преобразует символ строчной буквы в символ прописной. Используется, если точно известно, что с – строчная буква. Возвращает код прописной буквы
toupper(c)	Преобразует символ строчной буквы в символ прописной, не изменяя все остальные символы. В отличие от _toupper() сначала проверяет, является ли с строчной буквой. Не являющиеся строчными буквами символы не преобразуются. Возвращает код прописной буквы

В следующих пунктах данного раздела самые популярные из приведенных функций будут рассмотрены подробно, однако следует учесть, что в некоторых версиях поставки библиотек C++ данные функции могут осуществлять неправильно (или не выполнять вовсе) работу с национальными символами.

Тема 8.2
Определение длины строк

Очень часто при работе со строками необходимо знать, сколько символов содержит строка. Для выяснения информации о длине строки в заголовочном файле string.h описана функция strlen(). Синтаксис этой функции имеет вид:

```
size_t strlen(const char* string)
```

Данная функция в качестве единственного параметра принимает указатель на начало строки string, вычисляет количество символов строки и возвращает полученное беззнаковое целое число (size_t). Функция sizeof() возвращает значение на единицу меньше, чем отводится под массив по причине резервирования места для символа '\0'. Следующий фрагмент демонстрирует использование функции strlen():

```
char Str[] = "ABCDEFGHIJK";
unsigned int i;
i = strlen(Str);
```

Часто функция sizeof() используется при вводе строк в качестве второго параметра конструкции cin.getline(), что делает код более универсальным, так как не требуется явного указания числа вводимых символов. Если теперь потребуется изменить размер символьного массива, достаточно модифицировать лишь одно число при объявлении строки символов:

```
// Было:
// char myString[4];
// cin.getline(myString, 4);
// Стало:
char myString[20];
cin.getline(myString, sizeof(myString));
```

Тема 8.3
Копирование и конкатенация строк

Значения строк могут копироваться из одной в другую. Для этой цели используют ряд стандартных функций, описываемых ниже.

Функция strcpy() имеет прототип:

```
char* strcpy(char* str1, const char* str2)
```

и выполняет побайтное копирование символов из строки, на которую указывает str2, в строку по указателю str1. Копирование прекращается только в случае достижения нуль-терминатора строки str2, поэтому перед копированием необходимо удостовериться, что длина str2 меньше или равна длине str1. В противном случае возможно возникновение ошибок, связанных с наложением данных.

Например, следующий фрагмент копирует в строку Str значение строки "Проверка копирования":

```
char Str[20];
strcpy(Str, "Проверка копирования";
```

Использование указателей в сочетании с символьными массивами предоставляет программисту возможность производить копирование не всей строки, а лишь отдельного ее фрагмента. При этом второй параметр является указателем на некоторый элемент строкового массива. Например, следующий фрагмент скопирует в str2 окончание строки str1:

```
char str1[20] = "Проверка копирования";
char str2[20];
char* ptr = str1;
ptr += 9;
// ptr теперь указывает на подстроку "копирования" в строке str1
strcpy(str2, ptr);
cout << str2 << '\n';
```

Функция strncpy() отличается от strcpy() тем, что в ее параметрах добавляется еще один аргумент, указывающий количество символов, не больше которого будет скопировано. Ее синтаксис имеет вид:

```
char* strncpy(char* str1,const char* str2, size_t num)
```

Если длина str1 меньше длины str2, происходит урезание символов:

```
char cLong[] = "01234567890123456789012345678 9";
char cShort[]="abcdef";
strncpy(cShort, cLong, 4);
cout << cShort << '\n';
cout << cLong << '\n';
```

В результате будет выведено:

```
0123ef
01234567890123456789012345678 9
```

То есть из строки cLong в строку cShort скопировано четыре первых символа, затерев тем самым исходное значение начала короткой строки.

Функция strdup() в качестве параметра получает указатель на строку-источник, осуществляет распределение памяти, копирует в отведенную область строку и возвращает указатель на начало полученной строки-копии. Синтаксис функции следующий:

```
char* strdup(const char* source)
```

В следующем примере производится копирование строки str1 в строку str2:

```
char* str1 = "Процедура не найдена";
char* str2;
str2 = strdup(str1);
```

Конкатенация (или присоединение) строк довольно часто используется для образования новой строки символов. Для этой операции стандартная библиотека предлагает функции strcat() и strncat().

Функция strcat() имеет синтаксис:

```
char* strcat(char* str1, const char* str2)
```

В результате работы функции содержимое строки, на которую указывает str2, присоединяется к содержимому строки, на которую ссылается str1. Возвращаемый функцией указатель str1 указывает на результирующую строку. При этом величина строкового массива str1 должна быть достаточной для хранения объединенной строки.

В следующем примере строка str инициализируется с помощью функции копирования strcpy() и дополняется подстрокой, используя функцию strcat():

```
char str[81];
strcpy(str, "Для продолжения ");
strcat(str, "нажмите клавишу");
```

Функция strncat() также осуществляет конкатенацию строк, однако присоединяет лишь указанное в третьем параметре количество символов (беззнаковое целое):

```
char* strncat(char* str1, const char* str2, size_t num)
```

Функция возвращает указатель на начало сформированной строки str1. Следующий пример производит конкатенацию строки str1 с семью первыми символами подстроки str2:

```
char str1[90]="Для продолжения ";
char str2[30]="нажмите клавишу";
strncat(str1,str2,7);
cout << str1;
```

В результате будет выведена строка:

"Для продолжения нажмите".

Тема 8.4
Сравнение строк

Библиотека функций string.h предлагает к использованию готовые функции, выполняющие сравнение строк. Ниже приводятся функции, выполняющие посимвольное сравнение двух строк.

Функция strcmp() имеет синтаксис:

```
int strcmp(const char* str1, const char* str2)
```

После сравнения строк str1 и str2 данная функция возвращает в результате одно из следующих значений:

<0 – если строка str1 меньше, чем str2;
=0 – если строки эквивалентны;
>0 – если строка str1 больше, чем str2.

Эта функция производит сравнение, различая прописные и строчные буквы. Следующий пример иллюстрирует работу функции strcmp():

```
char str1[]="Ошибка открытия базы";
char str2[]="Ошибка открытия Базы";
int i;
i = strcmp(str1, str2);
```

В результате переменной i будет присвоено положительное значение, так как строка из str1 больше, чем строка из str2, по той причине, что прописные буквы имеют код символов меньше, чем те же символы в нижнем регистре (слово «базы» в первом случае начинается со строчной литеры, а во втором – с прописной).

Функция stricmp() имеет синтаксис:

```
int stricmp(const char* str1, const char* str2)
```

Данная функция сравнивает строки str1 и str2, не различая регистра символов. Возвращается одно из следующих целочисленных значений:

<0 – если строка str1 меньше, чем str2;
=0 – если строки эквивалентны;
>0 – если строка str1 больше, чем str2.

Следующий фрагмент программы демонстрирует применение функции stricmp():

```
char str1[]="Moon";
char str2[]="MOON";
int i;
i = stricmp(str1, str2);
```

В данном случае переменной i будет присвоено значение 0 (сигнализируя тем самым совпадение строк), так как str1 и str2 отличаются только регистром.

Функция strncmp() проводит сравнение определенного числа первых символов двух строк. Регистр символов при этом учитывается. Функция имеет следующий прототип:

```
int strncmp(const char* str1, const char* str2, size_t num)
```

Данная функция сравнивает num первых символов двух строк, на которые указывают str1 и str2, и возвращает одно из следующих значений:

<0 – если строка str1 меньше, чем str2;
=0 – если строки эквивалентны;
>0 – если строка str1 больше, чем str2.

Рассмотрим пример использования функции strncmp().

```
char str1[]="Ошибка открытия базы";
char str2[]="Ошибка Открытия базы";
int i;
i = strncmp(str1, str2, 12);
```

В результате сравнения первых 12-ти символов обеих строк переменная i получит положительное значение, так как подстроки "Ошибка откры" и "Ошибка Откры" отличаются одним символом и в первом случае код символа больше, чем во втором.

Функция strnicmp() производит сравнение определенного числа первых символов двух строк, не обращая внимания на регистр символов. Данная функция описана следующим образом:

```
int strnicmp(const char* str1, const char* str2, size_t num)
```

Функция возвращает целочисленное значение согласно следующему правилу:

<0 – если строка str1 меньше, чем str2;
=0 – если строки эквивалентны;
>0 – если строка str1 больше, чем str2.

В следующем примере производится сравнение заданного числа символов подстрок:

```
char str1[]="Opening error";
char str2[]="Opening Error...";
int i;
i = strnicmp(str1, str2, 13);
```

В результате переменной i будет присвоено значение 0, так как первые 13 символов обеих подстрок отличаются только регистром.

Тема 8.5
Преобразование строк

Элементы символьных строк могут быть преобразованы из одного регистра в другой. Для этого используются стандартные функции _strlwr() и _strupr(). Следует отметить, что в некоторых

версиях компиляторов имена данных функций могут следовать без ведущего символа подчеркивания.

Функция _strlwr() принимает в качестве параметра указатель на строку символов, преобразует эту строку к нижнему регистру (строчные символы) и возвращает указатель на полученную строку. Данная функция имеет следующий прототип:

```
char* _strlwr(char* str)
```

Следующий фрагмент показывает применение функции _strlwr():

```
char str[] = "ABRACADABRA";
_strlwr(str);
```

После вызова функции строка str будет содержать значение "abracadabra".

Функция _strupr() объявлена следующим образом:

```
char* _strupr(char* str)
```

Данная функция преобразует строку символов, на которую указывает str, в прописные буквы (к верхнему регистру). В результате работы функции возвращается указатель на полученную строку.

Следующий пример переводит символы заданной строки в верхний регистр:

```
char str[]="pacific ocean";
_strupr(str);
cout << str;
```

В результате будет выведено:

```
PACIFIC OCEAN
```

Приведенные выше функции преобразования строк, работая с указателями, преобразуют исходную строку, которая не всегда может быть восстановлена, поэтому, если в дальнейшем коде программы потребуется воспользоваться оригиналом символьной строки, перед использованием функций _strlwr() и _strupr() необходимо сделать копию их аргументов.

На практике довольно широко используются функции проверки принадлежности символов какому-либо диапазону, такие как isalnum(), isalpha(), isascii(), isdigit() и т.д. (см. табл. 8.2), объявленные в заголовочном файле ctype.h. Ниже рассматривается пример использования этого вида функций.

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

```
int main(void)
{
  char str[4];
  do
  {
    cout << "Enter your age, please...";
    cin.getline(str, 4);
    if(isalpha(str[0]))
    {
      cout << "You entered a letter,";
      cout << " try again\n";
      continue;
    }
    if(iscntrl(str[0]))
    {
      cout << "You entered a control";
      cout << " symbol, try again\n";
      continue;
    }
    if(ispunct(str[0]))
    {
      cout << "You entered a ";
      cout << "punctuation, try again\n";
      continue;
    }
    for(int i=0; i<strlen(str); i++)
    {
      if(!isdigit(str[i]))
        continue;
      else
      {
        cout << "Your age is " << str;
        return 0;
      }
    }
  }
  while(true);
}
```

Здесь пользователю предлагается ввести свой возраст. Функция cin.getline() помещает в строку str введенную последовательность (до трех) символов, после чего следует проверка первого введенного элемента массива на принадлежность к буквенной, управляющей последовательностей или символам пунктуации. Если результат соответствующей проверки положительный, пользователю предлагается ввести данные повторно. В против-

ном случае все введенные элементы строки проверяются на принадлежность к цифровому набору данных. Если хотя бы один из символов не удовлетворяет проверяемому условию, цикл ввода повторяется сначала. После корректного ввода данных на экран выводится сообщение о возрасте пользователя.

Тема 8.6
Обращение строк

Функция обращения строки strrev() меняет порядок следования символов на обратный (реверс строки). Данная функция имеет прототип:

```
char* strrev(char* str)
```

Следующий пример демонстрирует работу функции strrev().

```
char str[]="Привет";
cout << strrev(str);
```

В результате на экране будет выведена строка "тевирП". Эта функция также преобразует строку-оригинал.

Тема 8.7
Поиск символов

Одна из часто встречаемых задач при работе со строками – поиск отдельного символа или даже группы символов. Библиотека string.h предлагает следующий набор стандартных функций.

Функция нахождения символа в строке strchr() имеет следующий прототип:

```
char* strchr(const char* string, int c)
```

Эта функция производит поиск символа c в строке string и в случае успешного поиска возвращает указатель на место первого вхождения символа в строку. Если указанный символ не найден, функция возвращает NULL. Поиск символа осуществляется с начала строки.

Ниже рассматривается фрагмент, осуществляющий поиск заданного символа в строке.

```
char str[] = "абвгдеёжзийк";
char* pStr;
```

```
pStr = strchr(str, 'ж');
```

В результате работы программы указатель pStr будет указывать на подстроку "жзийк" в строке str.

Функция strrchr() осуществляет поиск заданного символа с конца строки. Она имеет следующий синтаксис:

```
char* strrchr(const char* string, int c)
```

Данная функция возвращает указатель на последний, совпавший с заданным c, символ в строке string. Если символ не найден, возвращается значение NULL.

В следующем примере производится поиск заданного символа с конца строки:

```
char str[] = "абвгдеёжзийк";
char* pStr;
pStr = strrchr(str, 'и');
```

Таким образом, указатель pStr будет указывать на подстроку "ийк" в строке str.

Функция strspn() проводит сравнение символов одной строки с символами другой и возвращает позицию (начиная с нуля), в которой строки перестают совпадать. Данная функция имеет следующий прототип:

```
size_t strspn(const char* string, const char* group)
```

Функция проверяет каждый символ строки string на соответствие каждому из символов строки group. В результате работы функции возвращается число совпавших символов.

Следующий пример демонстрирует использование данной функции:

```
char str[] = "Загрузка параметров БД";
char substr[] = "Загрузка параметрррррр";
int index=0;
index = strspn(str, substr);
cout << index;
```

На экран будет выведено число 17, так как символы строки str и подстроки substr совпадают вплоть до 17-й позиции.

Приведенная функция различает регистр символов.

Функция strcspn() имеет синтаксис:

```
size_t strcpn(const char* str1, const char* str2)
```

Эта функция сопоставляет символы строки str1 и str2 и возвращает длину строки str1, не входящей в str2. Таким образом, можно определить, в какой позиции происходит пересечение двух символьных массивов:

```
char str[] = "abcdefghijk";
int index;
index = strcspn(str, "elf");
```

Переменная index получит значение 4, так как в этой позиции строки имеют первый общий элемент.

Функция strpbrk() объявлена следующим образом:

```
char* strpbrk(const char* str1, const char* str2)
```

Эта функция отыскивает место вхождения в строку str1 любого из символов строки str2. Если символы найдены, возвращается место первого вхождения любого символа из str2 в строку str1. В противном случае функция возвращает NULL.

Ниже иллюстрируется использование функции strpbrk():

```
char str1[]="abcdefghijk";
char str2[] = "ecb";
char* ptr;
ptr=strpbrk(str1, str2);
cout << ptr << '\n';
```

В результате будет выведена подстрока "bcdefghijk", так как символ 'b' из строки str2 встречается в строке str1 раньше других.

Тема 8.8
Поиск подстрок

При необходимости поиска в одной строке последовательности символов, заданной в другом символьном массиве (подстроке, лексеме), стандартная библиотека string.h предлагает воспользоваться одной из следующих функций.

Функция strstr() описана следующим образом:

```
char* strstr(const char* str, const char* substr)
```

Данная функция осуществляет сканирование строки str и находит место первого вхождения подстроки substr в строку str. В случае успешного поиска функция strstr возвращает указатель на первый символ строки str, начиная с которого следует точное совпадение части str обязательно со всей лексемой substr. Если подстрока substr не найдена в str, возвращается NULL.

Следующий пример показывает использование функции strstr().

```
char str1[]="Производится поиск элемента";
char str2[] = "поиск";
char* ptr;
ptr=strstr(str1, str2);
cout << ptr << '\n';
```

На экран будет выведено "поиск элемента", так как подстрока, содержащаяся в str2, находится внутри строки str1 и функция strstr() установит указатель ptr на соответствующий элемент символьного массива str1.

Чтобы найти место последнего вхождения подстроки в строку, можно воспользоваться следующим приемом: *обе строки* реверсируются с помощью функции strrev(), а затем полученный результат анализируется в функции strstr().

Функция strtok() имеет синтаксис:

```
char* strtok(char* str, const char* delim)
```

Эта функция выполняет поиск в строке str подстроки, обрамленной с обеих сторон любым символом-разделителем из строки delim. В случае успешного поиска данная функция обрезает строку str, помещая символ '\0' в месте, где заканчивается найденная лексема. Таким образом, при повторном поиске лексемы в указанной строке str первым параметром следует указывать NULL. Так как strtok() модифицирует строку-оригинал, рекомендуется предварительно сохранять копию последней. Приведенный ниже пример иллюстрирует вышесказанное.

Предположим, необходимо разбить имеющееся в строковом массиве предложение по словам и вывести каждое из них на экран.

```
#include <string.h>
#include <iostream.h>
int main()
{
  char str[]="Язык программирования C++";
  char *Delimiters = " .!?,;:\"'/0123456789
     @#$%^&*()<>{}[]`~+-=";
  char *ptr;
  ptr = strtok(str, Delimiters);
  if(ptr)
    cout << ptr << '\n';
  while(ptr)
  {
    ptr = strtok(NULL, Delimiters);
```

```
    if(ptr)
      cout << ptr << '\n';
  }
  return 0;
}
```

В данной программе объявляется подлежащая анализу строка str, подстрока, содержащая набор разделителей Delimiters и указатель на символьный тип данных ptr. Вызов функции strtok(str, Delimiters) сканирует строку str, и как только в ней встретится *любой* символ, входящий в подстроку Delimiters (в данном случае это символ пробела), указатель ptr станет ссылаться на начало исходной строки до найденного символа. То есть ptr будет содержать:

*ptr = "Язык".

Благодаря тому что функция strtok() помещает в найденном месте нуль-терминатор ('\0'), исходная строка модифицируется. Таким образом, массив символов str примет значение:

"программирования C++"

После проверки указателя ptr на существование в операторе if(ptr) найденное слово выводится на экран. Далее в цикле с помощью функции strtok() находится последний нуль-терминатор строки str:

ptr = strtok(NULL, Delimiters);

что фактически соответствует локализации следующего слова предложения, и найденная последовательность символов выводится на экран.

Тема 8.9
Функции преобразования типа

Функции преобразования данных довольно часто используются, как следует из названия, для преобразования одного типа данных в другой тип. В приведенной ниже табл. 8.3 перечислены основные функции, их прототипы подключаются в заголовочном файле stdlib.h.

Чаще всего данные функции используются для преобразования чисел, введенных в виде символьных строк, в числовое представление, а также для выполнения определенных арифметиче-

ских операций над ними и обратное преобразование в строку символов. Рассмотрим самые широко используемые из них.

Таблица 8.3
Преобразование данных

Наименование	Краткое описание
atof	Преобразует строку символов в число с плавающей точкой
atoi	Преобразует строку символов в строку типа int
atol	Преобразует строку символов в число типа long
ecvt	Преобразует число с плавающей точкой типа double в строку символов; десятичная точка и знак числа не включаются в полученную строку; позиция точки и знак числа возвращаются отдельно
fcvt	Идентично ecvt, но округляет полученное значение до заданного числа цифр
gcvt	Преобразует число с плавающей точкой типа double в строку символов, включая символ десятичной точки и используя специфицированное число цифр
itoa	Преобразует число типа int в строку символов
ltoa	Преобразует число типа long в строку символов
strtod	Преобразует строку символов в число с плавающей точкой типа double
strtol	Преобразует строку символов в число типа long
strtoul	Преобразует строку символов в число типа unsigned long
ultoa	Преобразует число типа unsigned long в строку символов

Функция atoi(), синтаксис которой

```
int atoi(const char* ptr)
```

преобразует ASCIIZ-строку символов, на которую указывает ptr, в число типа int. Если в строке встречается символ, который не может быть преобразован, данная функция возвращает 0. В случае если преобразуемое число превышает диапазон представления типа int, возвращается только два младших байта числа.

В отличие от нее, функция atol() преобразует заданное строковое число в тип long. Эта функция имеет аналогичный синтаксис:

```
int atoi(const char* ptr)
```

Если преобразуемое число превышает диапазон значений типа long, функция возвратит непредсказуемое значение.

Рассмотрим пример преобразования строки цифровых символов в целое и длинное целое.

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

int main()
{
  char str[]="70000";
  int i = atoi(str);
  long l = atol(str);
  cout << i << '\n';
  cout << l;

  return 0;
}
```

Если используется модель памяти, в которой тип int представляется двумя байтами, результат работы приведенной программы будет выглядеть следующим образом:

```
4464
70000
```

Дело в том, что число 70000 в шестнадцатеричной системе счисления представляется как 0x11170, но, поскольку функция atoi() при переполнении результата возвращает только два младших байта, переменная i примет шестнадцатеричное значение 0x1170, что эквивалентно десятичному 4464. Так как atol() оперирует с четырехбайтными числами, переполнения не произойдет и переменной l присвоится значение 70000.

Функция atof(), определенная как

```
double atof(const char* ptr)
```

выполняет преобразование ASCIIZ-строки в число с плавающей точкой типа double. Строка символов должна быть представлена с учетом формата:

[пробелы][знак][цифры][.][цифры][e|E[знак]цифры],

где

[пробелы] – последовательность пробелов или табуляторов;
[знак] – символ '+' или '–';
[цифры] – десятичные цифры;
[e|E] – символ показателя степени.

Преобразование символов прекращается, как только найден первый неконвертируемый символ или достигнут конец строки.

Функций обратного преобразования itoa() и ltoa() производят конвертирование чисел типа int и long соответственно. Они имеют следующий синтаксис:

```
char *_ltoa(long num, char* str, int radix);
```

и

```
char* itoa(int num, char* str, int radix);
```

или

```
char* _itoa(int num, char *str, int radix);
```

Данные функции принимают в качестве аргумента число num и преобразуют его в строку str с учетом основания системы счисления, представленной в переменной radix. Следующий фрагмент программы преобразует целое число 98765 в строку, используя десятичную систему счисления:

```
int numb = 98765;
char str[10];
itoa(numb, str, 10);
cout << numb << '\n' << str;
```

Функция strtod() преобразует строку символов в число с плавающей точкой. Ее синтаксис имеет следующий вид:

```
double strtod(const char *s, char **endptr);
```

Эта функция, так же как и функция atof(), преобразует строку, на которую указывает s, в число типа double, с тем лишь отличием, что в случае прекращения конвертирования строки возвращает указатель на первый непреобразуемый символ. Это позволяет организовать в дальнейшем обработку оставшейся части строки.

Функция gcvt() имеет прототип:

```
char* gcvt(double val, int ndec, char *buf);
```

и осуществляет конвертирование числа val типа double в ASCIIZ-строку, помещая ее в буфер buf. Если число цифр, подлежащих преобразованию, меньше целого числа, указанного в ndec, в преобразованном числе указываются символы знака и десятичной точки, при этом младшие разряды дробной части отбрасываются. В противном случае число преобразуется в экспоненциальную форму. Функция возвращает указатель на начало сформированной строки.

Следующий ниже пример демонстрирует использование функции gcvt() для преобразования чисел, имеющих различное представление в массивы цифровых символов.

```
#include <stdlib.h>
#include <iostream.h>
int main(void)
{
  char str[10];
  double num;
  int sig = 4;   // Значащих цифр
  num = 3.547;   // Обычное представление числа
  gcvt(num, sig, str);
  cout << str << '\n';
  num = -843.7105;   // Отрицательное число
  gcvt(num, sig, str);
  cout << str << '\n';
  num = 0.135e4;   // Экспоненциальное представление
  gcvt(num, sig, str);
  cout << str << '\n';
  return 0;
}
```

В результате будет выведено:

```
3.547
-843.7
1350
```

Практикум
«Строки и операции с ними»

Упражнение 8.1
Работа с символьными массивами

Напишите текст программы, объявляющей 12 строковых массивов. Заполните их названиями месяцев в году с помощью функции cin.getline(). Выведите на консоль по очереди эти массивы.

Упражнение 8.2
Копирование и конкатенация строк

Проанализируйте следующий пример.

Объявляются два динамических строковых массива st1 и st2, каждый размерностью в 30 элементов. Функцией strcpy() заполняются оба массива, а затем к содержимому первого массива функцией strcat() добавляется содержимое второго.

```
#include <iostream.h>
int main()
{
   char *st1 = new char[30];
   char *st2 = new char[30];
   strcpy(st1, "alpha");
   strcpy(st2, "-beta");
   cout << st1 << endl;
   cout << st2 << endl;
   strcat(st1, st2);
   cout << st1 << endl;
   delete st1; delete st2;
   return 0;
}
```

Упражнение 8.3
Преобразование строк

В предлагаемой лабораторной работе Вам необходимо составить программу, которая осуществляет консольный ввод строки символов, затем преобразует введенную строку в верхний регистр и выводит на консоль обращенный массив символов.

Упражнение 8.4
Поиск в символьных массивах

Проанализируйте предлагаемый ниже листинг программы, подсчитывающей в строковом массиве количество встречающихся литер «е». Для этого будем использовать стандартную функцию поиска литеры в строке strchr().

Изначально вспомогательный символьный указатель ptr ссылается на начало строки для поиска Arr. Далее в цикле do ... while производится поиск заданной буквы (вызов функции strchr(ptr, Find)) с присвоением указателю ptr значения позиции найденной литеры.

Если буква найдена в исходной строке, значение счетчика counter инкрементируется и указатель смещается на следующий за найденным символ. В заключение значение счетчика counter выводится на консоль.

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

int main()
{
   char Arr[] = "Navigation functions can be controlled" \
                "through the keyboard as well as the " \
                "mouse. You can change focus areas, and " \
                "manipulate menus and dialog boxes.";
   char *ptr = Arr;
   char Find = 'e';
   int counter = 0;

   do{
      if(ptr = strchr(ptr, Find)){
         counter++;
         ptr++;
      }
   }while(ptr);
   cout << counter;
   return 0;
}
```

Раздел 9
Структуры и объединения

Тема 9.1
Структуры и операции с ними

Как правило, объекты, представляемые в программе, обладают рядом разнообразных и, что очень важно, разнотипных свойств. Конструирование таких объектов на C++ предусматривает использование *структур* и *объединений*.

Структура может быть представлена как некоторый набор разнотипных и/или однотипных данных, совокупность которых рассматривается как совершенно новый, *пользовательский* тип данных. Структура объявляется с помощью ключевого слова struct, за которым следует необязательное *имя тега* для создания нового типа данных и указываемый в фигурных скобках шаблон, по которому будут создаваться переменные структурного типа. Шаблон содержит указываемые через точку с запятой объявления *полей* или *членов* структуры. Объявление поля состоит из указания типа и имени переменной:

```
struct NewType
{
  type1 Name1;
  type2 Name2;
  ...
  typeN NameN;
};
```

Синтаксис описания структуры заканчивается символом точка с запятой (;).

Использование структурированных данных в теле программы возможно в том случае, если будет объявлен какой-нибудь объект вновь созданного типа. Например, для приведенного выше синтаксиса можно указать:

```
NewType Variable;
```

Таким образом будет создан структурированный объект Variable типа NewType. Кроме того, объект Variable мог быть создан непосредственно при объявлении структуры:

```
struct NewType
{
  type1 Name1;
  type2 Name2;
  ...
  typeN NameN;
} Variable;
```

Инициализация элементов структуры может быть произведена непосредственно при объявлении. При этом присваиваемые значения указываются через запятую в фигурных скобках, например:

```
struct MyStruct
{
  int  iVariable;
  long lValue;
  char Str[10];
} mystruct = {10, 300L, "Hello"};
```

Поясним вышесказанное на примере. Предположим, в программе создается *база данных* (подробно базы данных рассмотрены в книге «Базы данных. Учебный курс», издательства «Фолио»), содержащая информацию о жилых домах микрорайона. Изначально известно, что для дальнейшего использования потребуются данные о номере микрорайона, названии улицы, номере дома, количестве этажей, числе квартир, наличии прилегающей стоянки. Разработаем структуру, отвечающую указанным требованиям. Пусть под номер микрорайона отводится беззнаковое короткое целое, название улицы может быть закодировано строкой из 50 символов, номер дома представим как строку из пяти символов (на случай, если номер дома дополнительно содержит букву), количество этажей и число квартир – беззнаковое короткое целое, а информация о наличии стоянки – логическая переменная. В конечном итоге получится что-то вроде:

```
struct HOUSE
{
  unsigned short RegNum;
  char Street[51]; // с учетом '\0'
  char HouseNum[6];
  unsigned short MaxFloorNum;
  unsigned short MaxFlatNum;
  bool Parking;
};
```

Для использования полученного типа HOUSE объявим соответствующую переменную House:

```
HOUSE House;
```

Следующий важный момент – доступ к элементам структуры. Чтобы записать или прочитать данные структуры, после имени объекта ставится символ точки (.), за которым следует имя члена структуры. Вся подобная конструкция рассматривается как единая переменная. В качестве примера заполним уже имеющийся объект House:

```
House.RegNum = 524;
strcpy(MyHouse.Street, "ул. Гоголя");
strcpy(MyHouse.HouseNum, "2-а");
House.MaxFloorNum = 7;
House.MaxFlatNum = 84;
House.Parking = true;
```

Как видно из примера, целочисленным и логическим данным производится обычное присвоение, а заполнение строковых членов структуры осуществлено с помощью функции работы со строками strcpy().

Для определения размера структурированного объекта в памяти к нему применяют оператор (или функцию) sizeof. Таким образом, отдельный экземпляр структуры HOUSE будет занимать, например, 64 байта:

```
int i = sizeof(HOUSE);
```

Однако очень часто структуры помогают экономить память благодаря использованию так называемых *битовых полей*. В этом случае объявление поля структуры имеет вид:

```
объявление_поля : константное_выражение;
```

где

объявление_поля – объявление типа и имени поля структуры;
константное_выражение определяет длину поля в битах.

Тип поля должен быть целочисленным (int, long, unsigned, char). Объявление_поля может отсутствовать. В этом случае в шаблоне структуры пропускается указанное после двоеточия число битов. Таким образом, если разработчик знает наверняка, что элемент структуры может принимать, скажем, всего два значения (0 или 1), для него можно отвести один бит. Дальнейшая работа с таким элементом структуры ведется с использованием поразрядных логических операций (см. раздел 2 «Выражения и операто-

ры»). Реализация битовых полей тесно связана с аппаратной платформой, на которой функционирует компилятор. Поэтому детали использования битовых полей следует уточнить в документации, поставляемой с вашим компилятором.

Рассмотрим небольшой пример. Допустим, необходимо создать структуру, содержащую информацию о дате и времени некоторых событий. Этого можно добиться следующим путем:

```
struct DATETIME
{
  unsigned short Year;   // год
  unsigned short Month;  // месяц
  unsigned short Date;   // дата
  unsigned short Hour;   // часы
  unsigned short Minute; // минуты
  unsigned short Second; // секунды
};
```

Таким образом, объект типа DATETIME в памяти будет занимать 6 (элементов) x 2 (байта) = 12 байт. Нетрудно заметить, что в описании такой структуры присутствует значительная избыточность, так как год может принимать значения от 0 до 99 (задействуется всего 7 бит), месяц – от 1 до 12 (4 бита), дата – от 1 до 31 (5 бит), часы, минуты и секунды – от 0 до 59 (по 6 бит на каждый элемент). После применения битовых структур приведенная выше структура примет вид:

```
struct DATETIME2
{
  unsigned Year  : 7; // год
  unsigned Month : 4; // месяц
  unsigned Date  : 5; // дата
  unsigned Hour  : 6; // часы
  unsigned Minute: 6; // минуты
  unsigned Second: 6; // секунды
};
```

Экземпляр модифицированного типа DATETIME2 будет занимать не 64, а 5 байт (так как 34 бита могут быть размещены только в пяти байтах).

Тема 9.2
Структуры как аргументы функций

Зачастую в тело функции необходимо передать информацию, структурированную по определенному принципу. При этом в качестве параметра в прототипе функции указывается пользова-

тельский тип данных, сформированный при объявлении структуры. Так, если была объявлена структура

```
struct ALLNUMB
{
  int nVar;
  long lVar;
  short shVar;
  unsigned int uiVar;
};
```

прототип функции будет иметь, например, следующий вид:

```
void Func(ALLNUMB);
```

При этом происходит передача в тело функции параметра типа структуры по значению. Таким образом, оригинал структурированного объекта модификации не подлежит. Функция может также возвращать объект типа структуры:

```
ALLNUMB Func2(ALLNUMB);
```

В этом случае модифицированные данные при выходе из функции не потеряются, а будут переданы в точку вызова функции.

Рассмотрим пример передачи в функцию описанной ранее структуры House для вывода на экран названия улицы и номера дома, содержащихся в данной структуре.

```
#include <iostream.h>
#include <string.h>
struct HOUSE
{
  unsigned short RegNum;
  char Street[51]; // с учетом '\0'
  char HouseNum[6];
  unsigned short MaxFloorNum;
  unsigned short MaxFlatNum;
  bool Parking;
};
void OutAddress(HOUSE);
int main()
{
  HOUSE MyHouse;
  MyHouse.RegNum = 524;
  strcpy(MyHouse.Street, "ул. Гоголя");
  strcpy(MyHouse.HouseNum, "2-а");
  MyHouse.MaxFloorNum = 7;
  MyHouse.MaxFlatNum = 84;
  MyHouse.Parking = true;
```

```
  OutAddress(MyHouse);
  return 0;
}
void OutAddress(HOUSE house)
{
  cout << house.Street << ", ";
  cout << house.HouseNum << '\n';
}
```

Вызов функции OutAddress(MyHouse) передает в тело сформированную структуру, доступ к членам которой осуществляется в соответствии с описанным выше правилом, через символ «точка» (".."). В результате на экран будет выведено:

```
ул. Гоголя, 2-а
```

Тема 9.3
Массивы структур

Сама по себе одна единственная запись типа структуры в большинстве случаев вряд ли может вызывать повышенный интерес. Однако в случае, когда структурированные данные объединяются в массивы, речь идет уже не о единичном объекте, а о целой базе данных.

Объявление массива структур отличается от объявления обычных массивов лишь тем, что в качестве типа создаваемого массива указывается вновь образованный тип структуры. Таким образом, для создания базы данных, содержащей информацию о тридцати домах микрорайона, можно объявить следующий массив:

```
HOUSE mDistr[30];
```

Доступ к элементам такого массива осуществляется обычным способом, например по индексу (индексация ведется, начиная с нуля). Следующий фрагмент кода осуществляет в цикле упорядоченное заполнение номеров домов структуры HOUSE, а затем выводит записанные данные в столбик на экран:

```
HOUSE mDistr[30];
for(int i=0; i<30; i++)
  itoa(i+1,mDistr[i].HouseNum,10);
for(int i=0; i<30; i++)
  cout << mDistr[i].HouseNum << '\n';
```

Напомним, что для работы функции преобразования целого числа в строку символов itoa() необходимо подключить модуль stdlib.h.

Поскольку запись структуры очень часто имеет внушительный размер (может включать в себя массивы элементов, другие структуры и т.д.), следует принимать во внимание ограничения по памяти при выборе соответствующей модели.

Тема 9.4
Указатели на структуры. Передача по ссылке членов массивов структур

Разработчик ПО на C++ имеет возможность обращаться к элементам структуры через указатели. Для этого должна быть объявлена соответствующая переменная типа указателя на структуру, синтаксис которой может быть представлен в виде:

```
тип_структуры* идентификатор_указателя;
```

Доступ к элементам структуры через указатель осуществляется с использованием не точки, а символа стрелки (->). Например, создадим указатель на структуру HOUSE:

```
HOUSE *pHouse;
pHouse = &MyHouse;
```

Теперь запишем в структуру информацию о наличии автостоянки около дома посредством объявленного указателя и выведем записанную информацию:

```
pHouse->Parking = true;
cout << MyHouse.Parking;
// или так: cout << pHouse->Parking;
```

Помимо использования указателей возможно применение ссылок на структуры. Объявление такой ссылки имеет следующий синтаксис:

```
тип_структуры  &имя_ссылки = имя_переменной;
```

Как и ссылка на обычную переменную, ссылка на структуру должна быть инициализирована именем объекта, на который она указывает (в данном случае это имя_переменной).

Ссылки и указатели на структуры данных могут быть переданы в качестве аргументов в тело функции. При этом значительно снижается время (в сравнении с передачей по значению), за которое данный параметр передается в функцию, а также экономится стековая память.

Синтаксис прототипа функции при передаче структур посредством указателей и ссылок идентичен синтаксису обычной передачи параметров через указатели и ссылки:

```
// Пример передачи указателя и ссылки на
// целочисленную переменную:
bool Func(int* ptr, int& ref);

// Передача указателя и ссылки на структуру типа HOUSE:
char Func2(HOUSE* pMh, HOUSE& rMh);
```

Таким образом, в функцию Func2 будут переданы не сами значения структур, а соответствующие адреса, что в значительной степени экономит стековую память.

Тема 9.5
Объединения и операции с ними

В C++ имеется особая конструкция, объявление которой напоминает синтаксис объявления структуры, но имеющая совершенно другой физический смысл. Речь идет об *объединениях* (иногда называются *союзами*).

Объединение служит для размещения в одной и той же области памяти (по одному и тому же адресу) данных различных типов. Очевидно, что в отдельный момент времени в памяти может находиться только один из указанных при объявлении объединения типов.

Объявление объединения начинается с ключевого слова union, за которым следует идентификатор и блок описания элементов различного типа, например:

```
union MyData
{
  int iVar1;
  unsigned long ulFreq;
  char Symb[10];
};
```

При этом в памяти компилятором резервируется место для размещения самого большого элемента объединения, в данном случае – 10 байт под символьный массив. Чаще всего объединения включают в состав структур, а также они служат для преобразования данных одного типа в другой. Доступ к элементам объединения производится с помощью операции «точка» (.). Ниже приводится пример такого объединения.

В отличие от структур, переменная типа объединения может быть проинициализирована только значением первого объявленного члена (в данном случае – целочисленной переменной iVar1):

```
union MyData
{
  int iVar1;
  unsigned long ulFreq;
  char Symb[10];
} myX = {25};
```

В качестве примера рассмотрим использование объединений и структур для неявного преобразования данных, представленных в десятичном формате, в двоично-десятичный код. Как известно, двоично-десятичные данные являют собой разновидность упакованных значений, в которых для представления десятичной цифры используется одна *тетрада* (четыре бита). Таким образом, двоично-десятичное число может принимать значения от 0x00 до 0x99 или в десятичном виде – от 0 до 153. Для решения поставленной задачи сконструируем объединение BCD, содержащее однобайтный член. В качестве другого элемента объединения создадим однобайтную структуру с применением битовых полей:

```
#include <iostream.h>
union BCD
{
  unsigned char data;
  struct
  {
    unsigned char lo : 4;
    unsigned char hi : 4;
  } bin;
} bcd;
int main()
{
  bcd.data = 152;
  cout << "\nHigh: " << (int)bcd.bin.hi;
  cout << "\nLow : " << (int)bcd.bin.lo;
  return 0;
}
```

В теле функции main() производится инициализация двоично-десятичной переменной bcd двоичным значением 152 с последующим ее разложением на старший и младший полубайты.

Тема 9.6
Пользовательские типы данных

Помимо структур и объединений разработчик программного обеспечения на C++ имеет возможность моделирования новых типов данных на базе уже имеющихся в его распоряжении типов. Формирование пользовательских типов осуществляется с использованием ключевого слова typedef, за которым указывается какой-либо из имеющихся типов данных и следующий за ним идентификатор, назначающий новое имя для выбранного типа. Фактически, таким образом определяется *синоним* типа данных. Например, выражение

```
typedef unsigned char byte;
```

определяет новый тип данных byte, суть которого такая же, как и типа unsigned char. Следовательно, в программе возможно определение переменных типа byte, которые в памяти будут занимать один байт и смогут принимать значения от 0 до 255, например:

```
byte nInput = 0xFF;
byte nOutput = 0;
```

Синтаксис объявления пользовательского типа для массива выглядит следующим образом:

```
typedef char Names[27];
```

В этом случае объявление переменной типа Names

```
Names name;
```

будет означать, что такая переменная представляет собой строку из 27 символов.

Наиболее часто переопределение типов данных используется совместно со структурами. Благодаря этому становится возможным создание новых типов данных, характерных для сложных объектов, объединяющих разнотипные характеристики. Например, создадим тип данных COORD, объекты которого будут нести информацию о трех координатах в пространстве:

```
typedef struct
{
  double x;
  double y;
  double z;
} COORD;
```

Теперь можно объявить объект myPoint типа COORD:

```
COORD myPoint;
```

и обращаться к его элементам так, как если бы он был объявлен как соответствующая структура:

```
myPoint.x = 5.654;
myPoint.y = 0;
myPoint.z = 3.14;
```

Тема 9.7
Функции работы с датой и временем

Функции и типы данных, необходимые для работы с датой и временем, объявлены в заголовочном файле time.h. В частности, этот файл содержит определения типа данных time_t:

```
typedef long time_t;
```

и структуры tm, которая объявлена следующим образом:

```
struct tm
{
  int tm_sec;   // секунды
  int tm_min;   // минуты
  int tm_hour;  // часы (0–23)
  int tm_mday;  // дата (1–31)
  int tm_mon;   // месяц (0–11)
  int tm_year;  // год (текущий год минус 1900)
  int tm_wday;  // день недели (0–6; Воскр = 0)
  int tm_yday;  // день в году (0–365)
  int tm_isdst; // 0, если зимнее время
};
```

Ниже приводится табл. 9.1, обобщающая часть объявленных функций с кратким описанием их работы.

Таблица 9.1
Функции работы с датой и временем

Наименование	Краткое описание
asctime	Преобразует время и дату из формата структуры типа tm в символьную строку
clock	Возвращает число «тиков» процессора, прошедших от начала запущенного процесса

Наименование	Краткое описание
ctime	Преобразует время и дату из формата time_t в символьную строку
difftime	Вычисляет интервал между двумя заданными временными параметрами
gmtime	Преобразует дату и время из формата time_t в формат структуры tm по Гринвичу (GMT)
localtime	Преобразует дату и время из формата time_t в формат структуры tm
mktime	Преобразует дату и время в календарный формат
_strdate	Преобразует текущую дату в символьную строку в формате mm/dd/yy
strftime	Форматирует время для последующего вывода
_strtime	Возвращает текущее системное время в виде символьной строки
time	Возвращает время в секундах, прошедшее с полуночи (0 часов 0 минут 0 секунд) 1 января 1970 г. по Гринвичу
tzset	Устанавливает значения глобальных переменных _daylight, _timezone и _tzname

В качестве обобщающего примера рассмотрим программу работы с датой и временем:

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

int main()
{
  time_t tt;
  tm *pMyTime;
  tt = time(NULL);
  pMyTime = localtime(&tt);
  cout << "Текущее время: ";
  cout << asctime(pMyTime);
```

```
    return 0;
}
```

В рассматриваемом примере используется функция time(), имеющая прототип

```
time_t time(time_t *timer);
```

возвращающая число секунд, прошедших с 00:00:00 1 января 1970 г. Параметр timer также принимает возвращаемое функцией значение. Для более удобной работы с датой и временем полученные данные преобразуются функцией localtime() к формату структуры tm. Применение функции asctime() позволяет вывести полученную таким образом информацию в виде строки символов.

Практикум «Структуры и объединения»

Упражнение 9.1
Операции со структурами

Проанализируйте предлагаемый ниже пример, поясняющий обращение к элементам структуры. Пусть необходимо создать некоторый справочник, содержащий информацию о знакомых, а именно: имя, фамилию, телефон и город конкретного человека. Для хранения такого рода информации воспользуемся структурой RECORD. Тогда в теле программы достаточно вызвать заполнение данной структуры соответствующими данными:

```
#include <iostream.h>
struct RECORD
{
   int RecNo;
   char FirstName[31];
   char LastName[31];
   char Phone[11];
   char City[16];
};
int main()
{
   RECORD Rec;
   Rec.RecNo = 1;
   strcpy(Rec.FirstName, "Ivan");
```

```
    strcpy(Rec.LastName, "Petrov");
    strcpy(Rec.Phone, "223-322");
    strcpy(Rec.City, "Kharkov");
    cout << Rec.RecNo << Rec.FirstName << Rec.LastName;
    cout << Rec.Phone << Rec.City;
    return 0;
}
```

Упражнение 9.2
Структуры и функции

В данном практическом задании необходимо модифицировать предыдущий пример с тем, чтобы заполнение полей структуры происходило в некоторой функции AddRec().

```
#include <iostream.h>
struct RECORD
{
   int RecNo;
   char FirstName[31];
   char LastName[31];
   char Phone[11];
   char City[16];
};
RECORD AddRec(char* fName, char* lName, char* phone, char* city);
int main()
{
   RECORD Rec;
   Rec = AddRec("Ivan", "Petrov", "223-322", "Kharkov");
   cout << Rec.RecNo << Rec.FirstName << Rec.LastName;
   cout << Rec.Phone << Rec.City;
   return 0;
}
RECORD AddRec(char* fName, char* lName, char* phone, char* city)
{
   static int recno = 0;
   RECORD rec;
   recno++;
   rec.RecNo = recno;
   strcpy(rec.FirstName, fName);
   strcpy(rec.LastName, lName);
   strcpy(rec.Phone, phone);
   strcpy(rec.City, city);
   return rec;
}
```

Упражнение 9.3
Структурированные массивы

Теперь, когда мы подошли к созданию полноценного телефонного справочника, в приведенном задании необходимо с помощью массива структур Rec[10] заполнить справочник данными десяти клиентов. Вывод информации на консоль будем осуществлять путем передачи в функцию структурированного массива и соответствующего индекса элемента массива.

```
#include <iostream.h>
struct RECORD
{
   int RecNo;
   char FirstName[31];
   char LastName[31];
   char Phone[11];
   char City[16];
};
RECORD AddRec(char* fName, char* lName, char* phone, char* city);
void PrintRec(RECORD rec[], int index);
int main()
{
   RECORD Rec[10] = {0};
   Rec[0] = AddRec("Ivan", "Petrov", "223-322", "Kharkov");
   PrintRec(Rec, 0);
   return 0;
}
RECORD AddRec(char* fName, char* lName, char* phone, char* city)
{
   static int recno = 0;
   RECORD rec;
   recno++;
   rec.RecNo = recno;
   strcpy(rec.FirstName, fName);
   strcpy(rec.LastName, lName);
   strcpy(rec.Phone, phone);
   strcpy(rec.City, city);
   return rec;
}
void PrintRec(RECORD rec[], int idx)
{
   cout << rec[idx].RecNo << rec[idx].FirstName;
   cout << rec[idx].LastName << rec[idx].Phone;
   cout << rec[idx].City;
}
```

Упражнение 9.4
Указатели на структуры

В данной лабораторной работе модифицируйте приведенный выше пример таким образом, чтобы вместо самого массива структур использовался указатель на него.

Упражнение 9.5
Объединения

Рассмотрим структуру, в которой заранее неизвестен пол человека, данные о котором необходимо сохранить. Если помещаемые данные относятся к женщине – необходимо хранить в структуре ее девичью фамилию, а если к мужчине – указать признак, служил ли он в рядах Вооруженных Сил.

```
#include <iostream.h>
union Union
{
   char VirgSurname[30];
   bool ArmySrv;
};
typedef struct
{
   char FirstName[30];
   char LastName[30];
   Union Additional;
}Record;
int main()
{
   Record Rec;
   strcpy(Rec.FirstName, "Ivan");
   strcpy(Rec.LastName, "Petrov");
   Rec.Additional.ArmySrv = true;
   return 0;
}
```

Раздел 10
Директивы препроцессора

Тема 10.1
Директивы

Директивы препроцессора представляют собой команды компилятору, которые позволяют управлять компиляцией программы и сделать ее код более понятным. Все директивы начинаются с символа #. Перед начальным символом # и вслед за ним могут располагаться пробелы. Директивы обрабатываются во время первой фазы компиляции специальной программой – препроцессором.

С директивой #include мы уже встречались. Ради полноты изложения здесь мы приведем формальное описание ее использования. Директива #include позволяет включить в текст файла с исходным кодом текст, содержащийся в другом файле. Синтаксис ее использования следующий:

```
#include <header_name>
#include "header_name"
#include macro_identifier
```

Здесь header_name должно быть именем файла с расширением или именем заголовка в новом стиле. Традиционно используется расширение .h или .hpp. Кроме того, в header_name может указываться путь к файлу. Препроцессор удаляет директиву #include из текста файла с исходным кодом и направляет содержимое указанного в ней файла для обработки компилятором. Если указан путь к файлу, компилятор ищет его в указанном каталоге. Первая и вторая формы синтаксиса различаются используемым компилятором алгоритмом поиска файла, если полный путь к нему не задан. Форма <header_name>, в которой имя заголовочного файла задается в угловых скобках, указывает компилятору, что заголовочный файл следует искать в подкаталоге \include компилятора. Форма "header_name", в которой имя заголовочного фай-

ла задается в двойных кавычках, указывает, что заголовочный файл следует искать в *текущем* каталоге, затем – в подкаталоге \include компилятора.

Первая и вторая версии не подразумевают никакого раскрытия макроса. В третьей версии предполагается, что существует макроопределение, которое раскрывается в допустимое имя заголовочного файла.

Директива #define позволяет определить некоторый идентификатор или связать его с некоторой последовательностью лексем. В последнем случае он служит для определения макроса. Макросы мы рассмотрим чуть позже, а здесь приведем пример использования директивы #define для определения некоторого идентификатора или символической константы:

```
#define __cplusplus
```

После того, как символическая константа определена в данном файле, она может быть использована в других директивах препроцессора. Отменить определение символической константы можно с помощью директивы #undef. Например:

```
#undef __cplusp
```

Директива #ifdef позволяет проверить, определена ли символическая константа, и если да, следующие за ней строки будут направлены для обработки компилятором. Она относится к так называемым *условным директивам*, поскольку проверяет выполнение некоторого условия и, в зависимости от результата проверки, изменяет процесс компиляции. Используемая совместно с данной (и другими условными директивами), директива #endif сообщает препроцессору о конце условного блока кода.

Приведем примеры использования этих директив:

```
#ifdef _DEBUG
//Какая-то часть исходного кода, которая должна
// выполняться в отладочной версии программы
#endif
```

Другой случай использования этих директив, который вы часто можете встретить в стандартных заголовочных файлах – это случай использования функций языка C. В этом случае необходимо изменить объявление функции, добавив к нему модификатор extern "C", запрещающий использование декорирования имен. Обычно это выглядит следующим образом:

```
#ifdef __cplusplus
extern "C" {
```

```
#endif
//Объявления заголовочного файла,
//требующие запрета декорирования имен
#ifdef __clusplus
}
#endif
```

Сходной с директивой #ifdef является директива #ifndef. Эта директива также проверяет существование указанного в ней идентификатора, однако следующие за ней строки кода передаются компилятору, если этот идентификатор *не* определен. Приведем пример ее использования:

```
#ifndef WINVER
#define WINVER 0x0400
#endif
```

Директива #else применяется совместно с условными директивами и позволяет отделить часть кода, которая будет обрабатываться, если предыдущее условие не выполняется. Например:

```
#ifdef __FLAT__
#include <win32\windowsx.h>
#else
#include <win16\windowsx.h>
#endif
```

Две другие условные директивы – #if и #elif – также используются для проверки условия. Имя последней из них является сокращением английских слов "else if". Она позволяет проверить условие, альтернативное установленному в директиве #if. Приведем пример их использования:

```
#if _MSC_VER < 901
#elif _MSC_VER < 1001
#endif
```

При использовании условных директив нужно иметь в виду следующие правила:

- для каждой директивы #if должна присутствовать соответствующая директива #endif;
- директивы #elif и #else являются необязательными;
- если ни одно из выражений не истинно, компилируется секция кода, следующая за #else;
- значение выражения должно быть целой константой;
- в выражениях могут применяться операции сравнения ==, >, >=, <, <=.

Директива #line изменяет внутренний счетчик строк компилятора. Синтаксис ее использования следующий:

```
#line целая_константа  <"имя_файла">
```

Если ваша программа состоит из частей, составленных из других программных файлов, часто оказывается полезным пометить такие части номерами строк исходного файла, из которого они взяты, а не последовательными номерами, которые получаются в результате в объединенном файле. Директива #line указывает, что следующий номер строки исходного кода начинается с целой константы из файла с заданным именем, указанной в этой директиве. Если имя файла однажды указано в этой директиве, все последующие директивы #line задают нумерацию относительно того же файла. По умолчанию нумерация осуществляется относительно текущего файла. Макросы раскрываются в аргументы директивы #line. Эта директива чаще всего используется различными утилитами, поставляемыми с компилятором. Приведем пример использования этой директивы:

```
//Исходный файл B
#line 1
void AnotherFunc()
{
 //... (тело функции)
}
//Исходный файл A
#include "SourceB"
void main()
{
 //...(тело главного модуля)
 ANotherFunc();
}
```

В этом случае при включении в исходный файл кода из другого файла нумерация его строк начнется с единицы, а не с очередного номера в файле A.

Директива #error указывает компилятору, что нужно напечатать сообщение об ошибке и прекратить компиляцию. Обычно ее используют внутри условных директив. Вот типичный пример ее использования:

```
#if !defined(MYNAME)
#error Должна быть определена константа MYNAME
#endif
```

Директива #pragma позволяет управлять возможностями компилятора. Синтаксис ее использования следующий:

```
#pragma имя_директивы
```

К сожалению, имя_директивы зависит от реализации этой директивы в компиляторе. Поэтому вам придется изучить документацию, относящуюся к конкретному компилятору. В табл. 10.1 приведены несколько имен_директив, встречающихся во всех известных нам компиляторах, и их назначение.

Таблица 10.1
Имена директив и их назначение

Директива	Описание
Argsused	Подавляет предупреждение о неиспользуемом параметре для функции, следующей за директивой
Hdrfile	Задает компилятору имя файла прекомпилированных заголовков. Синтаксис: `#pragma hdrfile "filename"` Здесь filename – имя файла прекомпилированных заголовков
Hdrstop	Сообщает компилятору, что следующая за ней информация не включается в файл прекомпилированных заголовков
Option	Директива предписывает компилятору использовать указанные в ней опции при компиляции следующего за ней кода
Warn	Директива предписывает компилятору разрешить или подавить вывод предупреждающего сообщения. Синтаксис: `#pragma warn [+|-|.]xxx` где xxx – трехбуквенный идентификатор предупреждающего сообщения, используемый в командной строке с опцией -w. Если ему предшествует "+", то выдача сообщения разрешается, если "-", то запрещается. Если номеру сообщения предшествует точка, для него восстанавливается исходное состояние

Директива	Описание
Warning	Директива предписывает компилятору разрешить или подавить вывод предупреждающего сообщения. Синтаксис: #pragma warning(enable:xxxx) #pragma warning(disable:xxxx) #pragma warning(default:xxxx) Первая форма разрешает вывод предупреждающего сообщения компилятора с номером xxxx , вторая – запрещает, третья – восстанавливает поведение компилятора по умолчанию

Приведем примеры использования этой директивы:

```
#if defined(__STDC__)
//Следующая директива запрещает использование
//ключевых слов, не входящих в стандарт ANSI
#pragma warn -nak
#endif

#ifndef _DEBUG
//Следующая директива запрещает вывод предупреждающего
//сообщения с номером 4701
#pragma warning(disable:4701)
#endif

#ifndef _DEBUG
#pragma warning(default:4701)
#endif

#if defined(__STDC__)
#pragma warn .nak
#endif
```

Тема 10.2
Основные принципы использования файлов заголовков. Оператор defined

Заголовочные файлы обычно используются для объявления типов данных, констант, прототипов функций, определения структур и перечислимых типов, а также внешних ссылок, используемых совместно несколькими файлами исходного кода. Подключение объявлений, содержащихся в заголовочном файле, к файлу исход-

ного кода программы производится директивой #include. Тем самым гарантируется, что все файлы, к которым подключен заголовочный файл, содержат одинаковые объявления. С другой стороны, если нужно изменить некоторое объявление, это достаточно сделать в одном месте – в соответствующем заголовочном файле. Реально осуществляемые проекты могут быть достаточно сложны. В них бывает трудно отследить повторное подключение заголовочного файла, которое может произойти неявно. Для предотвращения многократных включений заголовочного файла используется стандартный прием, основанный на комбинации директив препроцессора. Рассмотрим пример:

```
#ifndef _HEADERFILENAME_
#define _HEADERFILENAME_
//текст заголовочного файла
#endif // _HEADERFILENAME_
```

В этом примере в начале заголовочного файла проверяется, не объявлена ли уже символическая константа _HEADERFILENAME_, и если нет (это означает, что данный заголовочный файл еще не подключен), то эта символическая константа объявляется. Затем идет остальной текст заголовочного файла. Последняя строка этого файла содержит директиву #endif, закрывающую объявления, начатые директивой #ifndef.

В директивах #if и #elif может применяться оператор defined. Он позволяет проверить, был ли перед этим определен идентификатор или макрос с указанным в этом операторе именем. Можно также применять операцию логического отрицания (!) для проверки того, что идентификатор или макрос не определен. Следующий пример демонстрирует это:

```
#if defined(__BORLANDC__)
   #if !defined(__MFC_COMPAT__)
      #undef  _ANONYMOUS_STRUCT
   #else
      #define _ANONYMOUS_STRUCT
   #endif
#endif
```

Могут использоваться и более сложные условия для проверки:

```
#if defined(__LARGE__) || defined(__HUGE__) ||\
   defined(__COMPACT__)
 typedef long  ptrdiff_t;
#else
 typedef int   ptrdiff_t;
#endif
```

Приведем пример использования директивы #elif:

```
#if defined(__OS2__)
#   define _RTLENTRY  __stdcall
#   define _USERENTRY __stdcall
#elif defined(__WIN32__)
#   define _RTLENTRY  __cdecl
#   define _USERENTRY __cdecl
#else
#   define _RTLENTRY  __cdecl
#   define _USERENTRY __cdecl
#endif
```

Тема 10.3
Макросы

Макрос – это фрагмент кода, который выглядит и работает так же, как функция. Однако функцией он не является. Имеется несколько различий между макросами и функциями:

- макрос заменяется своим определением во время работы препроцессора, то есть еще до компиляции программы. Поэтому макросы не вызывают дополнительных затрат времени, как при вызове функции;
- использование макросов приводит к разрастанию исходного кода и увеличению размера исполняемой программы. Функции являются в некотором смысле антиподом макросов. Код, который они представляют, встраивается в программу только один раз, что приводит к сокращению кода программы. С другой стороны, при выполнении программы требуется дополнительное время для организации вызова функции;
- компилятор не выполняет никаких проверок типов в макросе. Поэтому при передаче макросу аргумента, не соответствующего подразумеваемому типу, или неверного числа аргументов никаких сообщений об ошибке не будет;
- поскольку макрос является средством для встраивания фрагмента кода, не существует единого адреса, связанного с ним. Поэтому нельзя объявить указатель на макрос или использовать его адрес.

Для определения макросов используется директива #define. Как и функции, макросы могут иметь параметры. Например:

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

```
//Макроопределение
#define MULTIPLY(x,y) ((x)*(y))
int main()
{
 int a = 2, b =3;
 printf("%d", MULTIPLY(a,b));
}
```

Внешне использование макроса похоже на использование функции (из-за чего их иногда называют *псевдофункциями*). Поскольку, как отмечено выше, это все-таки разные объекты, принято имена макросов записывать прописными буквами.

Обратите внимание, что в определении макроса, приведенном выше, параметры заключены в круглые скобки. Если этого не сделать, использование макроса может привести к неожиданным результатам. Например, следующее макроопределение синтаксически правильно:

```
#define SQUARE(x) x*x
```

Если в коде программы встречается такое использование этого макроса

```
int y = SQUARE(2);
```

то в результате раскрытия (или подстановки) макроса получится следующая инструкция:

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

Однако если в коде программы встретится, например, такое использование макроса:

```
int y = SQUARE(x + 1);
```

то в результате раскрытия макроса получится следующая инструкция:

```
int y = x + 1* x + 1;
```

В результате компилятор вместо ожидаемого результата

```
int y = (x + 1)*(x + 1);
```

получит иной (y = 2*x + 1). По этой причине в качестве общего правила рекомендуется всегда заключать в скобки каждый параметр макроса. Кроме того, если вызов макроса может появляться в инструкциях, содержащих операторы приведения типа, например

```
doubleValue = (double)SQUARE(x + 1);
```

то рекомендуется заключать в скобки все тело макроса:

```
#define SQUARE(x) ((x)*(x))
```

И последнее замечание по поводу использования макросов. Обычно в тексте программы дополнительные пробелы не являются значащими. Они служат только для удобства чтения кода. В случае с макросами это не так. Например, пробел между именем макроса и его параметрами может привести к изменению смысла макроса, как в следующем макроопределении:

```
#define BAD_MACRO (x) printf("%d", x)
```

Оно будет раскрыто в следующую инструкцию:

```
(x) printf("%d", x);
```

Это приведет к ошибке компиляции. Чтобы исправить ошибку, достаточно удалить лишний пробел между именем макроса и его параметром:

```
#define BAD_MACRO(x) printf("%d", x)
```

Если определение макроса не умещается на одной строке, оно может быть продолжено в последующих строках. Для продолжения макроопределения в следующей строке достаточно в конце текущей строки поставить знак "\".

Например,

```
#define MAKEWORD(a, b) ((WORD)(((BYTE)(a)) \
|(((WORD)((BYTE)(b))) << 8)))
```

В определении макроса могут участвовать другие макросы, то есть макросы могут быть вложенными. Следующий пример демонстрирует использование вложенных макросов:

```
#define PI 3.14159
#define SQUARE(x) ((x)*(x))
#define CIRCLE_AREA(x) (PI * SQUARE(x))
```

Макрос может быть аннулирован в любом месте программы с помощью директивы #undef.

Тема 10.4
Предопределенные макросы

В приведенной ниже табл. 10.2 перечислены макросы ANSI, которые компилятор C++ определяет автоматически.

Таблица 10.2
Предопределенные макросы ANSI

Макрос	Описание
__DATE__	Строка, представляющая в форме mmm.dd.yyyy дату создания данного файла
__FILE__	Имя текущего обрабатываемого файла
__LINE__	Номер текущей строки обрабатываемого файла
__STDC__	Определен, если установлен режим совместимости с ANSI C
__TIME__	Время начала обработки текущего файла в формате hh:mm:ss

Следующий пример демонстрирует использование этих макросов:

```
#include <stdio.h>
char* Date = __DATE__;
char* Time = __TIME__;
int main()
{
 FILE* file;
printf("Дата создания %s, ""время создания %s\n", Date,Time);
file = fopen("Ex.cpp","r+t");
 if (file == NULL)
  {
   printf("Ошибка в вызове функции fopen()"
                "в файле %s, строка %d\n", __FILE__, __LINE__);
   return 1;
  }
return 0;
}
```

Тема 10.5
Операции, применяемые в директивах препроцессора

Существуют две операции, которые можно использовать в директивах препроцессора: *подстановка строки* (#) и *конкатенация* (##).

Если перед параметром макроса указана операция подстановки символа (#), то компилятор при раскрытии макроса вместо параметра подставит его имя. Как и любая строка, результат этой операции может объединяться со смежными строками, если отделяется он них только пробелами. Рассмотрим пример:

```
#include <stdio.h>
#define SHOWVAL(val)\
printf(#val " = %d \n", (int)(val))
int main()
{
int icount = 10;
SHOWVAL(icount);

 return 0;
}
```

При выполнении программа выводит на экран:

```
icount = 10;
```

Операция конкатенации (или склейки) ## вызывает объединение двух строк в одну. Перед объединением строк удаляются разделяющие их пробелы. Если для полученной в результате склейки строки существует макроопределение, препроцессор выполняет раскрытие макроса. Например:

```
#define MACRO1 printf("Вызов MACRO1.")
#define MACRO2 printf("Вызов MACRO2.")
#define CREATE_MACRO(n) MACRO ## n
CREATE_MACRO(1);
```

В результате раскрытия макроса препроцессором компилятор получит следующую инструкцию:

```
printf("Вызов MACRO1.");
```

Если операция конкатенации используется в макроопределении, то вначале препроцессор выполняет подстановку аргументов вместо формальных параметров, затем выполняет операцию конкатенации. После этого выполняются остальные операции макроопределения. Например:

```
#define DEFVAL(i) int var ## i
DEFVAL(1);
```

В результате раскрытия макроса препроцессором компилятор получит следующую инструкцию:

```
int var1;
```

Практикум «Директивы препроцессора»

Упражнение 10.1
Основные директивы

В данном упражнении Вам предлагается самостоятельно создать заголовочный файл, в котором при помощи директивы define будут определены 2 идентификатора: ON и OF. В основном модуле подключите заголовочный файл и, проверив, определен ли каждый из идентификаторов, выведите соответствующее сообщение.

Упражнение 10.2
Создание макросов

Проанализируйте предлагаемый ниже пример, который иллюстрирует различия в вызовах функции и макроса, осуществляющих одни и те же действия.

Пусть необходимо получить куб некоторого числа. Реализуем эту операцию посредством функции cube() и макроса CUBE(x). Вызов и функции, и макроса даст нам один результат – куб числа, переданного в качестве единственного параметра. Однако если мы захотим использовать преимущества синтаксиса C++ и, например, инкрементируем аргумент при вызове (см. пример ниже), функция и макрос вернут одно и то же значение, хотя аргумент преобразуется неодинаково. В макросе инкрементация произойдет трижды.

```
#include <iostream.h>
int cube(int x)
{
  return x*x*x;
}
#define CUBE(x)  ((x)*(x)*(x))
int main()
{
   int b = 0, a = 3;
   b = cube(a++);
   cout << "cube(a++) = " << b << endl;
   cout << "a = " << a << endl;
   a = 3;
   b = CUBE(a++);
```

```
    cout << "CUBE(a++) = " << b << endl;
    cout << "a = " << a << endl;
    return 0;
}
```

Упражнение 10.3
Предопределенные макросы

Ниже приведена попытка создания своеобразного «таймера», выводящего на консоль текущее время. Объясните, почему этот пример не будет работать.

```
#include <iostream.h>
int main()
{
   for(int i=0; i<10; i++){
      cout << __TIME__ << endl;
      Sleep(1000);
   }
   return 0;
}
```

Упражнение 10.4
Операции в макросах

Модифицируйте приведенный ниже пример таким образом, чтобы вызов макроса, помимо значений переменных, выводил и их имена.

```
#include <iostream.h>
#define COUT(x) cout<<x<<endl
int main()
{
   char Char = 'A';
   int Integer = 323;
   float Float = 3.222;
   double Double = 423.982;
   long Long = 2079879342;
   COUT(Char);
   COUT(Integer);
   COUT(Float);
   COUT(Double);
   COUT(Long);
   return 0;
}
```

Раздел 11
ФУНКЦИИ ВВОДА-ВЫВОДА

Тема 11.1
Текстовые и бинарные (двоичные) файлы

Язык C++ унаследовал от языка C библиотеку стандартных функций ввода-вывода. Функции ввода-вывода объявлены в заголовочном файле <stdio.h>. Операции ввода-вывода осуществляются с файлами. Файл может быть текстовым или бинарным (двоичным). Различие между ними заключается в том, что в текстовом файле последовательности символов разбиты на строки. Признаком конца строки является пара символов CR (возврат каретки) и LF (перевод строки) или, что то же самое, '\r' и '\n'. При вводе информации из текстового файла эта пара символов заменяется символом CR, при выводе, наоборот, – символ CR заменяется парой символов CR и LF. Бинарный (или двоичный) файл – это просто последовательность символов. Обычно двоичные файлы используются в том случае, если они являются источником информации, не предполагающей ее непосредственного представления человеку. При вводе и выводе информации в бинарные файлы никакого преобразования символов не производится.

Тема 11.2
Потоковый ввод-вывод. Стандартные потоки

Термин *поток* происходит из представления процесса ввода-вывода информации в файл в виде последовательности или потока байтов. Над потоком можно выполнять следующие операции:

- считывание блока данных из потока в оперативную память;
- запись блока данных из оперативной памяти в поток;
- обновление блока данных в потоке;
- считывание записи из потока;
- занесение записи в поток.

Все потоковые функции ввода-вывода обеспечивают буферизированный, форматированный или неформатированный ввод и вывод.

Когда начинается выполнение программы, автоматически открываются следующие потоки:

stdin - стандартное устройство ввода;
stdout - стандартное устройство вывода;
stderr - стандартное устройство сообщений об ошибках;
stdprn - стандартное устройство печати;
stdaux - стандартное вспомогательное устройство.

Все они называются *стандартными (*или *предопределенными) потоками ввода-вывода.* По умолчанию стандартным устройством ввода, вывода и сообщений об ошибках является пользовательский терминал. Поток стандартного устройства печати относится к принтеру, а поток стандартного вспомогательного устройства – к вспомогательному порту компьютера. По умолчанию при открытии все стандартные потоки, за исключением потоков stderr и stdaux, буферизуются.

В современных операционных системах клавиатура и дисплей рассматриваются как файлы (притом текстовые!), поскольку информация с клавиатуры в программу может считываться, а на дисплей – выводиться. При запуске программы на выполнение ей можно переназначить стандартное устройство ввода (клавиатуру) или стандартное устройство вывода (дисплей), назначив вместо этих устройств текстовый файл. При этом говорят, что происходит *переадресация* ввода или, соответственно, вывода. Для переназначения ввода используется символ "<", а для переназначения вывода – символ ">". Если выполняемая программа называется example.exe, следующая строка используется для переназначения ввода с клавиатуры на файл sample.dat:

```
example < sample.dat
```

Переназначение вывода с дисплея на вывод в файл осуществляется с помощью следующей строки:

```
example > output.dat
```

Наконец, следующая строка осуществляет одновременное переназначение ввода и вывода:

```
xample < sample.dat > output.dat
```

Можно также осуществить соединение выходного потока одной программы с входным потоком другой. Это называется *кон-*

вейерной пересылкой. Если имеются две выполнимые программы example1 и example2, то конвейерная пересылка между ними организуется с помощью символа вертикальной черты "|". Следующая строка организует конвейерную пересылку между example1 и example2:

```
example1 | example2
```

Организацию конвейерной пересылки обеспечит операционная система.

Тема 11.3
Функции ввода и вывода символов

Эта группа функций ввода-вывода на самом деле реализована в виде макросов.

Макрос getc() возвращает следующий символ из заданного входного потока и увеличивает указатель входного потока так, чтобы он указывал на следующий символ. В случае успеха getc() возвращает считанный символ, преобразованный в тип int без знака. Если прочитан конец файла или произошла ошибка, он возвращает EOF. Прототип этого макроса следующий:

```
int getc(FILE *stream);
```

Символ EOF определяется следующим образом:

```
#define EOF (-1)
```

В операциях ввода-вывода он служит для обозначения и проверки конца файла.

Подразумевается, что этот символ имеет тип signed char. Если в операциях ввода-вывода участвуют символы типа unsigned char, то использовать EOF нельзя.

Следующий пример демонстрирует использование этого макроса:

```
#include <stdio.h>
int main()
{
 char ch;
 printf("Ввод символа:");
 //Чтение символа из стандартного входного потока
 ch = getc(stdin);
 printf("Был введен символ: '%c'\n", ch);
 return 0;
}
```

Макрос getchar() имеет следующий прототип:

```
int getchar();
```

Этот макрос определен как getc(stdin). С помощью этого макроса из стандартного входного потока stdin считывается очередной символ и его значение преобразуется в тип int без учета знака. Если переадресация стандартного входного потока не производилась, то ввод осуществляется с клавиатуры. В противном случае ввод будет осуществляться из файла, назначенного для входного потока и указанного в командной строке при вызове программы.

Макрос putc() имеет следующий прототип:

```
int putc(int c, FILE *stream);
```

Он выводит символ в поток, заданный параметром stream. В случае успеха putc() возвращает выводимый символ c, преобразованный к типу int; в случае ошибки – EOF.

Макрос putchar() имеет следующий прототип:

```
int putchar(int c);
```

Этот макрос определяется как putc(c, stdout). В случае успеха он возвращает символ c, преобразованный к типу int. В случае ошибки он возвращает EOF.

Тема 11.4
Функции ввода и вывода строк

Функция gets() имеет следующий прототип:

```
char *gets(char *s);
```

Она выполняет считывание строки символов из стандартного входного потока и помещает их в переменную s. Символ перехода на новую строку "\n" заменяется символом "\0" при помещении в строку s. При использовании этой функции следует соблюдать осторожность, чтобы число символов, считанных из входного потока, не превысило размер памяти, отведенной для строки s.

Функция puts() имеет следующий прототип:

```
int puts(const char *s);
```

Она выводит строку в стандартный выводной поток и добавляет символ перевода на новую строку "\n". В случае успеха puts() возвращает неотрицательное значение. В противном случае она возвращает EOF.

Приведем пример использования функции вывода строк:

```
#include <stdio.h>
int main()
{
 char* string = "Выводимая строка";
 puts(string);
 return 0;
}
```

Следующие функции выполняют форматированный ввод или вывод.

Функция scanf() имеет следующий прототип:

```
int
scanf(const char *format[, address, ...]);
```

Она позволяет осуществлять форматированный ввод информации из стандартного потока ввода. Функция сканирует (откуда и ее название) последовательность входных полей по одному символу и форматирует каждое поле в соответствии со спецификатором формата, переданном в строке форматирования format. Для каждого поля должен существовать спецификатор формата и адрес переменной, предназначенной для размещения результата преобразования поля в соответствии с заданным спецификатором формата. Функция может прекратить сканирование поля до достижения конца поля (символа-заполнителя) или прекратить сканирование полей вообще. В случае успеха scanf() возвращает число успешно просканированных, преобразованных и сохраненных входных полей. В это число не входят несохраненные поля. Если ни одно поле не было сохранено, функция возвращает 0. Если делается попытка прочесть конец файла или конец строки, она возвращает EOF.

Строка форматирования format представляет собой символьную строку, которая содержит объекты трех типов:

- символы-заполнители;
- символы, отличные от символов-заполнителей;
- спецификаторы формата.

Символы-заполнители – это пробел, символ табуляции (\t) и перевод строки (\n). Эти символы считываются из форматной строки, но не сохраняются.

Символы, отличные от символов-заполнителей – это все остальные ASCII-символы, за исключением знака процента (%). Эти символы в форматной строке считываются, но не сохраняются.

Спецификаторы формата направляют процесс сканирования, считывания и преобразования символов из входного поля в переменные, заданные своими адресами. Каждому спецификатору формата должен соответствовать адрес переменной. Если спецификаторов формата больше, чем переменных, результат непредсказуем. Наоборот, если имеется больше переменных, чем спецификаторов формата, «лишние» переменные игнорируются.

Спецификаторы формата имеют следующий вид:

```
% [*] [width] [F|N] [h|l|L] type_char
```

Каждый спецификатор формата начинается с символа процента (%).

После знака процента идут следующие знаки в указанном в табл. 11.1 порядке.

Таблица 11.1

Знаки спецификатора формата и их назначение

Компонент	Обязательный или нет	Назначение
[*]	Нет	Символ подавления присвоения переменной значения следующего поля. Текущее поле ввода сканируется, но не сохраняется в переменной. Предполагается, что аргумент, соответствующий спецификатору формата, содержащему звездочку, имеет тип, указанный символом типа преобразования (type_char), который идет за звездочкой
[width]	Нет	Спецификатор ширины поля. Задает максимальное число считываемых символов. Функция может прочесть меньше символов, если в потоке ввода встретится символ-заполнитель или непреобразуемый символ

Компонент	Обязательный или нет	Назначение
[F|N]	Нет	Модификатор величины указателя. Переопределяет величину по умолчанию аргумента, задающего адрес: N = near pointer F = far pointer
[h|l|L]	Нет	Модификатор типа аргумента. Переопределяет тип по умолчанию аргумента, задающего адрес: h = short int; l = long int, если type_char задает преобразование в целое; l = double, если type_char задает преобразование в тип с плавающей точкой; L = long double (верно только для преобразования в тип с плавающей точкой)
type_char	Да	Символ типа (преобразования) (или спецификатор преобразования)

Возможные значения символа типа приведены в табл. 11.2.

Таблица 11.2
Символы типа преобразования

Тип данных	Ожидаемый ввод	Тип аргумента
Числовой		
D	Десятичное целое	Указатель на целое (int *arg)
D	Десятичное целое	Указатель на длинное целое (long *arg)
E,E	Число с плавающей точкой	Указатель на float (float *arg)
f	Число с плавающей точкой	Указатель на float (float *arg)

Тип данных	Ожидаемый ввод	Тип аргумента
G,G	Число с плавающей точкой	Указатель на float (float *arg)
o	Восьмеричное целое	Указатель на int (int *arg)
O	Восьмеричное целое	Указатель на long (long *arg)
i	Десятичное, восьмеричное или шестнадцатеричное целое	Указатель на int (int *arg)
I	Десятичное, восьмеричное или шестнадцатеричное целое	Указатель на long (long *arg)
u	Беззнаковое десятичное целое	Указатель на unsigned int (unsigned int *arg)
U	Беззнаковое десятичное целое	Указатель на unsigned long (unsigned long *arg)
x	Шестнадцатеричное целое	Указатель на int (int *arg)
X	Шестнадцатеричное целое	Указатель на int (int *arg)
Символы		
s	Символьная строка	Указатель на символьный массив (char arg[])
c	Символ	Указатель на символ (char *arg), если задана ширина поля для символов С-типа (например, %5c). Указатель на массив из N символов (char arg[N])

Тип данных	Ожидаемый ввод	Тип аргумента
%	% символ	Никакое преобразование не выполняется; знак % сохраняется
Указатели		
n	Указатель на int (int *arg)	Число успешно прочитанных символов, вплоть до %n, сохраняется в этом int.
p	Шестнадцатеричная форма YYYY:ZZZZ или ZZZZ	Указатель на объект (far* или near*). По умолчанию, %p преобразовывает в указатель, величина которого принята в данной модели памяти

Кроме того, в качестве спецификатора формата можно задать совокупность символов, заключенных в квадратные скобки. Соответствующий ему аргумент должен указывать на массив символов. Эти квадратные скобки определяют *совокупность символов поиска.* Если первым символом в квадратных скобках является каре (^), то условие соответствия инвертируется: во входном поле ищутся символы, *не* соответствующие указанным в квадратных скобках. Например:

%[abcd] – поиск во входном поле любого из символов a, b, c и d;

%[^abcd] – поиск во входном поле любого из символов, за исключением a, b, c и d.

Вы также можете указать диапазон поиска, задав символы через тире в качестве совокупности символов поиска. Например:

%[0-9] – поиск во входном поле любых десятичных цифр;

%[0-9A-Za-z] – поиск во входном поле любых десятичных цифр и букв.

Сканирование поля прекращается, если очередной символ не попадает в совокупность символов поиска.

Функция может прекратить сканирование какого-то поля еще до достижения его естественного конца (символа-заполнителя)

или даже прекратить сканирование совсем. В частности, это происходит, когда в поле ввода появляется символ, не соответствующий по типу заданному преобразованию в спецификаторе формата. В этом случае конфликтующий символ остается в поле ввода. Таким образом, использование функции scanf() может приводить к неожиданным результатам, если вы отходите от заданного формата. По этой причине рекомендуется вместо данной функции использовать комбинацию функций gets() и sscanf().

Рассмотрим пример, демонстрирующий возможности функции scanf():

```
#include <stdio.h>
int main()
{
 int i, res;
 float fp;
 char  c, s[81];
 printf( "\n\nEnter an int, a float, ""char and string\n");
 res = scanf( "%d %f %c  %s", &i, &fp, &c, s);
 printf( "\nThe number of fields ""input is %d\n", res);
 printf( "The contents are:\n"" %d %f %c  %s  \n", i, fp, c, s);
 return 0;
}
```

При выполнении программа выводит на экран:

```
Enter an int, a float, char and string
1234 1234.5678 F string

The number of fields input is 4
The contents are:
1234 1234.567749 F  string
```

Упомянутая выше функция sscanf() имеет следующий прототип:

```
int
sscanf(const char *buffer,
const char *format[, address, ...]);
```

Она во всем аналогична функции scanf(), за исключением того, что выполняет сканирование и форматирование ввода из строки символов. В частности, она использует те же спецификаторы формата в строке форматирования. Прочитанные из символьной строки в соответствии с заданным форматом значения помещаются в переменные, адреса которых заданы в списке аргументов функции. В случае успеха sscanf() возвращает число входных полей, успешно просканированных, преобразованных к

нужному формату и сохраненных в переменных. Если в sscanf() делается попытка прочесть конец строки, она возвращает EOF. В случае ошибки (если ни одно поле не было сохранено в переменной), она возвращает 0. Приведем пример использования функции sscanf():

```
#include <stdio.h>
int main()
{
 char  szBuf1[] = "***Эта функция может разбирать строку***";
 char  szBuf2[] = "Integer i = 12345 ,fl = 1234.567";
 char  s[5][11]; char  str1[11], str2[11];
 char  c1, c2, c3; int i; float fl;
 //Ввод различных данных из буфера строки
 sscanf( szBuf1, "%s %s %s %s %s", s[0], s[1], s[2],s[3], s[4]);
 sscanf( szBuf2, "%s %c %c %d %s %c %f",
       str1, &c1, &c2, &i, str2, &c3, &fl);
 //Вывод прочитанных данных
 printf( "sscanf() parse this string: \n""%s %s %s %s %s\n",
       s[0], s[1], s[2],s[3], s[4]);
 printf( "szBuf2 contains:\n%s %c %c %d %s %c %8.3f",
       str1, c1, c2, i, str2, c3, fl);
 return 0;
}
```

При выполнении программа выводит на экран:

```
sscanf() parse this string:
***Эта функция может разбирать строку***
szBuf2 contains:
Integer i = 12345 ,fl = 1234.567
```

Функция printf() имеет следующий прототип:

```
int printf(const char *format [, argument, ...]);
```

Она позволяет осуществлять форматированный вывод в стандартный поток вывода stdout. Функция принимает последовательность аргументов, применяя к каждому аргументу спецификатор формата, который содержится в строке форматирования format. Если аргументов меньше, чем спецификаторов формата, результат непредсказуем. С другой стороны, если аргументов больше, чем спецификаторов формата, лишние аргументы просто игнорируются. В случае успеха функция возвращает число выведенных байтов (без учета завершающего нуль-символа); в случае ошибки – значение EOF.

Строка форматирования в функции printf() управляет тем, как эта функция будет преобразовывать, форматировать и выво-

дить свои аргументы. Она представляет собой символьную строку, которая содержит два типа объектов:

- простые символы, которые копируются в выходной поток;
- спецификаторы формата, которые применяются к аргументам, выбираемым из списка аргументов.

Спецификаторы формата имеют следующий вид:

%[flags][width][.prec][F|N|h|l|L]type_char

Каждый спецификатор формата начинается с символа процента (%), после которого идут необязательные спецификаторы в порядке, указанном в табл. 11.3.

Таблица 11.3

Компоненты спецификатора формата и их назначение

Компонент	Обязательный или нет	Назначение
[flags]	Нет	Флаговые символы. Управляют выравниванием, знаком числа, десятичной точкой, конечными нулями, префиксами для восьмеричных и шестнадцатиричных чисел
[width]	Нет	Спецификатор ширины (поля). Указывает минимальное число выводимых символов (дополняемое в случае необходимости пробелами или нулями)
[prec]	Нет	Спецификатор точности. Указывает максимальное число выводимых символов; для целых чисел – минимальное число выводимых цифр
[F|N|h|l|L]	Нет	Модификатор размера. Переопределяет размер входного аргумента по умолчанию
Type_char	Да	Символ типа преобразования

Флаговые символы (флажки) могут появляться в любом порядке и комбинации.

Типы флажков и их назначение приведены в табл.11.4.

Таблица 11.4
Флажки и их назначение

Флажок	Назначение
-	Выравнивает результат преобразования влево, дополняя его справа пробелами. По умолчанию (т.е. когда этот флажок не задан) результат преобразования выравнивается вправо и дополняется слева нулями или пробелами
+	Если результат преобразования имеет знак, этот знак всегда выводится. Это означает, что для положительных чисел выводится знак плюс (+)
пробел (' ')	Если значение неотрицательно, вывод начинается с пробела вместо знака плюс. Отрицательные числа по-прежнему выводятся со знаком минус
#	Задает, что аргумент преобразуется с использованием альтернативной формы

Замечание: Флажок «плюс» (+) имеет преимущество перед флажком «пробел» (' '), если они заданы оба.

Спецификатор ширины поля устанавливает минимальную ширину поля для выводимого значения. Ширина задается одним из двух способов:

- непосредственно, с помощью десятичного числа;
- косвенно, с помощью символа звездочки (*).

Если Вы используете в качестве спецификатора ширины звездочку, следующий аргумент в списке аргументов (который должен быть целым числом) задает не выводимое значение, а минимальную ширину поля для вывода. Если ширина поля вывода не указана или мала, это не вызывает усечения значения до ширины поля. Вместо этого поле расширяется до нужного размера, чтобы вместить результат преобразования значения. Спецификаторы ширины и их влияние на вывод перечислены в табл. 11.5.

Спецификатор точности функции printf() устанавливает максимальное число выводимых символов или минимальное число выводимых цифр для целых чисел. Этот спецификатор всегда на-

чинается с точки, чтобы отделить его от предыдущего спецификатора ширины. Как и спецификатор ширины, он задается одним из двух способов:

- непосредственно, с помощью десятичного числа;
- косвенно, с помощью символа звездочки (*).

Таблица 11.5
Спецификаторы ширины и их влияние на вывод

Спецификатор ширины	Влияние на вывод
N	По крайней мере n символов выводятся. Если выводимое значение имеет менее чем n символов, вывод дополняется пробелами (справа, если указан флажок минус (-); слева – в противном случае)
0n	Выводится по меньшей мере n символов. Если выводимое значение имеет меньше чем n символов, оно дополняется слева нулями
*	Список аргументов предоставляет спецификатор ширины, который должен предшествовать реально выводимому аргументу

Если Вы используете звездочку в качестве спецификатора точности, то следующий аргумент в списке аргументов (который должен быть целым числом) задает точность. Если Вы задаете звездочки в качестве спецификаторов ширины и точности, то вслед за ними в списке аргументов должен идти спецификатор ширины, затем спецификатор точности, а за ними – само выводимое значение.

Символ типа преобразования задает преобразование типа выводимого аргумента. Возможные значения символа типа преобразования и их назначение представлены в табл. 11.6. Информация в этой таблице основывается на предположении, что спецификатор формата не содержит ни флажков, ни спецификатора ширины, ни спецификатора точности, ни модификатора размера входного значения.

Модификаторы размера определяют, как функция printf() интерпретирует следующий входной аргумент. Возможные значения модификатора размера и их действие представлены в табл. 11.7.

Таблица 11.6

Значения символа типа преобразования и их назначение

Символ типа преобразования	Ожидаемый ввод	Формат вывода
Числовые значения		
d	Целое число	десятичное целое со знаком
i	Целое число	десятичное целое со знаком
o	Целое число	восьмеричное целое без знака
u	Целое число	десятичное целое без знака
x	Целое число	шестнадцатеричное целое без знака (причем для 16-ричных цифр используются символы a, b, c, d, e, f)
X	Целое число	шестнадцатеричное целое без знака (причем для 16-ричных цифр используются символы A, B, C, D, E, F)
f	Число с плавающей точкой	Число с плавающей точкой в виде [-]dddd.dddd
e	Число с плавающей точкой	Число с плавающей точкой в виде [-]d.dddd или e[+/-]ddd
g	Число с плавающей точкой	Число с плавающей точкой в форме e или f с учетом заданного значения и точности. Конечные нули и десятичная точка выводятся, если это необходимо
E	Число с плавающей точкой	То же, что и e; но для экспоненты используется символ E
G	Число с плавающей точкой	То же, что и g; но для экспоненты используется символ E, если используется формат e

Символ типа преобразования	Ожидаемый ввод	Формат вывода
Символы		
c	Символ	Один символ
s	Указатель на строку	Выводит символы до тех пор, пока не встретится 0-символ или не будет достигнута заданная точность
%	Никакой	Выводит символ процента
Указатели		
n	Указатель на int	Сохраняет по адресу, указанному входным аргументом, число символов, записанных досих пор
p	Указатель	Выводит входной аргумент как указатель; формат зависит от используемой модели памяти. Указатель будет представлен в виде XXXX:YYYY или YYYY (только смещение)

Таблица 11.7

Модификаторы размера и их действие на входную величину

Модификатор размера	Символ типа преобразования	Приписываемый входному аргументу тип
F	p s	far-указатель
N	N	near-указатель
H	D i o u x X	short int
L	d i o u x X	long int
L	e E f g G	double
L	e E f g G	long double
L	d i o u x X	__int64
h	c C	Однобайтовый символ

Модификатор размера	Символ типа преобразования	Приписываемый входному аргументу тип
l	c C	Широкий символ
h	s S	Строка однобайтовых символов
l	s S	Строка широких символов

Модификаторы F и N изменяют интерпретацию входного аргумента в случае, если задан символ преобразования %p, %s или %n, трактуя его, соответственно, как дальний или ближний указатель. По умолчанию для этих символов преобразования входной аргумент трактуется как указатель в используемой модели памяти. Модификаторы размера h, l и L переопределяют величину по умолчанию числовых входных аргументов, как указано в табл. 11.7.

Приведем пример, демонстрирующий различные возможности функции printf():

```
#include <stdio.h>
int main()
{
 char   ch = 'A', *string = "string";
 int    count = -1234;
 double fp = 123.4567;
 //Отображение целых
 printf("Целочисленные форматы:\n\ tДесятичный: %d  "
        "Выровненный: %.6d\n Беззнаковый: %u\n",
        count, count, count, count);
 printf("Десятичный %d как:\n\tШестнадцатеричный: %Xh
        С-Шестнадцатеричный: 0x%x Восьмеричный: %o\n",
        count, count, count, count);
 //Отображение чисел с разным основанием
 printf("Число 10 эквивалентно:\n\tШестнадцатеричный: %i
         Восьмеричный: %i Десятичный: %i\n",
         0x10, 010, 10);
 //Отображение символов
 printf("Символы:\n%10c%5hc\n", ch, ch);
 //Отображение строк
 printf("Строки:\n%16s\n%16.4hs\n", string, string);
 //Отображение чисел с плавающей точкой
 printf("Действительные числа:\n%f %.2f %e %E\n", fp, fp, fp, fp);
 //Отображение указателя
 printf("\nАдрес как:\t%p\n", &count);
 //Вывод числа символов
```

```
    printf("1234567890123456%n78901234567890\n", &count);
    printf("Выведено чисел: %d\n\n", count );
    return 0;
}
```

При выполнении программа выводит на экран:

```
Целочисленные форматы:
 Десятичный: -1234 Выровненный: -001234 Беззнаковый: 4294966062
Десятичный -1234 как:
 Шестнадцатеричный: FFFFFB2Eh С-Шестнадцатеричный: 0xfffffb2e Восьмеричный: 37777775456
Число 10 эквивалентно:
 Шестнадцатеричный: 16 Восьмеричный: 8 Десятичный: 10
Символы:
A A
Строки:
string
 stri
Действительные числа:
123.456700 123.46 1.234567e+02 1.234567E+02
Адрес как: 0012FF80
12345678901234567890123456789O
Выведено чисел: 16
```

Функция вывода sprintf() является в некотором смысле обратной функции sscanf(): она осуществляет вывод в символьную строку. Функция имеет следующий прототип:

```
int sprintf(char *buffer,
const char *format[, argument, ...]);
```

Она во всем аналогична функции printf(), но осуществляет форматированный вывод не на стандартное устройство вывода, а в символьную строку, указатель на которую задан параметром buffer.

Тема 11.5
Функции файлового ввода и вывода

Перед тем как выполнять операции ввода и вывода в файловый поток, нужно его открыть с помощью функции fopen(). Эта функция имеет следующий прототип:

```
FILE *fopen(const char *filename,
const char *mode);
```

Она открывает файл с именем filename и связывает с ним поток. Функция fopen() возвращает указатель, используемый для

идентификации потока в последующих операциях. Параметр mode является строкой, задающей режим, в котором открывается файл. Он может принимать значения, указанные в табл. 11.8.

Таблица 11.8
Режимы открытия файла

Значение	Описание
R	Файл открывается только для чтения
W	Файл создается для записи. Если файл с этим именем уже существует, он будет перезаписан
A	Режим добавления записей (Append); файл открывается для записи в конец (начиная с EOF) или создается для записи, если он еще не существует
r+	Существующий файл открывается для обновления (считывания и записи)
w+	Создается новый файл для обновления (считывания и записи). Если файл с этим именем уже существует, он будет перезаписан
a+	Файл открывается для добавления (т.е. записывания, начиная с EOF); если файл еще не существует, он создается

Чтобы указать, что данный файл открывается или создается как текстовый, добавьте символ t в строку режима открытия (например, rt, w+t и т.д.). Аналогичным образом, чтобы сообщить, что файл открывается или создается как бинарный, добавьте в строку режима открытия символ b (например, wb, a+b и т.д.). Функция fopen() также позволяет вставить символы t или b между буквой и символом (+) в строке режима открытия (например, строка rt+ эквивалентна строке r+t). Когда файл открывается для обновления, можно вводить и выводить данные в результирующий поток. Однако вывод не может осуществляться непосредственно после ввода, если ему не предшествует вызов функции fseek() или rewind(). В случае успеха fopen() возвращает указатель на открытый поток; в случае ошибки – указатель NULL.

Например:

```
FILE* stream = fopen("Install.dat", "r");
```

Указатель на открытый файловый поток используется во всех последующих функциях работы с потоком.

По завершении работы с потоком он должен быть закрыт. Это осуществляется с помощью функции fclose(), которая имеет следующий прототип:

```
int fclose(FILE *stream);
```

Все буферы, связанные с потоком, освобождаются перед закрытием потока. В случае успеха fclose() возвращает 0; в случае ошибки – EOF. Если Ваша программа не закрывает поток с помощью явного вызова fclose(), то он закрывается автоматически по ее завершению.

Рассмотрим теперь функции, осуществляющие ввод-вывод в файловый поток.

Функция fgetc() имеет следующий прототип:

```
int fgetc(FILE *stream);
```

Она осуществляет ввод символа из файлового потока stream. В случае успеха функция преобразует прочитанный символ в тип int без учета знака. Если делается попытка прочесть конец файла или произошла ошибка, функция возвращает EOF. Как видим, эта функция во всем аналогична функции getc(), за исключением того, что чтение осуществляется из файлового потока. Более того, как мы уже отмечали, на самом деле getc() – это макрос, реализованный с помощью fgetc(). То же самое относится и к следующим функциям: все они имеют уже рассмотренные нами аналоги.

Функция fputc() имеет следующий прототип:

```
int fputc(int c, FILE *stream);
```

Она осуществляет вывод символа в файловый поток и во всем аналогична функции putc().

Функция fgets() имеет следующий прототип:

```
char *fgets(char *s, int n, FILE *stream);
```

Она осуществляет чтение строки символов из файлового потока в строку s. Функция прекращает чтение, если прочитано n - 1 символов или встретился символ перехода на новую строку "\n". Если встретился символ перехода на новую строку, он сохраняется в переменной s. В обоих случаях в переменную s добавляется символ "\0", который является признаком завершения строковой переменной. В случае успеха функция возвращает строку, на которую указывает параметр s. Если делается попытка чтения конца файла или произошла ошибка, она возвращает NULL.

Функция fputs() имеет следующий прототип:

```
int fputs(const char *s, FILE *stream);
```

Она осуществляет вывод строки в файловый поток. Символ перехода на новую строку не добавляется, и завершающий строку нуль-символ в файловый поток не копируется. В случае успеха fputs() возвращает неотрицательное значение. В противном случае она возвращает EOF.

Функция fscanf() имеет следующий прототип:

```
int fscanf(FILE *stream,
const char *format[, address, ...]);
```

Она во всем аналогична функции scanf(), за исключением того, что форматированный ввод осуществляется не со стандартного устройства ввода, а из файлового потока.

Функция fprintf() имеет следующий прототип:

```
int fprintf(FILE *stream,
const char *format[, argument, ...]);
```

Она во всем аналогична функции printf(), но осуществляет форматированный вывод не на стандартное устройство вывода, а в файловый поток.

Функция feof() является на самом деле макросом и позволяет осуществлять проверку на достижение символа конца файла при операциях ввода-вывода. Она имеет следующий прототип:

```
int feof(FILE *stream);
```

Она возвращает ненулевое значение, если был обнаружен конец файла при последней операции ввода в поток stream, и 0, если конец файла еще не достигнут.

Рассмотрим пример файлового ввода и вывода:

```
#include <stdio.h>
int main()
{
 FILE *in, *out;
 if ((in = fopen("C:\\AUTOEXEC.BAT", "rt")) == NULL)
   {
    fprintf(stderr, "Невозможно открыть файл для чтения.\n");
    return 1;
   }
   if ((out = fopen("C:\\AUTOEXEC.BAK", "wt")) == NULL)
   {
    fprintf(stderr, "Невозможно открыть файл для записи.\n");
```

```
    return 1;
  }
  while (!feof(in))
    fputc(fgetc(in), out);
  fclose(in);
  fclose(out);
  fprintf(stderr, "Файл успешно скопирован.\n");
  return 0;
}
```

Две следующие функции предназначены для осуществления неформатированного ввода и вывода в файловые потоки.

Функция fread() имеет следующий прототип:

```
size_t
fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Эта функция считывает из потока stream в буфер, указанный параметром ptr, n блоков данных, каждый из которых содержит size байтов. В случае успеха функция возвращает число прочитанных блоков. Если прочитан конец файла или произошла ошибка, она возвращает число полностью прочитанных блоков или 0.

Функция fwrite() имеет следующий прототип:

```
size_t
fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

Она записывает в выходной поток stream из буфера, указанного параметром ptr, n блоков данных, каждый из которых содержит size байтов. В случае успеха функция возвращает число записанных блоков. В случае ошибки она возвращает число полностью записанных блоков или 0.

Приведем пример, демонстрирующий работу с этими функциями:

```
#include <stdio.h>
#include <string.h>
struct Client
{
 int Num;
 char SurName[27];
 char Name[21];
 char SecName[21];
};
int main(void)
{
 FILE *stream;
```

```
Client AClient, RClient;
//Открываем файл для вывода
if ((stream = fopen("SAMPLE.DAT", "wb")) == NULL)
 {
  printf("Невозможно открыть файл для записи.\n");
  return 1;
 }
AClient.Num = 1;
strcpy(AClient.SurName,"Petrov");
strcpy(AClient.Name, "Petr");
strcpy(AClient.SecName, "Petrovich");
//Запись структуры в файл
fwrite(&AClient, sizeof(AClient), 1, stream);
//Закрываем файл
fclose(stream);
//Открываем файл для чтения
if ((stream = fopen("SAMPLE.DAT", "rb")) == NULL)
 {
  printf("Невозможно открыть файл для чтения.\n");
  return 2;
 }
fread(&RClient, sizeof(RClient), 1, stream);
//Закрываем файл
fclose(stream);
printf("Структура содержит:\n");
printf("Номер = %d Фамилия = %s Имя = %s Отчество = %s",
       RClient.Num, RClient.SurName,
       RClient.Name, RClient.SecName);
return 0;
}
```

В этом примере вначале файл открывается для записи в него значений полей из структуры AClient, затем этот файл открывается для чтения содержащейся в нем информации в структуру RClient. После чего поля этой структуры выводятся на экран. При выполнении программа выводит на экран:

```
Структура содержит:
Номер = 1 Фамилия = Petrov Имя = Petr Отчество = Petrovich
```

Тема 11.6
Функции позиционирования

Когда файл открывается для записи или чтения, с ним связывается структура FILE, определенная в заголовочном файле <stdio.h>. Эта структура связывает с каждым открытым файлом

счетчик положения текущей записи. Сразу после открытия файла его значение равно 0. Каждая операция вызывает приращение значения этого счетчика на число записанных или прочитанных из файла байтов. Функции позиционирования – fseek(), ftell() и rewind() – позволяют изменять или получать значение счетчика, связанного с файлом.

Функция ftell() возвращает текущее значение счетчика, связанного с файлом. Она имеет следующий прототип:

```
long int ftell(FILE *stream);
```

В случае ошибки она возвращает -1L.

Функция fseek() имеет следующий прототип:

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

Эта функция изменяет позиционирование файлового потока stream (изменяя значение указанного счетчика) на offset относительно позиции, определяемой параметром from. Для потоков в текстовом режиме параметр offset должен быть равен 0 или значению, возвращаемому функцией ftell(). Параметр from может принимать следующие значения:

SEEK_SET	(=0)	начало файла;
SEEK_CUR	(=1)	текущая позиция в файле;
SEEK_END	(=2)	конец файла.

Функция возвращает 0, если указатель текущей позиции в файле успешно изменен, и отличное от нуля значение в противном случае.

Функция rewind() имеет следующий прототип:

```
void rewind(FILE *stream);
```

Она устанавливает файловый указатель позиции в начало потока.

Рассмотрим пример, демонстрирующий работу этих функций:

```
#include <stdio.h>
#include <string.h>
struct Client
{
 int Num;
 char SurName[27];
 char Name[21];
 char SecName[21];
};
int main(void)
{
```

```
FILE *stream;
Client AClient, RClient;
long int pos;
//Открываем файл для вывода
if ((stream = fopen("SAMPLE.DAT", "wb")) == NULL)
 {
   printf("Невозможно открыть файл для записи.\n");
   return 1;
 }
AClient.Num = 1;
strcpy(AClient.SurName,"Petrov");
strcpy(AClient.Name, "Petr");
strcpy(AClient.SecName, "Petrovich");
//Запись структуры в файл
fwrite(&AClient, sizeof (AClient), 1, stream);
pos = ftell(stream);
//Выводим позицию файла и длину структуры
printf("Позиция файла = %d длина структуры= %d\n",
 pos, sizeof(AClient));
//Репозиционируем файл
rewind(stream);
//Открываем файл для чтения
if ((stream = fopen("SAMPLE.DAT", "rb")) == NULL)
 {
   printf("Невозможно открыть файл для чтения.\n");
   return 2;
 }
fread(&RClient, sizeof (RClient), 1, stream);
//Закрываем файл
fclose(stream);
printf("Структура содержит:\n");
printf("Номер = %d Фамилия = %s Имя = %s Отчество = %s",
      RClient.Num, RClient.SurName,
      RClient.Name, RClient.SecName);
return 0;
}
```

Этот пример представляет собой модификацию предыдущего примера. Здесь файл открывается лишь однажды, и после записи в него структуры на экран выводится значение указателя позиции и длины структуры. При выполнении программа выводит на экран:

```
Позиция файла = 73 длина структуры= 73
Структура содержит:
Номер = 1 Фамилия = Petrov Имя = Petr Отчество = Petrovich
```

В заключение заметим, что библиотека стандартных функций ввода-вывода содержит много других, не рассмотренных здесь функций.

Практикум
«Функции ввода-вывода»

Упражнение 11.1
Ввод-вывод символов

В предлагаемом занятии нам предстоит посредством символов псевдографики вывести на консоль прямоугольник.

В начале программы определим константы, соответствующие конкретным символам псевдографики для отображения той или иной части прямоугольника. Таким образом, в блоке директив define у нас будут объявлены левый верхний угол (LEFT_TOP), правый верхний угол (RIGHT_TOP), горизонтальная (HORIZ) и вертикальная (VERT) линии, а также левый и правый нижние (LEFT_BOT и RIGHT_BOT) углы соответственно.

Далее построчно начинаем выводить определенные символы на экран с помощью функции putchar():

```
#include <stdio.h>

#define LEFT_TOP  0xDA
#define RIGHT_TOP 0xBF
#define HORIZ     0xC4
#define VERT      0xB3
#define LEFT_BOT  0xC0
#define RIGHT_BOT 0xD9

int main(void)
{
  char i, j;
  putchar(LEFT_TOP);
  for (i=0; i<10; i++)
    putchar(HORIZ);
  putchar(RIGHT_TOP);
  putchar('\n');
  for (i=0; i<4; i++)
  {
    putchar(VERT);
    for (j=0; j<10; j++)
      putchar(' ');
    putchar(VERT);
    putchar('\n');
  }
  putchar(LEFT_BOT);
  for (i=0; i<10; i++)
    putchar(HORIZ);
```

```
    putchar(RIGHT_BOT);
    putchar('\n');
    return 0;
}
```

Упражнение 11.2
Ввод-вывод строк

Модифицируйте приведенный ниже пример таким образом, чтобы строка символов string не инициализировалась при объявлении, а заполнялась введенным пользователем с клавиатуры значением. Проверьте результат путем вывода строки на экран.

```
#include <stdio.h>
int main(void)
{
  char string[] = "Это пример выводимой строки\n";
  puts(string);
  return 0;
}
```

Упражнение 11.3
Использование функции scanf()

Пусть имеется некоторый массив структур entry типа Entry_struct, содержащий информацию о работнике – имя, возраст, зарплату. Необходимо в программе заполнить элементы этого массива, а затем сгенерировать в виде таблицы отчет обо всех работниках.

```
#include <stdio.h>
int main(void)
{
  char label[20];
  char name[20];
  int entries = 0;
  int loop, age;
  double salary;
  struct Entry_struct
  {
    char  name[20];
    int   age;
    float salary;
  } entry[20];
  printf("\n\nВведите метку (до 20 символов): ");
```

```
    scanf("%20s", label);
    fflush(stdin);
    printf("Сколько будет записей? (до 20) ");
    scanf("%d", &entries);
    fflush(stdin);
    for (loop=0;loop<entries;++loop)
    {
      printf("Entry %d\n", loop);
      printf(" Имя      : ");
      scanf("%[A-Za-z]", entry[loop].name);
      fflush(stdin);
      printf(" Возраст  : ");
      scanf("%d", &entry[loop].age);
      fflush(stdin);
      printf(" Зарплата : ");
      scanf("%f", &entry[loop].salary);
      fflush(stdin);
    }
    printf("\nВведите свое имя, возраст и зарплату\n");
    scanf("%20s %d %lf", name, &age, &salary);
    printf("\n\nTable %s\n",label);
    printf("Запросил: %s  возраст %d  $%15.2lf\n", name,
        ↳ age, salary);
    printf("-------------------------------------------------\n");
    for (loop=0;loop<entries;++loop)
      printf("%4d | %-20s | %5d | %15.2lf\n",
        loop + 1,
        entry[loop].name,
        entry[loop].age,
        entry[loop].salary);
    printf("-------------------------------------------------\n");
    return 0;
}
```

Упражнение 11.4
Применение функции sscanf()

Проанализируйте пример использования функции sscanf().

Определим символьный двумерный массив temp, который будет содержать 4 строки по 80 символов. С помощью функции sprintf() заполним этот массив в цикле именами из массива names и произвольными данными о возрасте и зарплате.

Теперь (опять-таки в цикле) будем перебирать строковые элементы массива temp и с помощью функции sscanf() разносить их в переменные для вывода на печать в виде таблицы.

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

char *names[4] = {"Иван", "Петр", "Михаил", "Василий"};

#define NUMITEMS 4

int main(void)
{
  int  loop;
  char temp[4][80];
  char name[20];
  int  age;
  long salary;

  for (loop=0; loop < NUMITEMS; ++loop)
    sprintf(temp[loop], "%s %d %ld", names[loop],
        ↳ random(10) + 20, random(5000) + 27500L);
  printf("%4s | %-20s | %5s | %15s\n", "#", "Name", "Age",
      ↳ "Salary");
  printf("  ------------------------------------------------\n");
  for (loop=0; loop < NUMITEMS; ++loop)
  {
    sscanf(temp[loop],"%s %d %ld", &name, &age, &salary);
    printf("%4d | %-20s | %5d | %15ld\n", loop + 1, name,
        ↳ age, salary);
  }
  return 0;
}
```

Упражнение 11.5
Файловый ввод-вывод

Проанализируйте методику применения файлового ввода-вывода на примере резервного копирования.

Пусть требуется создать копию исходного текста программы FOPEN.CPP в некотором файле FOPEN.BAK. Свяжем открываемый существующий файл с указателем in, а создаваемый резервный файл – с указателем out. В цикле while производим побайтное чтение-запись из файла-источника в файл-приемник, пока не достигнем последнего символа файла-источника.

В завершение операции чтения-записи необходимо закрыть оба файла.

```
#include <stdio.h>

int main(void)
{
  FILE *in, *out;
  if ((in = fopen("FOPEN.CPP", "rt")) == NULL)
```

```
  {
    fprintf(stderr, " Невозможно открыть файл для чтения.\n");
    return 1;
  }
  if ((out = fopen("FOPEN.BAK", "w+t")) == NULL)
  {
    fprintf(stderr, " Невозможно открыть файл для записи.\n");
    return 1;
  }
  while (!feof(in))
    fputc(fgetc(in), out);
  fclose(in);
  fclose(out);
  return 0;
}
```

Упражнение 11.6
Позиционирование в файле

Проанализируйте пример использования функции позиционирования fseek(), в котором определяется длина файла. Вначале создадим программно файл MYFILE.TXT и поместим в него строку "Это тестовый пример.". В функции filesize() проведем позиционирование на конец файла и с помощью функции ftell() определим номер символа, на который указывает счетчик.

```
#include <stdio.h>
long filesize(FILE *stream);
int main(void)
{
  FILE *stream;
  stream = fopen("MYFILE.TXT", "w+");
  fprintf(stream, "Это тестовый пример.");
  printf("Размер файла MYFILE.TXT %ld байт\n", filesize(stream));
  fclose(stream);
  return 0;
}
long filesize(FILE *stream)
{
  long curpos, length;
  curpos = ftell(stream);
  fseek(stream, 0L, SEEK_END);
  length = ftell(stream);
  fseek(stream, curpos, SEEK_SET);
  return length;
}
```

Раздел 12
Объектно-ориентированное программирование

Тема 12.1
Принципы объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) – это новый подход к программированию. По мере развития вычислительной техники и усложнения решаемых задач возникали разные модели (или, как любят говорить зарубежные авторы, парадигмы) программирования. Первые компиляторы (к ним, прежде всего, относится компилятор языка FORTRAN) поддерживали процедурную модель программирования, в основе которой лежит использование функций. Используя процедурную модель программирования, программисты могли писать программы до нескольких тысяч строк длиной. Следующий этап развития связан с переходом к структурной модели программирования (реализованной в компиляторах языков ALGOL, Pascal и C). Сутью структурного программирования является представление программы в виде совокупности взаимосвязанных процедур (или блоков) и тех данных, которыми эти процедуры (или блоки) оперируют. При этом широко используются программные блоки и допускается только минимальное использование операторов GOTO. Сопровождение таких программ намного проще, чем процедурно-ориентированных. Используя структурное программирование, средний программист может создавать и сопровождать программы до нескольких десятков тысяч строк длиной.

Для написания более сложных задач понадобился новый подход к программированию, который и был реализован в модели ООП. Модель ООП основана на нескольких основополагающих концепциях.

Абстракция данных – это возможность определять новые типы данных, с которыми можно работать почти так же, как и с основными типами данных. Такие типы часто называются абстрактными типами данных, хотя термин «типы данных, определяемые пользователем» является более точным.

Инкапсуляция – это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает то и другое от постороннего вмешательства.

Для реализации этих двух основополагающих концепций в языке C++ используются классы. Термином *класс* определяется тип объектов. При этом каждый представитель (или экземпляр) класса называется *объектом*. Каждый объект всегда имеет свое, уникальное *состояние*, определяемое текущими значениями его *данных-членов (элементов-данных)*. Функциональное назначение класса определяется возможными действиями над объектами класса, которые задаются его *функциями-членами (функциями-элементами,* или *методами)*. Причем термин «метод» чаще используется в литературе, посвященной языку Object Pascal. В каждом классе распределяется память для хранения данных и устанавливаются допустимые операции для каждого объекта данных этого типа. Создание объектов данного класса производится специальной функцией-членом, которая называется *конструктором*, а уничтожение – другой специальной функцией-членом, которая называется *деструктором*. Класс позволяет делать недоступными внутренние данные, представляя их как открытые (public), закрытые (private) и защищенные (protected). Класс устанавливает четко определенный интерфейс для взаимодействия объектов этого типа с остальным миром. Закрытые коды или данные доступны только внутри этого объекта. С другой стороны, открытые коды и данные, несмотря на то что они заданы внутри объекта, доступны для всех частей программы. Открытая часть объекта как раз и используется для создания интерфейса объекта с остальным миром. Полученными объектами можно управлять при помощи *сообщений (*или *запросов),* которые представляют собой просто вызовы функций-членов. Мы будем пользоваться термином «запрос», чтобы не путать это понятие с сообщениями операционной системы Windows.

Наследование – это процесс, посредством которого один объект может приобретать свойства другого. Это означает, что в ООП на основе уже существующих классов можно строить *производные классы*. При этом производный класс (называемый так-

же *классом-потомком*) *наследует* элементы-данные и функции-члены от своих *родительских классов* (*классов-предков*), добавляя к ним свои, которые позволяют ему реализовать черты, характерные только для него. Защищенные данные-члены и функции-члены родительского класса доступны из производного класса. Кроме того, в производном классе наследуемые функции могут быть переопределены. Таким образом, можно построить целую *иерархию классов*, связанных между собой отношением родитель – потомок. Термин *базовый класс* используется как синоним родительскому классу в иерархии классов. Если объект наследует свои атрибуты (данные-члены и функции-члены) от одного родительского класса, говорят об *одиночном* (или *простом*) *наследовании*. Если объект наследует атрибуты от нескольких родителей, говорят о *множественном наследовании*. Наследование позволяет значительно сократить определение класса-потомка благодаря тому, что классы-потомки являются расширениями родительских классов.

Полиморфизм (от греческого «polymorphos» – множественность форм) – это свойство кода вести себя по-разному в зависимости от ситуации, возникающей в момент выполнения. Полиморфизм – это не столько характеристика объектов, сколько характеристика функций-членов класса и проявляется, в частности, в возможности использования одного имени функции-члена для функций, имеющих различные типы аргументов, если выполняемые функцией действия определяются типом переданных ей аргументов. Это называется *перегрузкой функции*. Полиморфизм может применяться и к операциям, то есть выполняемые операцией действия также могут зависеть от типа данных (операндов). Такой тип полиморфизма называется *перегрузкой операции*.

В более общем смысле полиморфизм – это способность объектов различных классов отвечать на запрос функции сообразно типу своего класса. Другими словами, полиморфизм – это способность указателей (или ссылок) на базовый класс принимать различные формы при использовании их для вызовов *виртуальных функций*. Такая возможность в C++ является результатом *позднего связывания*. При позднем связывании адреса вызываемых функций-членов определяются динамически во время выполнения программы, а не статически во время компиляции, как в традиционных языках, в которых применяется *раннее связывание*. Позднее связывание выполняется только для виртуальных функций.

Теперь, когда мы рассмотрели основные концепции ООП, мы можем остановиться на методологии ООП. В этой методологии решаемая задача представляется не в виде алгоритмической модели, а в виде совокупности объектов различных классов, которые обмениваются запросами. Объект, получивший запрос, отвечает на него посредством вызова соответствующей функции-члена. Нахождение общности между типами объектов задачи – далеко не простой процесс. Он осуществляется на стадии *объектно-ориентированного проектирования*. На этой стадии вначале рассматривается вопрос о применимости ООП к решаемой задаче. При этом для принятия решения о применимости ООП решающую роль играет та степень общности между классами, которую можно использовать, применяя механизмы наследования и виртуальные функции. В некоторых задачах, таких как разработка графического интерфейса приложения, разработка приложений баз данных и компьютерная графика, простор для ООП поистине безграничен. Кроме того, методы ООП позволяют скрыть детали низкоуровневых сетевых протоколов. Поэтому они широко применяются при программировании приложений, работающих в сетях (как локальных, так и глобальных, например в Internet). В случае принятия решения о применимости ООП на следующем этапе проектирования разрабатываются классы как строительные блоки для других типов и изучаются возможности выделения в классах тех свойств, которые могут быть переданы базовому классу. Затем проектируются те запросы, которыми будут обмениваться объекты, и осуществляется реализация разработанных объектов и запросов.

Пусть Вас не смущает, если не все из этих понятий сразу станут ясны. Возвратитесь к данному разделу после прочтения нескольких последующих разделов.

Тема 12.2
Классы

Понятие класса является, пожалуй, наиболее важным в языке C++. Синтаксис описания класса похож на синтаксис описания структуры. Вот его основная форма:

```
class <имя_класса>
{
//закрытые функции-члены и данные-члены класса
public:
```

```
//открытые функции-члены и данные-члены класса
} <список_объектов>;
```

В описании класса <список_объектов> не является обязательным. Вы можете объявить объекты класса позже, по мере необходимости. Так обычно и поступают. Хотя <имя_класса> также необязательно, его обычно указывают. Дело в том, что <имя_класса> становится новым именем типа данных, которое используется для объявления объектов этого класса.

Функции и переменные, объявленные внутри объявления класса, становятся членами этого класса. Переменные, объявленные внутри объявления класса, называются *данными-членами* этого класса; функции, объявленные внутри объявления класса, называются *функциями-членами* класса. По умолчанию, все функции и переменные, объявленные в классе, становятся закрытыми для этого класса. Это означает, что они доступны только для других членов этого класса. Для объявления открытых членов класса используется ключевое слово public, за которым следует двоеточие. Все функции и переменные, объявленные после слова public, доступны и для других членов класса, и для любой другой части программы, в которой содержится класс.

Вот пример объявления класса:

```
class AnyClass
 {
   //закрытый элемент класса
   int a;
 public:
   int get_a();
   void set_a(int num);
};
```

Хотя функции get_a() и set_a() и объявлены в классе AnyClass, они еще не определены. Для определения функции-члена нужно связать имя класса, частью которого является функция-член, с именем функции. Это достигается путем написания имени функции вслед за именем класса с двумя двоеточиями. Операцию, записываемую в виде двух двоеточий, называют *операцией расширения области видимости*. Для задания функции-члена используется следующая общая форма:

```
<Тип> <имя_класса>::<имя_функций>
                        ↳(<список_параметров>)
{
//тело функции
}
```

Ниже приведены примеры определения функций-членов get_a() и set_a():

```
void AnyClass::get_a()
{
 return a;
}
int AnyClass::set_a(int num)
{
 a = num;
}
```

Объявление класса AnyClass не приводит к созданию объектов типа AnyClass. Чтобы создать объект соответствующего класса, нужно просто использовать имя класса как спецификатор типа данных. Например:

```
AnyClass ob1, ob2;
```

После того как объект класса создан, можно обращаться к открытым членам класса, используя операцию «точка», так же как к полям структуры.

Например:

```
ob1.set_a(10);
ob2.set_a(37);
```

Эти операторы устанавливают значения переменной a в объектах ob1 и ob2. Каждый объект содержит собственную копию всех данных, объявленных в классе. Это значит, что значение переменной a в ob1 отлично от значения этой переменной в ob2.

Доступ к элементам данных класса можно получить и с помощью указателя на объект этого класса. Следующий пример это иллюстрирует.

```
class Coord
{
 public:
   int x,y;
   void SetCoord(int _x, int _y);
};
void
Coord::SetCoord(int _x, int _y)
{
 x = _x;
 y = _y;
}
int main()
```

```
{
 Coord pt;
 Coord *ptPtr = &pt; //указатель на объект
 //...
 pt.x = 0;              //объект.член_класса
 ptPtr->y = 0;          //указатель->член_класса
 ptPtr->SetCoord(10,20);
 //...
 return 0;
}
```

Среди программистов часто возникает спор о том, какой метод доступа к членам класса – через оператор "." или через оператор "->" – работает быстрее. Внесем ясность в этот вопрос. Оператор выбора члена "." работает в точности так же, как оператор "->", за исключением того, что имени объекта предшествует неявно сгенерированный компилятором оператор адреса "&". Таким образом, инструкцию

```
ObjName.FuncName();
```

компилятор трактует, как

```
(&ObjName)->FuncName();
```

Имена элементов класса могут также использоваться с именем класса, за которым следует операция разрешения видимости (двойное двоеточие), то есть вызов элемента имеет вид:

```
<имя_класса>::<имя_члена>
```

Подробнее эту форму вызова элементов класса мы рассмотрим позднее.

Язык C++ накладывает определенные ограничения на данные-члены класса:

- данные-члены не могут определяться с модификаторами auto, extern или register;
- данным-членом класса не может быть объект этого же класса (однако данным-членом может быть *указатель* или *ссылка* на объект этого класса, или сам объект другого класса).

Имена всех элементов класса (данных и функций) имеют своей областью действия этот класс. Это означает, что функции-члены могут обращаться к любому элементу класса просто по имени.

Класс должен быть объявлен до использования его членов. Однако иногда в классе должен быть объявлен указатель или ссылка на объект другого класса еще до того, как он определен. В этом случае приходится прибегать к неполному объявлению класса, которое имеет вид:

```
class <имя_класса>;
```

Рассмотрим следующий пример.

```
//Неполное объявление класса
class PrevDecl;

class AnyClass
{
  int x;
  PrevDecl* obPtr;
 public:
  AnyClass(int _x){x = _x;}
};

int main()
{
 // тело основного блока
 return 0;
}

//Полное объявление класса
class PrevDecl
{
   int a;
 public:
   PrevDecl();
};
```

Нельзя создать объект не полностью определенного класса. Попытка создания такого класса приводит к ошибке компиляции.

Заметим, что определение класса напоминает определение структуры, за исключением того, что определение класса:

- обычно содержит один или несколько *спецификаторов доступа*, задаваемых с помощью ключевых слов public, protected или private;
- вместо ключевого слова struct могут применяться class или union;
- обычно включает в себя наряду с элементами данных функции-члены;
- обычно содержит некоторые специальные функции-члены, такие как *конструктор* и *деструктор*.

Ниже приведены примеры определения классов с использованием ключевых слов struct и union.

```
struct Point
{
 private:
  int x;
  int y;
```

```
  public:
    int GetX();
    int GetY();
    void SetX(int _x);
    void SetY(int _y);
};
union Bits
{
 Bits(unsigned int n);
 void ShowBits();
 unsigned int num;
 unsigned char c[sizeof(unsigned int)];
};
```

Спецификатор доступа применяется ко всем элементам класса, следующим за ним, пока не встретится другой спецификатор доступа или не закончится определение класса. Табл. 12.1 описывает три спецификатора доступа.

Таблица 12.1
Спецификаторы доступа к классу

Спецификатор	Описание
private:	Данные-члены и функции-члены доступны только для функций-членов этого класса
protected:	Данные-члены и функции-члены доступны для функций-членов данного класса и классов, производных от него
public:	Данные-члены и функции-члены класса доступны для функций-членов этого класса и других функций программы, в которой имеется представитель класса

В C++ *класс, структура* и *объединение* рассматриваются как типы классов. Структура и класс подобны друг другу, за исключением доступа по умолчанию: в структуре элементы имеют по умолчанию доступ public, в то время как в классе private. Объединение, как и структура, по умолчанию предоставляет доступ public.

В приведенной ниже табл. 12.2 перечислены эти отличия.

Структуры и объединения могут иметь конструкторы и деструкторы. Вместе с тем имеется несколько ограничений на использование объединений в качестве классов. Во-первых, они не

могут наследовать какой бы то ни было класс, и они сами не могут использоваться в качестве базового класса для любого другого типа. Короче говоря, объединения не поддерживают иерархию классов. Объединения не могут иметь членов с атрибутом static. Они также не должны содержать объектов с конструктором и деструктором.

Таблица 12.2

Различие между классом, структурой и объединением

Различие	Классы	Структуры	Объединения
Ключевое слово:	class	struct	union
Доступ по умолчанию:	private	public	public
Перекрытие данных:	Нет	Нет	Да

Рассмотрим пример использования объединения в качестве класса:

```
#include <iostream.h>
union Bits
{
 Bits(unsigned int n);
 void ShowBits();
 unsigned int num;
 unsigned char c[sizeof(unsigned int)];
};
Bits::Bits(unsigned int n)
{
 num = n;
}
void
Bits::ShowBits()
{
 int i, j;
 for (j=sizeof(unsigned int)-1; j>=0; j--)
 {
  cout << "Двоичное представление байта "
   << j << ":";
  for (i=128; i; i>>=1)
  {
      if (i & c[j]) cout << "1";
```

```
        else cout << "0";
    }
    cout << "\n";
  }
}
int main()
{
 Bits ob(2000);
 ob.ShowBits();
 return 0;
}
```

В этом примере осуществляется побайтовый вывод переданного объекту беззнакового целочисленного значения в двоичном виде.

Тема 12.3
Конструкторы и деструкторы. Список инициализации элементов

Создавая некоторый объект, его необходимо проинициализировать. Для этой цели C++ предоставляет функцию-член, которая называется *конструктором*. Конструктор класса вызывается всякий раз, когда создается объект его класса. Конструктор имеет то же имя, что и класс, членом которого он является, и не имеет возвращаемого значения. Например:

```
#include <iostream.h>
class AnyClass
{
  int var;
public:
  AnyClass(); //Конструктор
  void Show();
};
AnyClass:: AnyClass()
{
 cout << 'В конструкторе\n';
 var = 0;
}
void
AnyClass::Show()
{
 cout << var;
}
```

```
int main()
{
 AnyClass ob;
 ob.Show();
return 0;
}
```

В этом примере конструктор класса AnyClass выводит сообщение на экран и инициализирует значение закрытой переменной класса var.

Заметим, что программист не должен писать код, вызывающий конструктор класса. Всю необходимую работу выполняет компилятор. Конструктор вызывается тогда, когда создается объект его класса. Объект, в свою очередь, создается при выполнении оператора, объявляющего этот объект. Таким образом, в C++ оператор объявления переменной является выполнимым оператором.

Для глобальных объектов конструктор вызывается тогда, когда начинается выполнение программы. Для локальных объектов конструктор вызывается всякий раз при выполнении оператора, объявляющего переменную.

Функцией-членом, выполняющей действия, обратные конструктору, является *деструктор*. Эта функция-член вызывается при удалении объекта. Деструктор обычно выполняет работу по освобождению памяти, занятой объектом. Он имеет то же имя, что и класс, которому он принадлежит, с предшествующим символом "~" и не имеет возвращаемого значения.

Рассмотрим пример класса, содержащего деструктор:

```
#include <iostream.h>
class AnyClass
{
  int var;
 public:
  AnyClass();  //Конструктор
  ~AnyClass(); //Деструктор
  void Show();
};
AnyClass:: AnyClass()
{
 cout << "Мы в конструкторе\n";
 var = 0;
}
AnyClass::~ AnyClass()
{
 cout << "Мы в деструкторе\n";
}
```

```
void
AnyClass::Show()
{
 cout << var << "\n";
}
int main()
{
 AnyClass  ob;
 ob.Show();
 return 0;
}
```

Деструктор класса вызывается в момент удаления объекта. Это означает, что для глобальных объектов он вызывается при завершении программы, а для локальных – когда они выходят из области видимости. Заметим, что невозможно получить указатели на конструктор и деструктор.

Обычно конструктор содержит параметры, которые позволяют при построении объекта задать ему некоторые аргументы. В рассмотренном выше примере конструктор инициализировал закрытую переменную var класса AnyClass значением 0. Если нужно проинициализировать переменные класса, используется конструктор с параметрами. Модифицируем предыдущий пример:

```
#include <iostream.h>

class AnyClass
{
  int a, b;
 public:
  //Конструктор с параметрами
  AnyClass(int x, int y);
  //Деструктор
  ~AnyClass();void Show();
};
AnyClass:: AnyClass(int x, int y)
{
 cout << "Мы в конструкторе\n";
 a = x;
 b = y;
}
AnyClass::~ AnyClass()
{
 cout << "Мы в деструкторе\n";
}
```

```
void
AnyClass::Show()
{
 cout << a << ' ' << b << "\n";
}
int main()
{
 AnyClass  ob(3,7);
 ob.Show();
 return 0;
}
```

Здесь значение, переданное в конструктор при объявлении объекта ob, используется для инициализации закрытых переменных a и b этого объекта.

Фактически, синтаксис передачи аргумента конструктору с параметрами является сокращенной формой записи следующего выражения:

```
AnyClass ob = AnyClass(3,7);
```

Однако практически всегда используется сокращенная форма синтаксиса, приведенная в примере.

В отличие от конструктора, деструктор не может иметь параметров. Понятно, почему это сделано: незачем передавать аргументы удаляемому объекту.

Приведем правила, которые существуют для конструкторов:

- для конструктора не указывается тип возвращаемого значения;
- конструктор не может возвращать значение;
- конструктор не наследуется;
- конструктор не может быть объявлен с модификатором const, volatile, static или virtual.

Если в классе не определен конструктор, компилятор генерирует *конструктор по умолчанию*, не имеющий параметров.

Для деструкторов существуют следующие правила:

- деструктор не может иметь параметров;
- деструктор не может возвращать значение;
- деструктор не наследуется;
- класс не может иметь более одного деструктора;
- деструктор не может быть объявлен с модификатором const, volatile, static или virtual.

Если в классе не определен деструктор, компилятор генерирует *деструктор по умолчанию.*

Подчеркнем еще раз: конструкторы и деструкторы в C++ вызываются автоматически, что гарантирует правильное создание и удаление объектов класса.

Не все приведенные выше правила станут Вам ясны в данный момент. Вернитесь к ним после прочтения нескольких последующих разделов.

Обычно данные-члены класса инициализируются в *теле* конструктора, однако существует и другой способ инициализации – с помощью списка инициализации элементов. *Список инициализации* элементов отделяется двоеточием от заголовка определения функции и содержит данные-члены и базовые классы, разделенные запятыми. Для каждого элемента в круглых скобках непосредственно за ним указывается один или несколько параметров, используемых при инициализации. В следующем примере для инициализации класса используется список инициализации элементов.

```
class AnyClass()
{
  int a, b;
 public:
  AnyClass(int x, int y);
};
//Конструктор использует список инициализации
AnyClass::AnyClass(int x, int y): a(x), b(y)
{
  ...
}
```

Хотя выполнение инициализации в теле конструктора или с помощью списка инициализации – дело вкуса программиста, список инициализации является единственным методом инициализации данных-констант и ссылок. Если членом класса является объект, конструктор которого требует задания значений одного или нескольких параметров, то единственно возможным способом его инициализации также является список инициализации.

Тема 12.4
Конструкторы по умолчанию и конструкторы копирования

C++ определяет два специальных вида конструкторов: конструктор по умолчанию, о котором мы упоминали выше, и конструктор копирования. *Конструктор по умолчанию* не имеет пара-

метров (или все его параметры должны иметь значения по умолчанию) и вызывается при создании объекта, которому не заданы аргументы. Следует избегать двусмысленности при вызове конструкторов. В приведенном ниже примере два конструктора по умолчанию являются двусмысленными:

```
class T
{
 public:
//Конструктор по умолчанию
  T();
//Конструктор с одним параметром;
//может быть использован как конструктор по умолчанию
  T(int I=0);
};
int main()
{
 T ob1(10);//Использует T::T(int)
 T ob2;     //Не верно; неоднозначность
            //вызова T::T() или  T::T(int =0)
 return 0;
}
```

В данном случае, чтобы устранить неоднозначность, достаточно удалить из объявления класса конструктор по умолчанию.

Конструктор копирования (или *конструктор копии*) создает объект класса, копируя при этом данные из уже существующего объекта данного класса. В связи с этим он имеет в качестве единственного параметра константную ссылку на объект класса (const T&) или просто ссылку на объект класса (T&). Использование первого предпочтительнее, так как последний не позволяет копировать константные объекты. Приведем пример использования конструктора копирования:

```
class Coord
{
  int x, y;
 public:
    //Конструктор копирования
  Coord(const Coord& src);
};
Coord::Coord(const Coord& src)
{
 x = src.x;
 y = src.y;
}
```

```
int main()
{
 Coord ob1(2,9);
 Coord ob2 = ob1;
 Coord ob3(ob1);
return 0;
}
```

Ссылка передается всякий раз, когда новый объект инициализируется значениями существующего. Если вы не предусмотрели конструктор копирования, компилятор генерирует *конструктор копирования по умолчанию*. В C++ различают поверхностное и глубинное копирование данных. При *поверхностном копировании* происходит передача только адреса от одной переменной к другой, в результате чего оба объекта указывают на одни и те же ячейки памяти. В случае *глубинного копирования* происходит действительное копирование значений всех переменных из одной области памяти в другую. Конструктор копирования по умолчанию, реализуемый компилятором, создает буквальную (или побитную) копию объекта, то есть осуществляет поверхностное копирование. Полученная копия объекта скорее всего будет непригодной, если она содержит указатели или ссылки. Действительно, если эти указатели или ссылки ссылаются на динамически распределенные объекты или на объекты, встроенные в копируемый объект, они будут недоступны в созданной копии объекта. Поэтому для классов, содержащих указатели и ссылки, следует включать в определение класса конструктор копирования, который будет осуществлять глубинное копирование, не полагаясь на создаваемый компилятором конструктор копирования по умолчанию. В этом конструкторе, как правило, выполняется копирование динамических структур данных, на которые указывают или на которые ссылаются члены класса. Класс *должен* содержать конструктор копирования, если он перегружает оператор присваивания.

Если в классе не определен конструктор, компилятор пытается сгенерировать собственный конструктор по умолчанию и, если нужно, собственный конструктор копирования. Эти сгенерированные компилятором конструкторы рассматриваются как открытые функции-члены. Подчеркнем, что компилятор генерирует эти конструкторы, если в классе не определен *никакой другой* конструктор.

Сгенерированный компилятором конструктор по умолчанию создает объект и вызывает конструкторы по умолчанию для базовых классов и членов-данных, но не предпринимает никаких дру-

гих действий. Конструкторы базового класса и членов-данных вызываются, только если они существуют, доступны и являются недвусмысленными. Когда конструируется объект производного класса, конструирование начинается с объектов его базовых классов и продвигается вниз по иерархии классов, пока не будет создан заданный объект. Подробнее этот вопрос будет рассмотрен в следующем разделе.

Сгенерированный компилятором конструктор копирования создает новый объект и выполняет почленное копирование содержимого исходного объекта. Если существуют конструкторы базового класса или члена-класса, они вызываются; иначе выполняется побитное копирование. Если все базовые классы и классы членов-данных класса имеют конструкторы копирования, которые воспринимают const-аргумент, то сгенерированный компилятором конструктор копирования воспринимает аргумент типа const T&. В противном случае он воспринимает аргумент типа T& (без const).

Тема 12.5
Указатель this

Каждый объект в C++ содержит специальный указатель с именем this, который автоматически создается самим компилятором и указывает на данный объект. Типом this является T*, где T – тип класса данного объекта. Поскольку указатель this определен в классе, область его действия – класс, в котором он определен. Фактически this является скрытым параметром класса, добавляемым самим компилятором к его определению. При вызове обычной функции-члена класса ей передается указатель this так, как если бы он был первым аргументом. Таким образом, вызов функции-члена

```
ObjName.FuncName(par1, par2);
```

компилятор трактует так:

```
ObjName.FuncName(&ObjName, par1, par2);
```

Но, поскольку аргументы помещаются в стек справа налево, указатель this помещается в него последним. В теле функции-члена адрес объекта доступен как указатель this. Дружественным функциям и статическим функциям-членам класса (о них мы будем говорить позже) указатель this не передается. Нижеследующий пример демонстрирует использование этого указателя:

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

```
#include <string.h>
class T
{
 public:
  T(char*);
  void Greeting();
  char item[20];
};
T::T(char* name)
{
 strcpy(item, name);
 Greeting();                    //Все три
 this->Greeting();              //оператора
 (*this).Greeting();            //эквивалентны
}
void
T::Greeting()
{
 //Оба нижеследующих оператора эквивалентеы
 cout <<      "Hello, " << item << "\n";
 cout <<"Hello, " << this->item << "\n";
}
int main()
{
 T ob("dear.");
 return 0;
}
```

Как можно видеть, внутри конструктора класса и функции-члена Greeting() обращения к данным-членам класса и функциям-членам могут осуществляться как непосредственно по имени, так и с помощью указателя this. Поэтому на практике такое употребление указателя this встречается крайне редко. В основном указатель this используется для возврата указателя (в форме: return this;) или ссылки (в форме: return *this;) на соответствующий объект. Этот указатель находит широкое применение при перегрузке операторов.

Тема 12.6
Встраиваемые (inline) функции

В C++ можно задать функцию, которая фактически не вызывается, а ее тело встраивается в программу в месте ее вызова. Она действует почти так же, как макроопределение с параметрами. По сравнению с обычными функциями встраиваемые (inline) функции

обладают тем преимуществом, что их вызов не связан с передачей аргументов и возвратом результатов через стек и, следовательно, они выполняются быстрее обычных. Недостатком встраиваемых функций является то, что если они слишком большие и вызываются слишком часто, объем программы сильно возрастает. Из-за этого применение встраиваемых функций обычно ограничивается только очень простыми функциями.

Объявление встраиваемой функции осуществляется с помощью спецификатора inline, который вписывается перед определением функции.

Следует иметь в виду, что спецификатор inline только формулирует требование компилятору сформировать встроенную функцию. Если компилятор не в состоянии выполнить это требование, функция компилируется как обычная.

Компилятор не может сгенерировать функцию как встраиваемую, если она:

- содержит оператор цикла (for, while, do-while);
- содержит оператор switch или goto;
- содержит статическую переменную (static);
- является рекурсивной;
- имеет возвращаемый тип, отличный от void, и не содержит оператора return;
- содержит встроенный код ассемблера.

Компилятор может налагать и другие ограничения на использование inline-функций. Уточните их в описании конкретного компилятора. Ниже приведен пример использования встраиваемой функции:

```
#include <iostream.h>
inline int even(int x)
{
 return !(x%2);
}
int main()
{
  int n;
  cin >> n;
  if (even(n))
    cout << n << "является четным\n";
  else
    cout << n << " является нечетным \n";
 return 0;
}
```

В этом примере программа читает введенное целое число и сообщает, является ли оно четным. Функция even() объявлена как встраиваемая. Это означает, что оператор

```
if (even(n))
  cout << n << "является четным\n";
```

эквивалентен следующему:

```
if (!(n%2))
  cout << n << "является четным\n";
```

Важно отметить, что встраиваемая функция должна быть не только объявлена, но и *определена* до ее первого вызова. Поэтому определение inline-функции обычно размещается в заголовочном файле. В приведенном выше примере функция even() была определена до функции main(), в которой она используется.

Встроенными могут быть объявлены не только обычные функции, но и функции-члены. Для этого достаточно включить ее *определение* в объявление класса (в этом случае ключевое слово inline больше не нужно) или перед определением функции вставить ключевое слово inline.

Следующий пример демонстрирует использование встроенных функций-членов.

```
class Point
{
  int x;
  int y;
public:
  int GetX() {return x;}   //inline-функция
  int GetY() {return y;}   //inline-функция
  void SetX(int _x){x =_x;}//inline-функция
  void SetY(int _y){y =_y;}//inline-функция
};
```

Можно определить встроенную функцию-член и вне тела класса, предварив ее определение ключевым словом inline:

```
class Point
{
  int x;
  int y;
public:
  int GetX();
  int GetY();
  void SetX(int);
  void SetY(int);
};
```

```
inline int
Point::GetX()
{
 return x;
}
inline int
Point::GetY()
{
 return y;
}
void
Point::SetX(int _x)
{
 x = _x;
}
void
Point::SetY(int _y)
{
 y = _y;
}
```

Тема 12.7
Статические члены класса

Члены класса могут быть объявлены с модификатором static. Статический член класса может рассматриваться как глобальная переменная или функция, доступная только в пределах области класса.

Данное-член класса, определенное с модификатором static, разделяется всеми представителями этого класса, так как на самом деле существует только один экземпляр этой переменной. На самом деле, память под статические данные-члены выделяется, даже если нет никаких представителей класса. Поэтому класс должен не только объявлять статические данные-члены, но и определять их. Например:

```
class AnyClass
{
 public:
   AnyClass();
//объявление статического данного-члена
   static int count;
};
//определение статического данного-члена
int AnyClass::count = 0;
```

Заметим, что хотя к статическим данным-членам, объявленным в разделе public класса, можно обращаться с помощью конструкций, использующих имя объекта:

```
<объект>.<данное_член>
```

или

```
<указатель_на_объект>-><данное_член>
```

лучше использовать форму вызова

```
<имя_класса>::<данное_член>
```

которая является более правильной, так как отражает тот факт, что соответствующее данное-член является единственным для всего класса. Если статические данные-члены объявлены как закрытые, то доступ к ним можно получить с помощью обычных функций-членов. Доступ к статическим данным-членам с помощью обычных функций-членов ничем не отличается от доступа к другим данным-членам, но для этого необходимо создать хотя бы один объект данного класса. В связи со сказанным выше, можно дать следующие рекомендации:

- применяйте статические данные-члены для совместного использования данных несколькими объектами класса;
- ограничьте доступ к статическим данным-членам, объявив их в разделе protected или private.

Рассмотрим пример использования статического данного-члена класса:

```
#include <iostream.h>
class T
{
 public:
  T(){ObCount++;}
  ~T(){ObCount--;}
  static int ObCount;
  //...
 private:
  int x;
};
int T::ObCount = 0;
int main()
{
 T* pOb = new T[5];
 cout << " Имеется " << T::ObCount
      << " объектов типа T\n";
```

```
 delete[] pOb;
 return 0;
}
```

В этом примере статическое данное-член просто ведет учет созданных объектов. При выполнении программа выводит на экран:

```
Имеется 5 объектов типа T
```

Особенностью использования статических функций-членов является то, что они также определены в единственном экземпляре и не являются безраздельной собственностью какого-то представителя класса. В связи с этим им не передается указатель this. Эта особенность статических функций-членов используется при написании функций-обработчиков прерываний и callback-функций (при программировании для Windows).

Из сказанного выше вытекает несколько важных следствий:

- статическая функция-член может вызываться независимо от того, существует какой-либо представитель класса или нет;
- статическая функция-член может манипулировать только статическими данными-членами класса и вызывать только другие статические функции-члены класса;
- статическая функция-член не может быть объявлена с модификатором virtual.

Приведем пример использования статической функции-члена. Он представляет собой слегка отредактированный предыдущий пример:

```
#include <iostream.h>
class T
{
 public:
 T(){ObCount++;}
~T(){ObCount--;}
 //...
 static int GetCounts(){return ObCount;}
   private:
 int x;
 static int ObCount;
};
int T::ObCount = 0;
int main()
{
 T* pOb = new T[5];
 cout << "Имеется " << T::GetCounts()
```

```
    << " объектов типа  T\n";
  delete[] pOb;
  return 0;
}
```

Тема 12.8
Константные объекты и константные функции-члены класса. Ключевое слово mutable

Функция-член класса может быть объявлена с модификатором const, который следует за списком параметров. Такая функция не может изменять значение данных-членов класса и не может вызывать неконстантные функции-члены класса. Приведем пример объявления константной функции-члена класса:

```
class Coord
{
  int x,y;
 public:
  Coord(int _x, int _y);
  void SetVal(int _x, int _y);
  //Константная функция-член
  void GetVal(int_x,int_y) const;
};
```

Можно также создавать константные объекты. Для этого их объявления предваряют модификатором const. Например:

```
const Coord ob(3,5);
```

Ключевое слово const информирует компилятор, что состояние данного объекта не должно изменяться. В связи с этим компилятор генерирует сообщение об ошибке, если для константного объекта вызывается функция-член (которая может изменить его данные-члены, изменив тем самым его состояние). Исключением из этого правила являются константные функции-члены, которые, в силу своего определения, не изменяют состояние объекта.

Приведем следующий пример:

```
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y);
  void SetVal(int _x, int _y);
  void GetVal(int &_x, int &_y) const;
```

```
};
Coord::Coord(int _x, int _y)
{
 x = _x;
 y = _y;
}
void
Coord::SetVal(int _x, int _y)
{
 x = _x;
 y = _y;
}
//Константная функция-член
void
Coord::GetVal(int &_x, int &_y) const
{
 _x = x;
 _y = y;
}
int main()
{
 Coord pt1(3,8);
 const Coord pt(6,9);//Константный объект
 int x, y;
 pt1.GetVal(x,y);
 //Ошибка. Вызов неконстантной функции-члена
 pt2.SetVal(x,y);
 pt2.GetVal(x, y);
 return 0;
}
```

Чтобы обойти указанные выше ограничения на использование константных функций-членов класса, в стандарт языка C++ было введено новое ключевое слово mutable. Это ключевое слово позволяет указать, какие данные-члены класса могут быть модифицированы константными функциями-членами. Ключевое слово mutable нельзя использовать для статических и константных членов-данных; оно используется как *модификатор* типа, то есть синтаксис его использования имеет вид:

```
mutable <тип_данных> <имя_переменной-члена>;
```

Приведем пример использования ключевого слова mutable:

```
#include <iostream.h>
class AnyClass
{
  mutable int count;
  mutable const int* iptr;
```

```
public:
  int func(int i=0) const
      {
       count = i++;
       iptr = &i;
       cout << iptr;
       return count;
  }
};
```

Здесь в операторе

```
mutable const int* iptr;
```

модификатор mutable допустим, так как iptr является указателем на целое число, которое есть константа, хотя сам указатель константой не является.

Тема 12.9
Использование указателей на функции-члены класса

Можно определить указатель на функцию-член класса. Синтаксис определения следующий:

```
<возвр_тип>
      (<имя_класса>::*<имя_указателя>)(<параметры>);
```

Например:

```
void (T::*funcPtr)(int x, int y);
```

Таким образом, синтаксис определения указателя на функцию-член класса отличается от объявления указателя на обычную функцию тем, что в круглых скобках указывается имя класса, отделенное от имени указателя символом расширения области видимости, а само имя указателя предваряется звездочкой. Эта звездочка должна напомнить программисту, что при использовании такой указатель должен разыменовываться (в отличие от указателя на обычную функцию). Например:

```
this->*funcPtr(x,y);
```

или

```
*this.* funcPtr(x,y);
```

Указателю на функцию-член класса передается скрытый указатель this (иначе ему нельзя было бы присвоить значение адреса некоторой функции-члена класса).

Следующий пример демонстрирует использование указателей на функцию-член.

```
clas T
{
  int i;
 public:
  T(int _I): i(_i){}
  void MembFunc() {printf("Hello!");}
  void CallMembFunc(void (T::*funcPtr)() )
  {
      this->*funcPtr)();
   }
};
int main()
{
 void (T::*funcPtr)() = &T::MembFunc;
 T ob(2000);
 ob.CallMembFunc(funcPtr);
 return 0;
}
```

C++ накладывает определенные ограничения на использование указателей на функции-члены класса:

- указатель на функцию-член класса не может ссылаться на статическую функцию-член класса (так как ей не передается указатель this);
- указатель на функцию-член класса не может быть преобразован в указатель на обычную функцию, не являющуюся членом какого-нибудь класса (по той же причине).

Тема 12.10
Массивы объектов класса

Из объектов класса, как и из обычных переменных, можно строить массивы. Синтаксис объявления массива объектов аналогичен синтаксису объявления массивов обычных переменных. Например, следующее объявление создает массив из 10 элементов, которые являются объектами класса AnyClass:

```
AnyClass obArr[10];
```

Однако для того, чтобы компилятор смог создать этот массив, он должен использовать конструктор по умолчанию объектов класса. В отношении конструктора по умолчанию действуют пе-

речисленные нами ранее правила. Рекомендуем не полагаться на то, что компилятор сам создаст конструктор по умолчанию, и *всегда при объявлении массива объектов некоторого класса включать в этот класс конструктор по умолчанию.*

Доступ к элементам массива также аналогичен доступу к массивам переменных любого другого типа. Например:

```
#include <iostream.h>
class AnyClass
{
  int a;
 public:
  AnyClass(int n){a = n;}
  int GetA(){return a;}
};
main()
{
  //Объявление и инициализация массива объектов
 AnyClass obArr[5] = {13, 17, 21, 23, 27};
 int i;
 for (i=0; i<4; i++)
  {
      //Обращение к элементам массива
      cout << obArr[i].GetA() << ' ';
  }
 cout << "\n";
 return 0;
}
```

Эта программа выводит на экран проинициализированные значения массива объектов obArr. Фактически, примененный здесь синтаксис инициализации массива является сокращенной формой следующей конетрукции:

```
AnyClass obArr[5] = {AnyClass(13),
                     AnyClass(17),
                     AnyClass(21),
                     AnyClass(23),
                     AnyClass(27)};
```

К сожалению, ею можно воспользоваться только при инициализации массивов объектов, конструктор которых содержит только один параметр. При инициализации массивов объектов с конструктором, содержащим несколько параметров, приходится использовать полную (или длинную) форму конструкции. В следующем примере создается двумерный массив объектов, конструктор которого содержит два параметра.

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
  int GetX(){return x;}
  int GetY(){return y;}
};
main()
{
 Coord coordArr[4][2] =
{Coord(3,4),Coord(5,6),  Coord(7,8), Coord(9,10),
 Coord(11,12), Coord(13,14),  Coord(15,16), Coord(17,18),
};
int i, j;
for (i=0; i<4;i++)
for (j=0; j<2; j++)
  {
   cout << coordArr[i][j].GetX() << ' ';
   cout << coordArr[i][j].GetY() << ' ';
  }
 cout << "\n";
 return 0;
}
```

Использование указателей для доступа к объектам массива совершенно аналогично их использованию для обычных переменных и структур. Арифметика указателей также аналогична. Инкрементирование указателя приводит к тому, что он указывает на следующий объект массива, декрементирование – к тому, что он указывает на предыдущий объект массива. Рассмотрим пример:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
  int GetX(){return x;}
  int GetY(){return y;}
};
main()
{
 Coord obArr[4] = {
```

```
    Coord(3,4), Coord(5,6), Coord(7,8), Coord(9,10),
    Coord(11,12), Coord(13,14), Coord(15,16), Coord(17,18)
      };
    int i;
    Coord* ptr;
    ptr = obArr; //Инициализация указателя адресом массива
    for (i=0; I<4; i++)
      {
        cout << ptr->GetX() << ' ';
        cout << ptr->GetY() << "\n";
        ptr++;  //Переход на следующий объект
      }
   cout << "\n";
   return 0;
  }
```

Эта программа выводит на экран в каждой строке значения переменных x и y текущего элемента массива объектов.

Массивы объектов классов могут располагаться в куче. Например:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
  Coord(){x = 0; y = 0;}
  int GetX(){return x;}
  int GetY(){return y;}
};
main()
{
 int i;
 Coord* ptr;
  //Создание массива объектов в куче
 ptr = new Coord[6];
 for (i=0; I<6; i++)
  {
      cout << ptr->GetX() << ' ';
      cout << ptr->GetY() << "\n";
      ptr++;  //Переход на следующий объект
  }
cout << "\n";
//Удаление массива объектов из кучи
delete[] ptr;
return 0;
}
```

Следует обратить внимание, что для удаления массива объектов используется форма оператора delete с квадратными скобками.

Тема 12.11
Дружественные функции и друзья класса

C++ предоставляет возможность обойти (или нарушить) одну из основополагающих концепций ООП – концепцию инкапсуляции данных – с помощью друзей. Однако использовать ее без веских причин не стоит. C++ позволяет объявлять два вида друзей класса: дружественную функцию или дружественный класс.

Обычный способ доступа к закрытым членам класса – использование открытой функции-члена. Однако C++ поддерживает и другой способ получения доступа к закрытым членам класса – с помощью дружественных функций. Дружественные функции не являются членами класса, но тем не менее имеют доступ к его закрытым членам. Более того, одна такая функция может иметь доступ к закрытым членам нескольких классов.

Чтобы объявить функцию дружественной некоторому классу, в определение этого класса включают ее прототип, перед которым ставится ключевое слово friend. Приведем пример:

```
#include <iostream.h>
class AnyClass
{
  int n, d;
 public:
  AnyClass(int _n, int _d)
{n = _n; d = _d;}
  //Объявление дружественной функции
  friend bool IsFactor(AnyClass ob);
};
//Определение дружественной функции
bool IsFactor(Anyclass ob)
{
 if (!(ob.n % ob.d)) return true;
 else                return false;
}
main()
{
 AnyClass ob(12,3);
 if (IsFactor(ob))
```

```
  cout <<"12 делится без остатка на 3\n";
   else
     cout << "12 не делится без остатка на 3\n";
   return 0;
}
```

Подчеркнем еще раз, что дружественная функция не является членом класса, в котором она объявлена. Поэтому, вызывая дружественную функцию, не нужно указывать имя объекта или указатель на объект и операцию доступа к члену класса (точку или стрелку). Доступ к закрытым членам класса дружественная функция получает только через объект класса, который, в силу этого, должен быть либо объявлен внутри функции, либо передан ей в качестве аргумента. Дружественная функция не наследуется, то есть она не является таковой для производных классов. С другой стороны, функция может быть дружественной сразу нескольким классам.

Например:

```
#include <iostream.h>
class AnyClass1;//Неполное объявление класса
class AnyClass2
{
  int d;
 public:
  AnyClass2(int _d) {d = _d;}
  //Объявление дружественной функции
  friend bool IsFactor(AnyClass1 ob1,
  AnyClass2 ob2);
};
class AnyClass1
{
  int n;
 public:
  AnyClass1(int _n) {n = _n;}
  //Объявление дружественной функции
  friend bool IsFactor(AnyClass1 ob1,
  AnyClass2 ob2);
};
//Определение дружественной функции
bool IsFactor(AnyClass1 ob1, AnyClass2 ob2)
{
 if (!(ob1.n % ob2.d)) return true;
 else                  return false;
}
```

```
main()
{
 AnyClass1 ob1(12);
 AnyClass2 ob2(3);
 if (IsFactor(ob1, ob2))
   cout << "12 делится без остатка на 3\n";
 else
   cout << "12 не делится без остатка на 3\n";
 return 0;
}
```

Кроме того, эта программа демонстрирует важный случай применения неполного объявления класса: без применения этой конструкции в данном случае было бы невозможно объявить дружественную функцию для двух классов. Неполное объявление класса AnyClass1 дает возможность использовать его имя в объявлении дружественной функции еще до его определения.

Функция может быть членом одного класса и дружественной другому.

Например:

```
#include <iostream.h>
class AnyClass1;//Неполное объявление класса
class AnyClass2
{
  int d;
public:
  AnyClass2(int _d) {d = _d;}
  bool IsFactor(AnyClass1 ob1);
};
class AnyClass1
{
  int n;
public:
  AnyClass1(int _n) {n = _n;}
  //Объявление дружественной функции
  friend bool
  AnyClass2::IsFactor(AnyClass1 ob1);
};

//Определение дружественной функции
bool AnyClass::IsFactor(AnyClass1 ob1)
{
 if (!(ob1.n % d)) return true;
 else                    return false;
}
```

```
main()
{
 AnyClass1 ob1(12);
 AnyClass2 ob2(3);
 //IsFactor() вызывается как функция-член класса AnyClass2
 if (ob2.IsFactor(ob1))
   cout <<"12 делится без остатка на 3\n";
 else
   cout <<"12 не делится без остатка на 3\n";
 return 0;
}
```

C++ позволяет объявить не только дружественную функцию, но и дружественный класс, предоставив ему полный доступ к членам своего класса. Для этого достаточно включить в объявление класса имя другого класса, объявляемого дружественным, перед которым ставится ключевое слово friend. Например:

```
class AnyClass
{
friend class AnotherClass;
}
```

Класс не может объявить сам себя другом некоторого другого класса. Для того чтобы механизм дружественности сработал, он должен быть объявлен дружественным в этом другом классе.

Например:

```
#include <iostream.h>
class A
{
  // Класс B объявлен другом класса A
     friend class B;
     int x;
     void IncX(){x++;}
public:
     A(){x = 0;}
     A(int _x){x = _x;}
};
class B
{
     A  obA;
public:
     void ShowValues();
};
void
```

```
B::ShowValues()
{
 cout << "Сначала obA.x = "
  << obA.x << "\n";
 obA.IncX();
 cout << "Затем obA.x = " << obA.x;
}
main()
{
 B obB; // Конструктор по умолчанию,
        // генерируется компилятором
 obB.ShowValues();
 return 0;
}
```

Два класса могут объявить друг друга друзьями. С практической точки зрения такая ситуация свидетельствует о плохой продуманности иерархии классов, тем не менее язык C++ допускает такую возможность. В этом случае объявления классов должны иметь вид:

```
class B; //Неполное объявление класса
class A
{
 friend class B;
 //...
};
class B
{
 friend class A;
 //...
};
```

Неполное объявление класса, которое приведено в данном фрагменте, может понадобиться, только если в классе A имеется ссылка на класс B, например в параметре функции-члена.

По отношению к дружественным классам действуют следующие правила:

- дружественность не является взаимным свойством: если A друг B, это не означает, что B – друг A;
- дружественность не наследуется: если B – друг A, то классы, производные от B, не являются друзьями A;
- дружественность не переходит на потомки базового класса: если B – друг A, то B не является другом для классов, производных от A.

Практикум «Объектно-ориентированное программирование»

Упражнение 12.1
Классы

Проанализируйте предлагаемый ниже пример небольшого класса. Все, что он делает, – хранит, позволяет получать и модифицировать закрытую целочисленную переменную. Наличие конструктора с параметром типа int позволяет создавать объект класса с заданным начальным значением члена класса. Благодаря наличию открытых функций GetNumber() и SetNumber(), пользователь имеет возможность обращаться к закрытой переменной Number.

Обратите внимание на то, каким образом можно осуществлять реализацию функции при описании класса на примере функции чтения значения GetNumber().

```
// Модуль заголовка "header.h"
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
public:
   MyClass();
   MyClass(int number);
   ~MyClass();
   //
   int GetNumber(){return Number;}
   void SetNumber(int number);
private:
   int Number;
};
MyClass::MyClass()
{
   Number = 0;
}
MyClass::MyClass(int number)
{
   Number = number;
}
MyClass::~MyClass()
{
}
```

```
void MyClass::SetNumber(int number)
{
   Number = number;
}
#endif//_HEADER_H_
// Модуль программы
#include <iostream.h>
#include "header.h"
void main()
{
   MyClass Object(3);
   cout << Object.GetNumber() << endl;
   Object.SetNumber(123);
   cout << Object.GetNumber() << endl;
}
```

Упражнение 12.2
Конструкторы и деструкторы

Напишите класс, содержащий закрытые целочисленные переменные, характеризующие координаты точки в пространстве и ее температуру (с плавающей точкой). Класс должен содержать конструктор со списком инициализации координат точки, а также конструктор, в котором инициализируется значение температуры. В деструкторе должно выводиться сообщение о разрушении объекта.

Упражнение 12.3
Конструкторы по умолчанию и копирования

Проанализируйте пример, иллюстрирующий работу конструктора по умолчанию и конструктора копирования.

Имеется класс MyClass, в котором содержится указатель на целочисленную переменную iptr. Для того чтобы использовать этот указатель, необходимо выделить место в памяти с помощью оператора new. Эта операция осуществляется в конструкторе по умолчанию MyClass(), конструкторе с целочисленным аргументом MyClass(int number) либо в конструкторе копирования MyClass(const MyClass& src). Там же выполняется разыменование и инициализация переменной iptr. Во избежание утечек памяти при разрушении объектов данного класса в деструкторе оператором delete высвобождается выделенное место.

```
// Модуль заголовка "header.h"
class MyClass
{
public:
   // Конструктор по умолчанию
   MyClass();
   MyClass(int number);
   // Конструктор копирования
   MyClass(const MyClass& src);
   ~MyClass();
   int *iptr;
};

MyClass::MyClass()
{
   iptr = new int;
}
MyClass::MyClass(int number)
{
   iptr = new int;
   *iptr = number;
}
MyClass::MyClass(const MyClass& src)
{
   iptr = new int;
   *iptr = *src.iptr;
}
MyClass::~MyClass()
{
   delete iptr;
}

// Модуль программы
#include <iostream.h>
#include "header.h"
void main()
{
   // Вызов конструктора по умолчанию
   MyClass Object1;
   *Object1.iptr = 555;
   cout << *Object1.iptr << endl;
   MyClass Object2(123);
   cout << *Object2.iptr << endl;
   // Вызов конструктора копирования
   MyClass Object3 = Object1;
   cout << *Object3.iptr;
}
```

Упражнение 12.4
Встраиваемые (inline) функции

Модифицируйте пример упражнения 12.1 таким образом, чтобы все функции данного класса были встраиваемыми.

Упражнение 12.5
Использование статических данных в классе

Проанализируйте рассмотренный ниже пример. Пусть необходимо создать класс, который будет контролировать количество своих экземпляров и запрещать создание новых экземпляров, если хоть один из них уже существует.

Воспользуемся для этого статическим целочисленным членом класса Instance. В основном модуле программы проинициализируем его значением 0, а в конструкторе будем использовать его значение для принятия решения о возможности создания экземпляра объекта данного класса.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
#include <iostream.h>
class MyClass
{
public:
   MyClass();
   ~MyClass();
   static int Instance;
};
MyClass::MyClass()
{
   if(MyClass::Instance > 0)
   {
      cout << "Должен быть только один!\n";
      return;
   }
   MyClass::Instance++;
}
MyClass::~MyClass()
{
   MyClass::Instance--;
}
#endif//_HEADER_H_
```

```
// Основной модуль программы
#include "header.h"
int MyClass::Instance = 0;
void main()
{
   MyClass *MyObject[10];
   for(int i=0; i<10; i++)
   {
      MyObject[i] = new MyClass;
   }
   cout << MyClass::Instance;
   delete[] MyObject;
}
```

Упражнение 12.6
Работа с константными данными класса

Проанализируйте предлагаемый ниже пример, иллюстрирующий создание объектов, содержащих константные члены-данные. Для себя отметьте, что такие данные инициализируются с помощью списка.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
   const int x;
   char letter;
public:
   MyClass():x(10),letter('x'){};
   MyClass(int _x):x(_x){letter = 'a';}
   MyClass(char _letter):x(12){letter = _letter;}
   int GetX()const {return x;}
   char GetLetter()const {return letter;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
#include <iostream.h>
void main()
{
   MyClass DefaultObject;
   MyClass IntObject(13);
```

```
    MyClass CharObject('Z');
    cout << DefaultObject.GetX() << '\t';
    cout << DefaultObject.GetLetter() << endl;
    cout << IntObject.GetX() << '\t';
    cout << IntObject.GetLetter() << endl;
    cout << CharObject.GetX() << '\t';
    cout << CharObject.GetLetter() << endl;
}
```

Упражнение 12.7
Указатели на функции-члены класса

В предлагаемом упражнении Вам необходимо разработать класс, использующий указатель на функцию-член.

Упражнение 12.8
Работа с массивами объектов

Проанализируйте приведенный ниже пример. Для наглядной демонстрации массивов сложных объектов разработаем небольшой телефонный справочник. Каждая запись справочника (класс MyClass) будет содержать закрытую информацию об имени, фамилии и телефонном номере некоторого абонента. Доступ к элементам будем осуществлять посредством соответствующих функций-членов.

В основном модуле приведенной программы создадим одиночный объект (Record) разработанного класса, а также массив из пяти аналогичных объектов (recs).

```
// Заголовочный файл
#include <string.h>
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
    char Name[20];
    char SurName[20];
    char Phone[15];
public:
    MyClass(){};
    void SetName(char* _name){strcpy(Name, _name);}
    void SetSurName(char* _surname)
{strcpy(SurName, _surname);}
    void SetPhone(char* _phone){strcpy(Phone, _phone);}
```

```
    char* GetName(){return Name;}
    char* GetSurName(){return SurName;}
    char* GetPhone(){return Phone;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
#include <iostream.h>
void main()
{
    MyClass Record;
    Record.SetName("Helen");
    Record.SetSurName("Ripley");
    Record.SetPhone("555-43-25");
    cout << Record.GetName() << '\t';
    cout << Record.GetSurName() << '\t';
    cout << Record.GetPhone() << endl;
    MyClass *recs[5];
    for(int i=0; i<5; i++)
    {
        recs[i] = new MyClass;
        recs[i]->SetName("James");
        recs[i]->SetSurName("Bond");
        recs[i]->SetPhone("123-45-67");
        cout << recs[i]->GetName() << '\t'
             << recs[i]->GetSurName() << '\t'
             << recs[i]->GetPhone() << endl;
    }
    delete[] recs;
}
```

Упражнение 12.9

Примеры дружественных функций

Составьте пример класса, содержащего два целочисленных данных-члена, а также дружественную функцию, выполняющую суммирование и вывод на экран этих данных.

Раздел 13
Наследование

Тема 13.1
Простое наследование

Язык C++ позволяет классу *наследовать* данные-члены и функции-члены одного или нескольких других классов. При этом новый класс называют *производным классом* (или *классом-потомком*). Класс, элементы которого наследуются, называется *базовым классом* (*родительским классом*, или *классом-предком*) для своего производного класса. Наследование дает возможность некоторые общие черты поведения классов абстрагировать в одном базовом классе. Производные классы, наследуя это общее поведение, могут его несколько видоизменять, переопределяя некоторые функции-члены базового класса, или дополнять, вводя новые данные-члены и функции-члены. Таким образом, определение производного класса значительно сокращается, поскольку нужно определить только отличающие его от производных классов черты поведения.

Синтаксис объявления производного класса имеет следующий вид:

```
//Базовый класс
class Base1
{
 //...описание класса Base1
};
//Базовый класс
class Base2
{
 //... описание класса Base2
};
//Базовый класс
class BaseN
{
```

```
 //... описание класса BaseN
};
//Производный класс
class Derived :<спецификатор_доступа> Base1,
<спецификатор_доступа> Base2,
                    ...,
<спецификатор_доступа> BaseN
{
 //... описание спецификатора
};
```

Здесь <спецификатор_доступа> – это public, protected или private; он не является обязательным и по умолчанию принимает значение private для классов и public для структур. Спецификатор доступа при наследовании определяет уровень доступа к элементам базового класса, который получают элементы производного класса. В приведенной ниже табл. 13.1 описаны возможные варианты наследуемого доступа.

Таблица 13.1
Наследуемый доступ

Спецификатор наследуемого доступа	Доступ в базовом классе	Доступ в производном классе
public	public protected private	public protected недоступны
protected	public protected private	protected protected недоступны
private	public protected private	private private недоступны

Изучение таблицы показывает, что спецификатор наследуемого доступа устанавливает тот уровень доступа, до которого понижается уровень доступа к членам, установленный в базовом классе. Из этого правила можно сделать исключения. Если спецификатор наследуемого доступа установлен как private, то public-члены базового класса будут являться private-членами в производном классе. Однако можно сделать некоторые из членов базового класса открытыми в производном классе, объявив их в секции public производного класса. Например:

```
class Base
{
      int x,y;
 public:
      int GetX(){return x;}
      int GetY(){return y;}
};
class Derived : private Base
{
 public:
     Base::GetX;
};
main()
{
 int x;
 Derived ob;
 x = ob.GetX();
 return 0;
}
```

Таким образом, при таком наследовании открытые и защищенные члены базового класса будут доступны только для членов данного производного класса, и все члены базового класса, кроме явно объявленных в разделе public или protected, будут закрытыми для следующих производных классов. Этот прием позволяет отсечь доступ к членам базового класса при построении иерархии классов с отношением родитель – потомок.

Напомним, что при любом способе наследования в производном классе *доступны* только открытые (public) и защищенные (protected) члены базового класса (хотя *наследуются* все члены базового класса). Иначе говоря, закрытые члены базового класса остаются закрытыми, *независимо от того, как этот класс наследуется*.

Как мы ранее уже отмечали, не все члены класса наследуются. Для удобства мы перечислим здесь не наследуемые члены класса:

- конструкторы;
- конструкторы копирования;
- деструкторы;
- операторы присвоения, определенные программистом;
- друзья класса.

Здесь упомянут еще не рассмотренный нами вид членов класса. Он будет рассмотрен позже.

Если у производного класса имеется всего один базовый класс, то говорят о *простом* (или *одиночном*) *наследовании*. Следующий пример иллюстрирует простое наследование.

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
  Coord(){x = 0; y = 0;}
  int GetX(){return x;}
  int GetY(){return y;}
  void SetX(int _x){x = _x;}
  void SetY(int _y){y = _y;}
};
class OutCoord: public Coord
{
 public:
  OutCoord(int _x, int _y): Coord(_x, _y){}
  void ShowX(){cout << GetX() << ' ';}
  void ShowY(){cout << GetY()<< ' ';}
};
main()
{
 OutCoord* ptr;
 ptr = new OutCoord(10,20);//Создание объекта в куче
 ptr->ShowX();
 ptr->ShowY();
 cout << "\n";
 delete ptr;    //Удаление объекта из кучи
 return 0;
}
```

Здесь класс OutCoord является производным от базового класса Coord. К членам класса Coord он добавляет две открытые функции и конструктор. Напомним, что *конструкторы не наследуются.* Поэтому производный класс либо должен объявить свой конструктор, либо предоставить возможность компилятору сгенерировать конструктор по умолчанию. Правила, действующие в отношении конструкторов по умолчанию, были изложены выше. Рассмотрим теперь, как строится конструктор производного класса, предоставляемый программистом. Поскольку производный класс должен унаследовать все члены родительского, при построении объекта своего класса он должен обеспечить инициализацию унаследованных данных-членов, причем она должна быть выполнена *до* инициализаций данных-членов производного класса, так как последние могут использовать значения первых. В связи с этим для построения конструктора производного класса применяется следующая конструкция:

```
<констр_произв_класса>(<список параметров>) :
  <констр_базового_класса>(<список_арг>)
{<тело_конструктора>}
```

То есть используется список инициализации элементов, в котором указывается конструктор базового класса. Часть параметров, переданных конструктору производного класса, обычно используется в качестве аргументов конструктора базового класса. Затем в теле конструктора производного класса выполняется инициализация данных-членов, принадлежащих собственно этому классу.

В приведенном выше примере эта конструкция использована для создания конструктора класса OutClass. В данном случае конструктор имеет вид:

```
OutCoord(int _x, int _y): Coord(_x, _y){}
```

Тело конструктора оставлено пустым, так как класс OutCoord не имеет собственных данных-членов. Все параметры конструктора производного класса просто переданы конструктору базового класса для осуществления инициализации. Аргументы конструктору производного класса передаются в операторе new, который неявно вызывает этот конструктор. Затем программа вызывает функции-члены ShowX() и ShowY() объекта, являющегося представителем производного класса, которые выводят на экран в одной строке заданные значения координат. Наконец, оператор delete вызывает удаление объекта, для чего он неявно вызывает деструктор производного класса.

Если предоставляемый программистом конструктор не имеет параметров, то есть является конструктором по умолчанию, то при создании экземпляра производного класса *автоматически* вызывается конструктор базового класса. После того как объект создан, конструктор базового класса становится недоступным.

Хотя конструктор базового класса и наследуется, вызывается он только компилятором, когда конструируется объект производного класса. Конструктор, в отличие от других унаследованных функций, вызвать явно нельзя.

В отношении деструкторов производных классов также действуют определенные правила. Деструктор производного класса должен выполняться *раньше* деструктора базового класса (иначе деструктор базового класса мог бы разрушить данные-члены, которые используются и в производном классе). Когда деструктор производного класса выполнит свою часть работы по уничтожению объекта, вызывается деструктор базового класса. Причем вся

работа по организации соответствующего вызова возлагается на компилятор, программист не должен заботиться об этом.

Чтобы более наглядно изучить работу конструкторов и деструкторов базового и производного классов, рассмотрим следующий пример:

```
#include <iostream.h>
class Base
{
 public:
   Base()
{
 cout << "Мы в конструкторе базового класса" << "\n";
  }
~Base()
  {
   cout << "Мы в деструкторе базового класса" << "\n";
    }
  };
class Derived: public Base
{
 public:
   Derived()
   {
 cout <<"Мы в конструкторе производного класса"<< "\n";
   }
      ~Derived()
      {
 cout <<"Мы в деструкторе производного класса"<< "\n";
}
};
main()
{
 Derived ob;
 return 0;
}
```

Эта программа выводит на экран следующее:

```
Мы в конструкторе базового класса
Мы в конструкторе производного класса
Мы в деструкторе производного класса
Мы в деструкторе базового класса
```

Для дальнейшего очень важно понимать работу конструкторов и деструкторов при наследовании.

В производном классе обычно добавляются новые члены к членам базового класса. Однако существует также возможность *пере-*

определения (или *замещения*) членов базового класса. Обычно используется переопределение функций-членов базового классса. Чтобы переопределить функцию-член базового класса в производном классе, достаточно включить ее прототип в объявление этого класса и затем дать ее определение. Конечно, прототипы переопределяемой функции в базовом и производном классах должны совпадать.

Модифицируем рассмотренный нами пример так, чтобы производный класс переопределял функции-члены своего базового класса GetX() и GetY():

```
#include <iostream.h>
class Coord
{
 protected:
      int x, y;
 public:
   Coord(int _x, int _y){x = _x; y = _y;}
      Coord(){x = 0; y = 0;}
      int GetX(){return x;}
      int GetY(){return y;}
      void SetX(int _x){x = _x;}
      void SetY(int _y){y = _y;}
};
class OutCoord: public Coord
{
 public:
OutCoord(int _x, int _y): Coord(_x,_y){}
      int GetX(){return ++x;}
      int GetY(){return ++y;}
      void ShowX(){cout << GetX() << ' ';}
      void ShowY(){cout << GetY()<< ' ';}
};
main()
{
 OutCoord* ptr;
 ptr = new OutCoord(10,20); //Создание объекта в куче
 ptr->ShowX();
 ptr->ShowY();
 cout << "\n";
 delete ptr;                    //Удаление объекта из кучи
 return 0;
}
```

В этом случае функции-члены ShowY() и ShowX() объекта ptr используют для получения значений данных-членов переопределенные варианты функций базового класса GetX() и GetY(). В ре-

зультате программа выведет на экран инкрементированные значения заданных координат.

Иногда возникает необходимость вызвать функцию-член базового класса, а не ее переопределенный вариант, это можно сделать с помощью *операции разрешения области видимости*, применяемой в форме:

```
<имя_класса>::<имя_члена>
```

Это дает возможность компилятору «видеть» за пределами текущей области видимости. Особенно часто это приходится делать при переопределении функций-членов. Переопределяемая функция-член может вызывать соответствующую функцию-член базового класса, а затем выполнять некоторую дополнительную работу (или наоборот). Например, в предыдущем примере можно было переопределить функцию GetX() следующим образом:

```
int
OutCoord::GetX()
{
 int z;
   //Вызов переопределенной функции
 z = Coord::GetX();
 return ++z;
}
```

Точно так же можно было бы использовать операцию разрешения области видимости для квалификации вызываемой перегруженной функции в ShowX():

```
void ShowX(){cout << Coord::GetX() << ' ';}
```

Производный класс сам может служить в качестве базового класса для некоторого другого класса. При этом исходный базовый класс называется *косвенным* базовым классом для второго базового класса.

Тема 13.2
Множественное наследование

Если у производного класса имеется несколько базовых классов, то говорят о *множественном наследовании*. Множественное наследование позволяет сочетать в одном производном классе свойства и поведение нескольких классов.

Следующий пример демонстрирует множественное наследование.

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

class Coord
{
      int x, y;
 public:
Coord(int _x, int _y){x = _x; y = _y;}
      int GetX(){return x;}
      int GetY(){return y;}
      void SetX(int _x){x = _x;}
      void SetY(int _y){y = _y;}
};

class SaveMsg
{
      char Message[80];
 public:
      SaveMsg(char* msg){SetMsg(msg);}
      void SetMsg(char* msg)
{strcpy(Message, msg);}
      void ShowMsg(){cout << Message;}
};

class PrintMsg: public Coord, public SaveMsg
{
 public:
   PrintMsg(int _x, int _y, char* msg):
      Coord(_x, _y),SaveMsg(msg){}
   void Show();
};

void
PrintMsg::Show()
{
 gotoxy(GetX(), GetY());
 ShowMsg();
}

main()
{
 PrintMsg * ptr;
 ptr =new PrintMsg(10, 5, "Множественное ");
 ptr->Show();
 ptr->SetX(25);
 ptr->SetY(5);
 ptr->SetMsg("наследование");
 ptr->Show();
 delete ptr;
 return 0;
}
```

В этом примере класс Coord отвечает за хранение и установку значений координат на экране, класс SaveMsg хранит и устанавливает сообщение, а класс PrintMsg, являющийся производным от них, выводит это сообщение на экран в заданных координатах. Этот пример демонстрирует также, как осуществляется передача параметров конструкторам базовых классов. При множественном наследовании, как и при простом, конструкторы базовых классов вызываются компилятором *до* вызова конструктора производного класса. Единственная возможность передать им аргументы – использовать список инициализации элементов. Причем порядок объявления базовых классов при наследовании определяет и порядок вызова их конструкторов, которому должен соответствовать порядок следования конструкторов базовых классов в списке инициализации. В данном случае класс PrintMsg содержит такое объявление о наследовании:

```
class PrintMsg: public Coord, public SaveMsg
```

Этому объявлению соответствует следующий конструктор:

```
PrintMsg(int _x, int _y, char* msg):
Coord(_x, _y), SaveMsg(msg){}
```

В основной программе этому конструктору передаются аргументы при создании объекта класса PrintMsg с помощью оператора new:

```
ptr = new PrintMsg(10, 5, "Множественное ");
```

Выполнение функции-члена Show() приводит к выводу на экран в указанных координатах сообщения, заданного третьим аргументом.

Следующие за этим вызовы функций-членов SetX(), SetY() и SetMsg() приводят к установке новых значений координат и нового сообщения, которое выводится повторным вызовом функции Show().

Рассмотрим еще один пример, который демонстрирует порядок вызова конструкторов и деструкторов базовых классов.

```
#include <iostream.h>
class Base1
{
 public:
   Base1(){cout <<"Мы в конструкторе Base1\n";}
   ~Base1(){cout << "Мы в деструкторе Base1\n";}
};
```

```
class Base2
{
 public:
   Base2() {cout <<"Мы в конструкторе Base2\n";}
   ~Base2(){cout << "Мы в деструкторе Base2\n";}
};
class Derived: public Base1, public Base2
{
 public:
   Derived()
{
 cout << "Мы в конструкторе Derived \n";
}
     ~Derived()
{
 cout << "Мы в деструкторе Derived \n";
}
};
main()
{
 Derived ob;
 return 0;
}
```

Эта программа выводит на экран следующее:

```
Мы в конструкторе Base1
Мы в конструкторе Base2
Мы в конструкторе Derived
Мы в деструкторе Derived
Мы в деструкторе Base2
Мы в деструкторе Base1
```

Этот пример наглядно демонстрирует, что конструкторы базовых классов вызываются в порядке их объявления. Деструкторы вызываются в обратном порядке.

Тема 13.3
Виртуальные базовые классы

В сложной иерархии классов при множественном наследовании может получиться так, что производный класс *косвенно* унаследует два или более экземпляров одного и того же класса. Рассмотрим пример, который проясняет эту ситуацию.

```
class IndBase
{
```

```
  private:
    int x;
  public:
    int GetX(){return x;}
    void SetX(int _x){x = _x;}
    double var;
};
class Base1: public IndBase
{
   ...
};
class Base2: public IndBase
{
   ...
};
class Derived: public Base1, public Base2
{
   ...
};
main()
{
  Derived ob;
  ob.var = 5.0;
  ob.SetX(0);
  int z = ob.GetX();
  //ob.Base1::var = 5.0;
  //ob. Base1::SetX(0);
  //int z = ob.Base1::GetX();
  return 0;
}
```

Здесь класс Derived косвенно наследует класс IndBase через свои базовые классы Base1 и Base2. Поэтому при компиляции приведенного примера возникнут ошибки, вызванные неоднозначностью обращения к членам класса var и GetX() в строках:

```
ob.var = 5.0;
ob.SetX(0);
int z = ob.GetX();
```

Чтобы избежать этой неоднозначности, можно использовать квалификацию имен, применив операцию разрешения видимости:

```
ob.Base1::var = 5.0;
ob.Base1::SetX(0);
int z = ob.Base1::GetX();
```

Можно также квалифицировать эти вызовы следующим образом:

```
ob.Base2::var = 5.0;
ob.Base2::SetX(0);
int z = ob.Base2::GetX();
```

Хотя этот способ и позволяет избежать неоднозначности при вызове, тем не менее класс IndBase будет включен в состав класса Derived дважды, увеличивая его размер. Избежать повторного включения косвенного базового класса в производный класс можно, дав указание компилятору использовать *виртуальный базовый класс*. Это осуществляется с помощью ключевого слова virtual, которое указывается перед спецификатором наследуемого доступа или после него. Следующий пример является модифицированным вариантом предыдущего, использующим класс IndBase в качестве виртуального базового класса.

```
class IndBase
{
 private:
   int x;
public:
   int GetX(){return x;}
   void SetX(int _x){x = _x;}
      double var;
};
class Base1: virtual public IndBase
{
   ...
};
class Base2: virtual public IndBase
{
   ...
};
class Derived: public Base1, public Base2
{
   ...
};
main()
{
  Derived ob;
  ob.var = 5.0;
  ob.SetX(0);
  int z = ob.GetX();
  return 0;
}
```

В этом случае класс Derived содержит один экземпляр класса IndBase, и вызовы

```
ob.var = 5.0;
ob.SetX(0);
int z = ob.GetX();
```

не приводят к появлению сообщений компилятора об ошибках, связанных с неоднозначностью.

Отсюда следует общая рекомендация: если разрабатывается иерархия классов и данный класс наследуется несколькими классами, включите перед спецификаторами наследуемого доступа ключевое слово virtual, то есть наследуйте его как виртуальный базовый класс.

Наконец, отметим один важный момент. В случае, если класс имеет один или несколько базовых классов, их конструкторы вызываются перед вызовом конструктора производного класса. Конструкторы базовых классов вызываются в том порядке, в котором они объявлены в списке наследования. Например, в случае объявления

```
class Base1{};
class Base2{};
class Derived : public Base1,
 public Base2 {};
main()
{
 Derived ob;
   ...
 return 0;
}
```

конструкторы вызываются в следующем порядке:

```
Base1();
Base2();
Derived();
```

Объявление базового класса виртуальным изменяет порядок вызова конструкторов при создании экземпляра производного класса. Конструкторы виртуальных базовых классов вызываются первыми, раньше конструкторов невиртуальных базовых классов. Если виртуальных базовых классов несколько, их конструкторы вызываются в порядке их объявления в списке наследования. Затем вызываются конструкторы невиртуальных базовых классов в порядке их объявления в списке наследования и, наконец, вызы-

вается конструктор производного класса. Если какой-то виртуальный класс является производным невиртуального базового класса, этот невиртуальный базовый класс конструируется первым (иначе нельзя будет вызвать конструктор виртуального базового класса). Например, в случае объявления

```
class Base1{};
class Base2{};
class Derived: public Base1,
virtual public Base2 {};
main()
{
 Derived ob;
   ...
return 0;
}
```

конструкторы вызываются в следующем порядке:

```
Base2();
Base1();
Derived();
```

Если иерархия классов содержит несколько экземпляров виртуального базового класса, этот базовый класс конструируется только один раз. Если все же существуют как виртуальный, так и невиртуальный экземпляры этого базового класса, конструктор базового класса вызывается один раз для всех виртуальных экземпляров этого базового класса, а затем еще раз для каждого невиртуального экземпляра этого базового класса.

Деструкторы вызываются в порядке, в точности обратном конструкторам.

Практикум «Наследование»

Упражнение 13.1
Создание производных классов

Разработайте класс-предок, содержащий один закрытый целочисленный член-данные с соответствующими открытыми функциями-членами, и класс-потомок с защищенным наследуемым доступом, добавляющий открытый член-данные вещественного типа.

Упражнение 13.2
Применение множественного наследования

Проанализируйте пример, иллюстрирующий множественное наследование. Класс MyPoint является потомком сразу трех базовых классов – ProjX, ProjY и ProjZ. Каждый из классов-предков описывает координату и время для каждой проекции, в то время как класс-потомок добавляет функции-члены для работы с вышеприведенными данными-членами.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class ProjX
{
public:
   double X;
   float TimeX;
   ProjX(){X = 0; TimeX = 0;}
   ~ProjX(){};
};
class ProjY
{
public:
   double Y;
   float TimeY;
   ProjY(){Y = 0; TimeY = 0;}
   ~ProjY(){};
};
class ProjZ
{
public:
   double Z;
   float TimeZ;
   ProjZ(){Z = 0; TimeZ = 0;}
   ~ProjZ(){};
};
class MyPoint : protected ProjX, protected ProjY, protected ProjZ
{
public:
   MyPoint(){SetXYZ(0, 0, 0); SetTime(0, 0, 0);};
   ~MyPoint(){};
   void GetXYZ(double& x, double& y, double& z)
   {
      x = X; y = Y; z = Z;
   }
```

```
    void SetXYZ(double x, double y, double z)
    {
      X = x; Y = y; Z = z;
    }
    void GetTime(float& tx, float& ty, float& tz)
    {
      tx = TimeX; ty = TimeY; tz = TimeZ;
    }
    void SetTime(float tx, float ty, float tz)
    {
      TimeX = tx; TimeY = ty; TimeZ = tz;
    }
};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
#include <iostream.h>
void main()
{
  MyPoint pnt;
  double x, y, z;
  float a, b, c;
  pnt.SetXYZ(10.23, .254, 111325.88895);
  pnt.GetXYZ(x, y, z);
  cout << x << endl;
  cout << y << endl;
  cout << z << endl;
  pnt.GetTime(a, b, c);
  cout << a << endl;
  cout << b << endl;
  cout << c << endl;
}
```

Упражнение 13.3

Использование виртуальных базовых классов

В приведенном ниже примере Вам предлагается модифицировать код таким образом, чтобы программа компилировалась без ошибок.

```
// Заголовочный файл
#include <string.h>
#include <iostream.h>
#ifndef _HEADER_H_
#define _HEADER_H_
```

```
class Base
{
   char Name[20];
   char SurName[20];
public:
   Base(){strcpy(Name, ""); strcpy(SurName, "");}
   void SetName(char* _name){strcpy(Name, _name);}
   char* GetName(){return Name;}
   void SetSurName(char* _sname){strcpy(SurName, _sname);}
   char* GetSurName(){return SurName;}
};
class Female : public Base
{
   char VirgName[20];
public:
   Female(){}
   char* GetVirgName(){return VirgName;}
   void SetVirgName(char* _vname){strcpy(VirgName, _vname);}
};
class Male : public Base
{
   bool ArmyServe;
public:
   Male(){}
};
class MyRecord : public Female, public Male
{
public:
   MyRecord(){}
   void PrintName(){cout << GetName() << ' ' <<GetSurName();}
};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
void main()
{
   MyRecord rec;
   rec.SetName("Alain");
   rec.SetSurName("Cardin");
   rec.PrintName();
}
```

Раздел 14
Перегрузка функций

Тема 14.1
Почему следует использовать перегрузку

Перегрузка функций – это один из типов полиморфизма, обеспечиваемого C++. В C++ несколько функций могут иметь одно и то же имя. В этом случае функция, идентифицируемая этим именем, называется перегруженной. Перегрузить можно только функции, которые отличаются либо типом, либо числом своих аргументов. Перегрузить функции, которые отличаются только типом возвращаемого значения, нельзя. Возникает вопрос: зачем нужно использовать перегрузку? Ответ прост: перегружаемые функции дают возможность упростить программы, допуская обращение к одному имени для выполнения близких по смыслу действий.

Чтобы перегрузить некоторую функцию, нужно просто ее объявить, а затем определить все требуемые варианты ее вызова. Компилятор автоматически выберет правильный вариант вызова на основании числа и типа используемых аргументов. Пример:

```
#include <iostream.h>

//Перегрузка функции abs()
int abs(int n);
long abs(long l);
double abs(double d);

main()
{
  int n=255; long l=-25L; double d=-15.0L;
  cout << "Абсолютная величина " << n <<" равна: "
       << abs(n) << "\n";
  cout << "Абсолютная величина " << l <<" равна: "
       << abs(l) << "\n";
  cout << "Абсолютная величина " << d <<" равна: "
     << abs(d) << "\n";
  return 0;
```

```
}
int abs(int n)
{
 return n<0? -n:n;
}
long abs(long l)
{
 return l<0? -l:l;
}
double abs(double d)
{
 return d<0? -d:d;
}
```

Как известно, библиотека времени выполнения C++ использует три разных функции abs(), labs() и fabs() для вычисления абсолютного значения аргумента. Использование той или другой из них зависит от типа аргумента. Так, в данном примере осуществляется перегрузка функции abs(), что упрощает программу. В зависимости от переданного аргумента вызывается нужный вариант функции.

Этот пример наглядно демонстрирует, как использование перегрузки может упростить код, предоставляя программисту возможность пользоваться одной функцией, которой передаются аргументы различного типа. Разумеется, возможность использования перегрузки предъявляет повышенные требования к компилятору.

Чтобы реализовать концепцию перегрузки, разработчикам компиляторов C++ пришлось ввести декорирование имен. Последнее означает, что все функции в коде программы получают от компилятора имена, основываясь на имени, заданном программистом, и количестве и типах аргументов. Различные компиляторы делают это несколько отличным друг от друга образом. Здесь мы опишем, как это делает компилятор фирмы Inprise (ранее – фирма Borland): вначале идет имя класса (если речь идет о функции-члене класса), предваряемое символом "@", затем имя функции, снова предваряемое символом "@". У всех идентификаторов различается регистр букв. Затем следует последовательность символов "@q", начиная с которой идут кодированные обозначения параметров функции. Для обозначения указателей и ссылок к кодам встроенных типов добавляются буквы "p" и "r" соответственно. Например, если дано такое определение класса:

```
class AnyClass
{
 public:
```

```
    void SetVal();
    void SetVal(int);
    void SetVal(int, int);
    void SetVal(int, int, int);
    void SetVal(int&, int, int);
};
```

компилятор Borland C++ сгенерирует такие имена функций:

```
@AnyClass@SetVal@qv
@AnyClass@SetVal@qi
@AnyClass@SetVal@qii
@AnyClass@SetVal@qiii
@AnyClass@SetVal@qriii
```

Подробнее о декорировании имен, в частности о кодировании имен конструкторов и деструкторов, а также имен, указанных через операцию разрешения области видимости, можно прочесть в документации, поставляемой с компилятором.

Отсюда видно, что в декорировании имен возвращаемое функцией значение не участвует. Именно по этой причине нельзя перегружать функции, которые отличаются только типом возвращаемого значения. Разобраться с тем, как декорирует имена конкретный компилятор, довольно легко. Достаточно потребовать от него выдать ассемблерный эквивалент текста программы.

Декорирование имен порождает и определенные проблемы. Если нужно написать код, который будет вызываться из программы, написанной на другом языке или даже на другом компиляторе C++, нужно отключить декорирование имен, иначе нужная функция не будет найдена. Это делается с помощью ключевого слова extern в форме extern "C"{}. Буква "C" здесь напоминает о языке C, в котором декорирования имен не было.

Тема 14.2
Перегрузка функций

Перегрузка – это практика предоставления более чем одного определения для данного имени функции в одной и той же области видимости. Возможность выбрать соответствующую версию функции, основываясь на типах и числе аргументов, с которыми она вызывается, предоставляется компилятору. При этом два типа данных считаются различными, если для них используются различные инициализаторы. Поэтому аргумент данного типа и ссылка на этот тип рассматриваются как одно и то же с точки зрения перегрузки. Например, объявление двух функций

```
int func(int, int)
```

и

```
int func(int&, int&)
```

приведет к ошибке, так как с точки зрения перегрузки они считаются одинаковыми. По той же причине аргументы функции, относящиеся к некоторому типу, модифицированные const или volatile, не рассматриваются как отличные от базового типа с точки зрения перегрузки. Указатели на const- и volatile-объекты также не рассматриваются как отличные от указателей на базовый тип с точки зрения перегрузки. Однако механизм перегрузки может различать *ссылки*, которые имеют модификаторы const и volatile, и ссылки на базовый тип. Например, следующий код вполне допустим:

```
#include <iostream.h>
class AnyClass
{
 public:
   AnyClass()
   {cout <<"Конструктор по умолчанию"
<<"для AnyClass" << "\n";}
   AnyClass(AnyClass& ob)
 {cout <<"Конструктор копирования для " <<"AnyClass"<<"\n";}
   AnyClass(const AnyClass& ob)
    {cout <<"Конструктор копирования для "<<"AnyClass"<< "\n";}
};
  main()
{
  AnyClass  ob1;
  AnyClass ob2(ob1);
  const AnyClass ob3;
  AnyClass ob4(ob3);
}
```

Вместе с тем, на перегруженные функции накладываются несколько ограничений:

- любые две перегруженные функции должны иметь различные списки параметров;
- перегрузка функций с совпадающими списками аргументов на основе лишь типа возвращаемых ими значений недопустима;
- функции-члены не могут быть перегружены исключительно на основе того, что одна из них является статической, а другая – нет;

- typedef-определения не влияют на механизм перегрузки, так как они не вводят новых типов данных, а определяют лишь синонимы для существующих типов. Например, следующее определение:

  ```
  typedef char* PSTR;
  ```

  не позволит компилятору рассматривать две приведенные ниже функции

  ```
  void SetVal(char* sz);
  ```

  и

  ```
  void SetVal(PSTR sz);
  ```

  как различные. Поэтому их одновременное объявление в классе вызовет ошибку;
- все enum-типы данных рассматриваются как различные и могут использоваться для различения перегруженных функций;
- типы «массив (чего-то)» и «указатель (на что-то)» рассматриваются как идентичные с точки зрения перегрузки. Это верно только для одномерных массивов. Поэтому в случае перегрузки следующих функций будет выдана ошибка:

  ```
  void SetVal(char  sx);
  void SetVal(char* psz);
  ```

Для многомерных массивов вторая и последующие размерности рассматриваются как часть типа данных. Поэтому они могут использоваться для различения перегруженных функций. Например, вполне допустимы следующие определения:

```
void SetVal(char szStr[]);
void SetVal(char szStr[][4]);
```

При объявлении перегруженных функций компилятор отслеживает объявления, данные в пределах одной и той же области видимости. Поэтому, если в производном классе объявлена функция с тем же именем, что и функция в базовом классе, функция из производного класса скрывает функцию базового класса вместо того, чтобы осуществлять перегрузку (так как их области видимости различны). Короче говоря, нельзя путать переопределение функций с перегрузкой. Точно так же компилятор отслеживает и другие области видимости. Поэтому, так как функция, объявленная в области видимости файла, находится не в той же области видимости, что и функция, объявленная в блоке, даже если они имеют одинаковые имена, компилятор не рассматривает их как перегруженные. Локально объявленная функция просто скрывает глобально объявленную. Например:

```
#include <iostream.h>
void func(int i)
{
   cout << "Вызов глобально объявленной " <<"функции:"
        << i << endl;
}
void func(char* str)
{
 cout <<"Вызов локально объявленной " << "функции:"
     << str << endl;
}
main()
{
extern void func(char*);
 //Следующий вызов функции вызовет ошибку
 func(100);
 //А это – верно
 func("переопределение!");
 return 0;
}
```

Перегруженные функции-члены могут быть объявлены с различными спецификаторами доступа. Поскольку область видимости у них по-прежнему одна, компилятор рассматривает функции-члены класса с одинаковыми именами, даже если они объявлены в разделах класса с разными спецификаторами доступа, как перегруженные. Например:

```
class AnyClass
{
 public:
   AnyClass();
   double MembFunc(double, char*);
 private:
   int MembFunc(int);
};

double
AnyClass:: MembFunc(double, char*)
{
 //Тело функции
}
int
AnyClass:: MembFunc(int)
{
 //Тело функции
}
```

```
main()
{
AnyClass* pCl = new AnyClass;
//Следующая инструкция вызовет ошибку,
//т.к. она содержит вызов private-функции
pCl-> MembFunc(104);
//А здесь – все верно, т.к. вызывается public-функция
pCl-> MembFunc(104.22, "стрpковый аргумент");
delete pCl;
return 0;
}
```

Концептуально можно перегружать и статические функции-члены класса, хотя это не имеет большого смысла. Ниже приведен пример перегруженной статической функции-члена.

```
class AnyClass
{
public:
static void SetVal(int);
static void SetVal(double);
      private:
static int x;
static double y;
};
int   AnyClass::x = 0;
double AnyClass::y = 0;
void  AnyClass::SetVal(int _x){x = _x;}
void  AnyClass::SetVal(double _y){y = _y;}
main()
{
AnyClass::SetVal(10);
AnyClass::SetVal(10.4567);
return 0;
}
```

Тема 14.3
Перегрузка конструкторов

Чаще всего перегрузка применяется при создании перегруженных конструкторов (перегружать деструктор нельзя!). Целью этой перегрузки является желание предоставить пользователю как можно больше вариантов создания представителей класса. На самом деле, мы уже неоднократно встречались с перегрузкой конструкторов, хотя и не говорили об этом. Если класс предо-

ставляет конструктор с параметрами и конструктор по умолчанию, мы уже имеем дело с перегрузкой конструкторов. Как мы уже знаем, конструктор по умолчанию необходим при выделении динамической памяти массиву объектов (ибо динамический массив объектов не может быть инициализирован).

Другой случай, когда возникает необходимость в перегрузке конструкторов, – желание обеспечить пользователя *конструкторами преобразования типа*. Рассмотрим функцию, которой передается аргумент, являющийся ссылкой на объект класса AnyClass:

```
void func(AnyClass& rOb);
```

Любая попытка вызвать эту функцию с аргументом другого типа приведет к ошибке компиляции, потому что компилятор не знает, как осуществить нужное преобразование типа. Чтобы указать компилятору, как осуществить преобразование какого-то типа T в тип данных AnyClass, нужно просто задать в соответствующем классе конструктор, принимающий единственный параметр типа T. При этом будет использовано уже отмечавшееся нами ранее свойство компилятора трактовать конструкцию

```
AnyClass ob(t);
```

как

```
AnyClass ob = AnyClass(t);
```

если в классе определен конструктор

```
AnyClass(T t);
```

В рассматриваемом случае это означает, что вызов функции с аргументом t типа T, то есть

```
func(t);
```

трактуется как

```
func(AnyClass(t));
```

При этом осуществляется неявное преобразование типа данных. Неявное потому, что исходный код не вызывает явно конструктор для преобразования. С этой целью компилятор каждый раз будет создавать временный объект класса AnyClass, которому и будет передавать параметр, переданный в функцию.

Следующий пример это демонстрирует.

```
class AnyClass
{
  int x;
```

```
 public:
  //Конструктор по умолчанию
  AnyClass();
  //Конструктор преобразования типа int
  AnyClass(int);
  //Конструктор преобразования типа long
  AnyClass(long);
  //Конструктор преобразования типа double
  AnyClass(double);
};
AnyClass:: AnyClass() {x = 0;}
AnyClass:: AnyClass(int _x) {x = _x;}
AnyClass:: AnyClass(long _x) {x = int(_x);}
AnyClass:: AnyClass(double _x) {x = _x;}
void func(AnyClass& rOb)
{
 //Тело функции
}
main()
{
 func(10);  //Преобразование int в AnyClass
 func(10L); //Преобразование long в AnyClass
 func(10.0); //Преобразование double в AnyClass
 AnyClass ob;
 func(ob);  //Преобразование не нужно
 return 0;
}
```

Этот же способ преобразования типа данных может быть использован и для преобразования из одного класса в другой. Приведенный ниже пример демонстрирует конструктор преобразования из одного класса в другой.

```
class AnyClass
{
 int x;
 public:
  AnyClass(){x = 0;}
  int GetX(){return x;}
};
class OtherClass
{
 int z;
 public:
  OtherClass(){z = 0;}  //Конструктор по умолчанию
  OtherClass(AnyClass& cl);  //Конструктор преобразования
};
```

```
//Определение конструктора преобразования
OtherClass::OtherClass(AnyClass& cl)
{
 z = cl.GetX();
}
void func(OtherClass& Ob)
{
 //Тело функции
}
main()
{
 AnyClass ob;
func(ob); //Преобразование AnyClass в Othlass
 return 0;
}
```

Несмотря на то что данный пример работает, преобразование типа, которое он осуществляет, не имеет большого смысла. Ответственность за осуществляемое преобразование типа, как всегда, лежит на программисте.

Определенные пользователем преобразования типа применяются компилятором в следующих ситуациях:

- при инициализации объектов;
- при вызове функций;
- при возврате функцией значений.

Разумеется, их можно применять и для явного преобразования типа. Например:

```
main()
{
 AnyClass ob;
 //Явное преобразование типа
 OtherClass newOb = (OtherClass) ob;
}
```

Рассмотрим теперь пример, который демонстрирует пользу перегрузки другого конструктора – конструктора копирования. Пусть дан класс, который предназначен для использования в программе прямоугольных областей:

```
class Rect
{
 public:
   Rect();
   Rect(int, int);
   Rect(int, int, int, int);
```

```
    Rect(const Rect&);
    Rect(const Rect&, int =0, int =0);
  private:
    int x, y, w, h;
};
Rect:: Rect()
{
  x = y = w = h = 0;
}
Rect:: Rect(int _x, int _y)
{
 x = _x;
 y = _y;
 w = h = 100;
}
Rect:: Rect(int _x, int _y, int _w, int _h)
{
 x = _x;
 y = _y;
 w = _w;
 h = _h;
}
Rect:: Rect(const Rect& rc)
{
 x = rc.x;
 y = rc.y;
 w = rc.w;
 h = rc.h;
}
Rect:: Rect(const Rect& rc, int _x, int _y)
{
 x = _x;
 y = _y;
 w = rc.w;
 h = rc.h;
}
main()
{
 Rect rc(5,10,10,200);
 Rect newrc(rc, 15, 40);
 return 0;
}
```

Данный класс содержит несколько конструкторов, которые должны предоставить пользователю класса достаточную гиб-

кость в создании нужных ему прямоугольных областей. Среди прочих данный класс содержит перегруженный конструктор копирования

```
Rect(const Rect&, int =0, int =0);
```

Приведенный код перегруженного конструктора копирования копирует некоторые из данных-членов объекта, указанного ссылкой, но какие-то данные-члены нового объекта инициализирует по-другому. Данный конструктор можно квалифицировать как конструктор копирования благодаря тому, что первый его параметр представляет собой (константную) ссылку на объект класса, а остальные – это аргументы по умолчанию. Таким образом, вполне возможно создавать конструкторы копирования, содержащие дополнительные параметры. Это предоставляет чрезвычайную гибкость при создании разнообразных конструкторов копирования.

Тема 14.4
Создание и использование конструкторов копирования

Ранее мы уже встречались с этим понятием. Здесь мы подробно рассмотрим, как и зачем используются конструкторы копирования. Вначале рассмотрим вопрос: зачем понадобилось вводить такое понятие, как конструктор копирования.

В случае, если в функцию в качестве аргумента передается объект (то есть происходит передача аргумента по значению), делается точная (или побитовая) копия этого объекта, которая и передается параметру функции. Однако, если объект содержит указатель на выделенную область памяти, то в копии указатель будет указывать на *ту же самую* область памяти, на которую указывает исходный указатель. Поэтому, если в функции будет использован указатель из копии, будет изменена область памяти, связанная с *исходным* объектом. Кроме того, когда функция завершается, вызывается деструктор, который уничтожает копию объекта. Это опять приведет к изменению содержимого области памяти, связанной с исходным объектом.

Если объект является возвращаемым значением функции, то перед возвратом из функции компилятор генерирует временный объект для хранения, возвращаемого функцией значения. Когда происходит возврат из функции, этот временный объект выходит

из области видимости. Поэтому компилятор вызывает для него деструктор. Однако если деструктор вызывает освобождение динамической области памяти, выделенной самому объекту или одному из содержащихся в нем объектов, это приведет к ошибке.

В основе этих проблем лежит использование сгенерированного компилятором конструктора копирования по умолчанию. В том случае, если программист предоставляет конструктор копирования, именно он будет использоваться компилятором при инициализации одного объекта другим (и в частности, при передаче объекта в функцию).

Следует отличать конструктор копирования от оператора присваивания. Например:

```
AnyClass ob1;
AnyClass func();//объявление функции

//ob2 инициализируется ob1
//с помощью конструктора копирования
AnyClass ob2(ob1);

//Следующий оператор эквивалентен предыдущему
AnyClass ob2 = ob1;

//Возврат объекта из функции с помощью
//оператора копирования
ob1 = func();

//Напротив, следующая инструкция использует
//оператор присваивания
ob2 = ob1;
```

Таким образом, несмотря на наличие в инструкции знака присваивания, она не всегда означает выполнение присваивания. Инструкция

```
AnyClass ob2 = ob1;
```

представляет собой просто другую форму записи конструктора копирования.

Тема 14.5
Устаревшее ключевое слово overload

В старых версиях C++ для создания перегруженных функций требовалось ключевое слово overload. Синтаксис его использования был следующим:

```
overload <имя_функции>;
```

Упомянутая в overload функция должна была объявляться до использования этого оператора. Таким образом, ключевое слово overload сообщало компилятору о намерении перегрузить упомянутую после него функцию. Хотя ключевое слово overload в современных компиляторах больше не поддерживается, его все еще можно встретить в старых программах, поэтому полезно знать, как и для чего оно использовалось.

Тема 14.6
Перегрузка и неоднозначность. Ключевое слово explicit

Перегруженные функции выбираются по принципу наилучшего соответствия объявлений функций в текущей области видимости аргументам, предоставленным в вызове функции. Если подходящая функция найдена, эта функция и вызывается. Слово «подходящая» в этом контексте означает одно из следующего (в порядке ухудшения соответствия):

- найдено точное соответствие;
- выполнено тривиальное преобразование;
- выполнено преобразование целочисленных типов;
- существует стандартное преобразование к желаемому типу аргумента;
- существует определенное программистом преобразование (оператор преобразования или конструктор) к требуемому типу аргумента;
- были найдены аргументы, представленные многоточием.

Компилятор создает совокупность функций-кандидатов на соответствие для каждого аргумента. Затем для каждого аргумента строится совокупность функций, соответствующих наилучшим образом. Наконец, определяется функция, которая больше всего соответствует вызову, как пересечение всех этих совокупностей. Если это пересечение содержит несколько функций, – *перегрузка неоднозначна* и компилятор генерирует сообщение об ошибке.

Еще одно важное замечание. Неоднозначность перегруженных функций не может быть определена, пока не встретится вызов функции.

Функция с n аргументами по умолчанию с точки зрения соответствия аргументов рассматривается как совокупность из n+1 функций, каждая из которых отличается от предыдущей заданием

одного дополнительного аргумента. Многоточие (...) действует как произвольный символ: оно соответствует любому заданному аргументу. Это может служить источником неоднозначности при выборе перегруженной функции.

Все сказанное выше относилось ко всем перегруженным функциям, безотносительно к тому, являются ли они функциями-членами или нет. Рассмотрим теперь специфику перегруженных функций-членов.

Функции-члены класса рассматриваются различным образом в зависимости от того, объявлены ли они статическими или нет, потому что нестатические функции имеют неявный аргумент, через который передается указатель this. При определении функции, которая наилучшим образом соответствует вызову, для нестатических функций-членов рассматриваются только те перегруженные функции-члены, у которых скрытый указатель this соответствует типу объекта, который передан функции при вызове. В отличие от других аргументов при попытке установить соответствие аргумента с указателем this никакие преобразования не производятся.

В отношении перегруженных конструкторов язык C++ предоставляет дополнительную возможность по управлению процессом поиска соответствия вызываемого конструктора. Ключевое слово explicit представляет собой спецификатор объявления, который может применяться только в *объявлениях* конструкторов (но не в *определениях* конструкторов вне класса). Например:

```
class T
{
 public:
  explicit T(int);
  explicit T(double)
  {
   // Тело конструктора
  }
};
T::T(int)
{
 // Тело конструктора
}
```

Конструктор, объявленный со спецификатором explicit, не принимает участия в неявных преобразованиях типа. Он может быть использован только в том случае, когда никакого преобразования типа аргументов не требуется. Например, для класса, объявленного выше:

```
//Определение функции
void f(T){};

void g(int i)
{
 //Следующая инструкция вызовет ошибку, т.к. неявное
 // преобразование типа int в тип T запрещено
 f(i);
}

void h()
{
 T ob(10);// А это – верно.
}
```

Замечание. Нет смысла применять ключевое слово explicit к конструкторам с несколькими параметрами, так как такие конструкторы не принимают участия в неявных преобразованиях типа.

Тема 14.7
Определение адреса перегруженной функции

Рассмотрим теперь, как найти адрес перегруженной функции. Адрес функции можно найти, если поместить имя этой функции без аргументов и круглых скобок в правой части оператора присваивания. В левой части оператора присваивания должен находиться указатель на функцию соответствующего типа. Например:

```
//Первая версия функции
int Func(int i, int j);
int Func(long l); //Вторая версия функции
   ....
//Определение адреса первой версии функции
int (*pFunc)(int, int) = Func;
//Вызов функции
pFunc(10,20);
```

Компилятор определяет, какую версию функции нужно выбрать путем отыскания функции, которая в точности соответствует заданному указателем типу. Если точное соответствие не найдено, выражение, которое получает адрес функции, неоднозначно. В этом случае генерируется ошибка.

Выше мы рассмотрели случай, когда перегружалась глобальная функция, однако те же правила применимы и к перегруженным функциям-членам.

Практикум «Перегрузка функций»

Упражнение 14.1
Применение перегрузки функций

Проанализируйте предлагаемый ниже пример перегрузки функций-членов некоторого класса.

Пусть имеется некоторый класс MyClass, содержащий два закрытых целочисленных члена x и y. Функции GetX() и GetY() возвращают их значения, в то время как запись новых значений осуществляется с помощью функции SetVar. Если данной функции передается один целочисленный аргумент, выполняется функция SetVar(int _x), а если передается два параметра – функция SetVar(int _x, int _y).

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
   int x;
   int y;
public:
   MyClass(){x = 0; y = 0;}
   ~MyClass(){}
   void SetVar(int _x){x = _x;}
   void SetVar(int _x, int _y){x = _x; y = _y;}
   int GetX(){return x;}
   int GetY(){return y;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include <iostream.h>
#include "header.h"
void main()
{
   MyClass obj;
   obj.SetVar(10, 15);
   cout << obj.GetX() << ' ' << obj.GetY() << endl;
   obj.SetVar(11);
   cout << obj.GetX() << ' ' << obj.GetY() << endl;
}
```

Упражнение 14.2
Конструктор и его перегрузка

Проанализируйте пример перегрузки конструктора. Экземпляр класса MyClass может быть сконструирован с помощью конструктора по умолчанию MyClass(), конструктора с одним целочисленным параметром MyClass(int _x) или с двумя целочисленными аргументами MyClass(int _x, int _y). При этом закрытые члены-данные класса x и y заполняются соответствующими значениями.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
   int x;
   int y;
public:
   MyClass(){x = 0; y = 0;}
   MyClass(int _x){x = _x;}
   MyClass(int _x, int _y){x = _x; y = _y;}
   ~MyClass(){}
   void SetX(int _x){x = _x;}
   void SetY(int _y){y = _y;}
   int GetX(){return x;}
   int GetY(){return y;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include <iostream.h>
#include "header.h"
void main()
{
   MyClass obj(10, 12);
   cout << obj.GetX() << ' ' << obj.GetY() << endl;
   MyClass obj1(11); obj1.SetY(13);
   cout << obj1.GetX() << ' ' << obj1.GetY() << endl;
}
```

Упражнение 14.3
Конструкторы копирования

Проанализируйте предлагаемый ниже пример.

Пусть имеется некоторый класс Coord с конструктором, заполняющим два закрытых целочисленных данных-члена этого

класса. Создадим конструктор копирования, в котором инициализируются данные. Благодаря разработанному конструктору копирования экземпляр класса ob3 при создании инициализируется теми же параметрами, что и объект ob1.

```
class Coord
{
   int x, y;
public:
   Coord(int _x, int _y):x(_x), y(_y){};
   //Конструктор копирования
   Coord(const Coord& src);
};
Coord::Coord(const Coord& src)
{
   x = src.x;
   y = src.y;
}
int main()
{
   Coord ob1(2,9);
   Coord ob2 = ob1;
   Coord ob3(ob1);
   return 0;
}
```

Упражнение 14.4
Перегруженные функции

Дополните рассматриваемый ниже пример таким образом, чтобы функция SetVal с одним параметром вызывалась посредством указателя на перегруженную функцию-член.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
   int x;
   float fl;
public:
   MyClass(){x = 0; fl = 0.;}
   ~MyClass(){}
   void SetVal(int _x){x = _x;}
   void SetVal(int _x, float _fl){x = _x; fl = _fl;}
```

```
    void GetVal(int& _x, float& _fl){_x = x; _fl = fl;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include <iostream.h>
#include "header.h"
int main()
{
    MyClass obj;
    obj.SetVal(10, 12.23);
    int a; float b;
    obj.GetVal(a, b);
    cout << a << ' ' << b << endl;
    return 0;
}
```

Раздел 15
Перегрузка операторов

Тема 15.1
Понятие перегрузки операторов

В C++ имеется возможность перегрузки большинства встроенных операторов. Операторы могут быть перегружены глобально или в пределах класса. Перегруженные операторы реализуются как функции с помощью ключевого слова operator. Имя перегруженной функции должно быть operatorX, где X – это перегруженный оператор. Ключевое слово operator, за которым следует символ оператора, называется *именем операторной функции*. Операторы, которые можно перегружать, приведены в табл. 15.1. Например, чтобы перегрузить оператор сложения, нужно определить функцию с именем operator+, а чтобы перегрузить оператор сложения с присваиванием, нужно определить функцию operator+=. Обычно компилятор вызывает эти функции неявно, когда перегруженные операторы встречаются в коде программы. Тем не менее, их можно вызывать и непосредственно. Например:

```
Point pt1, pt2, pt3;
//Вызов перегруженного оператора сложения
pt3 = pt1.operator+(pt2);
```

Таблица 15.1
Перегружаемые операторы

Оператор	Название	Тип
,	Запятая	Бинарный
!	Логическое НЕ	Унарный
!=	Не равно	Бинарный
%	Модуль	Бинарный

Оператор	Название	Тип
%=	Модуль с присваиванием	Бинарный
&	Побитовое И	Бинарный
&	Адрес	Унарный
&&	Логическое И	Бинарный
&=	Побитовое И с присваиванием	Бинарный
( )	Вызов функции	—
*	Умножение	Бинарный
*	Разыменование	Унарный
*=	Умножение с присваиванием	Бинарный
+	Сложение	Бинарный
+	Унарный плюс	Унарный
++	Инкремент	Унарный
+=	Сложение с присваиванием	Унарный
-	Вычитание	Унарный
-	Унарный минус	Унарный
--	Декремент	Унарный
-=	Вычитание с присваиванием	Бинарный
->	Выбор члена	Бинарный
->*	Выбор члена по указателю	Бинарный
/	Деление	Бинарный
/=	Деление с присваиванием	Бинарный
<	Меньше	Бинарный
<<	Сдвиг влево	Бинарный
<<=	Сдвиг влево с присваиванием	Бинарный
<=	Меньше или равно	Бинарный

Оператор	Название	Тип
=	Присваивание	Бинарный
==	Равно	Бинарный
>	Больше	Бинарный
>=	Больше или равно	Бинарный
>>	Сдвиг вправо	Бинарный
>>=	Сдвиг вправо с присваиванием	Бинарный
[ ]	Индекс массива	—
^	Побитовое исключающее ИЛИ	Бинарный
^=	Побитовое исключающее ИЛИ с присваиванием	Бинарный
|	Побитовое ИЛИ	Бинарный
|=	Побитовое ИЛИ с присваиванием	Бинарный
||	Логическое ИЛИ	Бинарный
~	Дополнение до единицы	Унарный
delete	Удаление	—
new	Создание нового элемента	—

Существуют две версии унарных операторов инкремента и декремента: префиксная и постфиксная.

Операторы, перечисленные в табл. 15.2, не могут быть перегружены.

Таблица 15.2
Неперегружаемые операторы

Оператор	Название
.	Выбор члена
.*	Выбор члена по указателю
::	Оператор расширения области видимости
? :	Оператор условия

Оператор	Название
#	Препроцессорный символ
##	Препроцессорный символ

Операторные функции перегруженных операторов, за исключением new и delete, должны подчиняться следующим правилам:

- операторная функция должна быть либо нестатической функцией-членом класса, либо принимать аргумент типа класса или перечислимого типа, или аргумент, который является ссылкой на тип класса или перечислимый тип.

```
class Point
{
 public:
  //Объявление перегруженной операторной
  //функции-члена для оператора "меньше".
  Point operator<(Point&);
  ...
  //Объявления операторов сложения.
  friend Point operator+(Point&, int);
  friend Point operator+(int, Point&);
};
```

В предыдущем примере оператор «меньше» объявляется как функция-член класса, а операторы сложения объявляются как глобальные функции, которые имеют дружественный доступ к членам класса. В частности, этот пример демонстрирует, что для одного оператора может быть предусмотрено несколько реализаций:

- операторная функция не может изменять число аргументов или приоритеты операций и их порядок выполнения по сравнению с использованием соответствующего оператора для встроенных типов данных;
- операторная функция унарного оператора, объявленная как функция-член, не должна иметь параметров; если же она объявлена как глобальная функция, она должна иметь один параметр;
- операторная функция бинарного оператора, объявленная как функция-член, должна иметь один параметр; если же она объявлена как глобальная функция, она должна иметь два параметра;
- операторная функция не может иметь параметров по умолчанию;

- первый параметр (если он есть) операторной функции, обявленной как функция-член, должен иметь тип класса объекта, для которого вызывается соответствующий оператор; никакие преобразования типа для первого аргумента не выполняются;
- за исключением операторной функции оператора присваивания все операторные функции класса наследуются его производными классами;
- операторные функции =, (), [] и -> должны быть нестатическими функциями-членами (и не могут быть глобальными функциями).

Правила для операторов new и delete будут рассмотрены отдельно.

Заметим, что с помощью перегрузки можно совершенно изменить смысл оператора для некоторого класса. Однако рекомендуется не делать этого без веских на то причин, так как это может сильно затруднить понимание кода программы. Перегруженный оператор, как правило, должен следовать семантике его поведения для встроенных типов данных.

Тема 15.2
Перегрузка бинарных операторов

Когда операторная функция бинарного оператора объявляется как нестатическая функция-член, она должна быть объявлена в виде:

```
<возвр_тип> operatorX(<тип_пар> par);
```

где <возвр_тип> – тип возвращаемого функцией значения, X – перегружаемый оператор, <тип_пар> – тип параметра и par – сам параметр.

В этот параметр будет передан тот объект, который стоит *справа* от оператора. Объект, стоящий *слева* от оператора, передается неявно с помощью указателя this.

Когда операторная функция бинарного оператора объявляется как глобальная, она должна быть объявлена в виде:

```
<возвр_тип> operatorX(<тип_пар1> par1, <тип_пар2> par2);
```

где <возвр_тип> и X – те же, что и выше, а <тип_пар1>, <тип_пар2> – типы параметров и par1, par2 – сами параметры.

По крайней мере один из этих параметров должен иметь тип класса, для которого перегружается оператор.

Хотя нет никаких ограничений на тип возвращаемого значения бинарных операторов, большинство определяемых пользователем операторов возвращают либо тип класса, либо ссылку на тип класса.

Здесь мы приведем несколько примеров написания операторных функций. Не следует их рассматривать как исчерпывающие все возможные варианты, однако они иллюстрируют несколько наиболее общих приемов.

Вначале рассмотрим случай, когда операторная функция определяется как функция-член класса. В следующей программе перегружаются операторы сложения и вычитания для класса Coord:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(){x = 0; y = 0;}
  Coord(int _x, int _y){x = _x; y = _y;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  Coord operator+(Coord& ob);
  Coord operator-(Coord& ob);
};
Coord
Coord::operator+(Coord& ob)
{
 Coord tempOb;
 tempOb.x = x + ob.x;
 tempOb.y = y + ob.y;
 return tempOb;
}
Coord
Coord::operator-(Coord& ob)
{
 Coord tempOb;
 tempOb.x = x - ob.x;
 tempOb.y = y - ob.y;
 return tempOb;
}
main()
{
 int x,y;
```

```
  Coord PtA(10,20), PtB(3,8), PtC;
  //Вызов перегруженного оператора сложения
  PtC = PtA + PtB;
  PtC.GetCoord(x,y);
  cout << "PtC.x =" << x
    << " PtC.y =" << y << endl;
  //Вызов перегруженного оператора вычитания
  PtC = PtA - PtB;
  PtC.GetCoord(x,y);
  cout << "PtC.x =" << x
    << " PtC.y =" << y << endl;
  return 0;
}
```

При выполнении программа выводит на экран

```
PtC.x=13 PtC.y=28
PtC.x=7 PtC.y=12
```

Несколько замечаний по поводу этого примера. При передаче аргумента в операторные функции здесь использована передача по ссылке. Это повышает эффективность работы программы, поскольку такой аргумент не копируется при передаче; в функцию фактически передается указатель на объект. Другой довод в пользу использования ссылки в качестве параметра – она позволяет избежать вызова деструктора для копии объекта, созданной при передаче объекта-параметра. Конечно, с таким же успехом можно было бы передавать в функцию сам объект, однако это менее эффективно. Хотя нет никаких требований к возвращаемому операторной функцией значению, здесь обе операторные функции возвращают объект, который имеет тип класса. Смысл этого состоит в том, что это позволяет использовать результат сложения двух объектов в сложном выражении. Например, становятся вполне допустимыми следующие выражения:

```
PtC = PtA + PtB - PtC;
```

С другой стороны, допустимы выражения вида:

```
(PtA + PtB).GetCoord(x, y);
```

Здесь для вызова функции-члена GetCoord(x, y) используется временный объект, возвращаемый оператором сложения.

Следующий пример демонстрирует возможность использования в качестве параметра операторной функции-члена перечислимого типа данных. Надо только не забывать, что этот параметр представляет *правый* операнд оператора.

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(){x = 0; y = 0;}
  Coord(int _x, int _y){x = _x; y = _y;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  Coord operator+(Coord& ob);
  Coord operator+(int n);
};
Coord
Coord::operator+(Coord& ob)
{
 Coord tempOb;
 tempOb.x = x + ob.x;
 tempOb.y = y + ob.y;
 return tempOb;
}
Coord
Coord::operator+(int n)
{
 Coord tempOb;
 tempOb.x = x + n;
 tempOb.y = y + n;
 return tempOb;
}
main()
{
 int x,y;
 Coord PtA(10,20), PtB(3,8), PtC;
 //Вызов перегруженного оператора сложения
 PtC = PtA + PtB;
 PtC.GetCoord(x,y);
 cout << "PtC.x =" << x << " PtC.y =" << y << endl;
 //Вызов перегруженного оператора сложения
 PtC = PtA + 30;
 PtC.GetCoord(x,y);
 cout << "(PtA+100).x =" << x <<" (PtA+100).y ="<<y<< endl;
return 0;
}
```

При выполнении программа выводит на экран

```
PtC.x=13 PtC.y=28
(PtA+30).x=40 (PtA+30).y=50
```

Согласно сделанному выше замечанию, компилятор поймет, что означает инструкция

```
PtC = PtA + 30;
```

Однако для него останется совершенно непонятной инструкция

```
PtC = 30 + PtA;
```

Более того, нет никакого способа дать разумное определение этой инструкции, ограничиваясь средствами операторной функции-члена. Путь для решения такой задачи открывает использование дружественной операторной функции.

Рассмотрим пример использования дружественной операторной функции для перегрузки оператора сложения класса Coord. Как известно, дружественной функции не передается скрытый указатель this. Поэтому в случае перегрузки бинарного оператора ей приходится передавать два аргумента, первый из которых соответствует левому операнду, а второй – правому.

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(){x = 0; y = 0;}
  Coord(int _x, int _y){x = _x; y = _y;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  friend Coord operator+(Coord& ob1, Coord& ob2);
  friend Coord operator+(Coord& ob, int n);
  friend Coord operator+(int n, Coord& ob);
};
//Перегрузка оператора сложения объектов
//с помощью дружественной функции
Coord operator+(Coord& ob1, Coord& ob2)
{
 Coord tempOb;
 tempOb.x = ob1.x + ob2.x;
 tempOb.y = ob1.y + ob2.y;
 return tempOb;
}
//Перегрузка оператора сложения ob + int
//с помощью дружественной функции
Coord operator+(Coord& ob, int n)
{
 Coord tempOb;
 tempOb.x = ob.x + n;
 tempOb.y = ob.y + n;
 return tempOb;
}
```

```
//Перегрузка оператора сложения int + ob
//с помощью дружественной функции
Coord operator+(int n, Coord& ob)
{
 Coord tempOb;
 tempOb.x = ob.x + n;
 tempOb.y = ob.y + n;
 return tempOb;
}
main()
{
 int x,y;
 Coord PtA(10,20), PtB(3,8), PtC;
 //Вызов перегруженного оператора сложения
 PtC = PtA + PtB;
 PtC.GetCoord(x,y);
 cout << "PtC.x =" << x << " PtC.y =" << y << endl;
 //Вызов перегруженного оператора сложения для ob + int
 PtC = PtA + 30;
 PtC.GetCoord(x,y);
 cout << "(PtA+100).x =" << x << " (PtA+100).y =" << y << endl;
 //Вызов перегруженного оператора сложения для int + ob
 PtC = 30 + PtA;
 PtC.GetCoord(x,y);
 cout << "(100+PtA).x =" << x << " (100+PtA).y =" << y << endl;
 return 0;
}
```

При выполнении программа выводит на экран

```
PtC.x=13 PtC.y=28
(PtA+30).x=40 (PtA+30).y=50
(30+PtA).x=40 (30+PtA).y=50
```

Таким образом, в результате перегрузки дружественных функций обе инструкции:

```
PtC = PtA + 30;
PtC = 30 + PtA;
```

становятся правильными.

Тема 15.3
Перегрузка операторов отношения и логических операторов

Несмотря на то что операторы отношения и логические операторы являются бинарными, мы рассмотрим их здесь отдельно. При перегрузке этих операторов соответствующие операторные

функции не должны возвращать объект класса, для которого они определены. Вместо этого они должны возвращать значение булевского типа (или целого типа, интерпретируемого как булевские значения true и false). Рассмотрим пример, в котором перегружаются операторы == и &&:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
  Coord(){x = 0; y = 0;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  bool operator==(Coord ob);
  bool operator&&(Coord ob);
};
bool
Coord::operator==(Coord ob)
{
 return (x == ob.x && y == ob.y);
}
bool
Coord::operator&&(Coord ob)
{
 return (x && ob.x) && (y && ob.y);
}
int main()
{
 Coord Pt1(10,20), Pt2(10,25),
   Pt3(10,20), Pt4;
 if (Pt1==Pt2)
      cout << "Pt1 равна Pt2\n";
 else cout << "Pt1 не равна Pt2\n";
 if (Pt1==Pt3)
      cout << "Pt1 равна Pt3\n";
 else
      cout << "Pt1 не равна Pt3\n";
 if (Pt1 && Pt3)
cout << "Pt1 && Pt3 истинно\n";
 else
      cout << "Pt1 && Pt2 ложно\n";
 if (Pt1 && Pt4)
      cout << "Pt1 && Pt4 истинно\n";
 else
      cout << "Pt1 && Pt4 ложно\n";
 return 0;
}
```

При выполнении программа выводит на экран:

```
Pt1 не равна Pt2
Pt1 равна Pt3
Pt1 && Pt3 истинно
Pt1 && Pt4 ложно
```

Пользуясь определениями операций == и &&, легко определить, что программа верно вычислила отношения между точками с заданными координатами.

Тема 15.4
Перегрузка оператора присваивания

Оператор присваивания также является бинарным, однако процедура его перегрузки имеет ряд особенностей:

- операторная функция оператора присваивания не может быть объявлена как глобальная функция (функция, не являющаяся членом класса);
- операторная функция оператора присваивания не наследуется производным классом;
- компилятор может сгенерировать операторную функцию оператора присваивания, если она не определена в классе.

Оператор присваивания по умолчанию, сгенерированный компилятором, выполняет почленное присваивание нестатических членов класса. Здесь та же ситуация, что и с генерируемым компилятором конструктором копирования по умолчанию: результат окажется непригодным для использования, если класс содержит указатели.

Необходимо отметить, что левый операнд после выполнения оператора присваивания меняется, так как ему присваивается новое значение. Поэтому функция оператора присваивания *должна* возвращать ссылку на объект, для которого она вызвана (и которому присваивается значение), если мы хотим сохранить семантику оператора присваивания для встроенных типов данных. Проще всего это сделать, возвратив разыменованный указатель this. Это, в частности, позволяет использовать перегруженный оператор присваивания в выражениях следующего вида:

```
Pt3 = Pt2 = Pt1;
```

Следующий пример иллюстрирует, как осуществляется перегрузка оператора присваивания:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
   void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  Coord& operator=(Coord& rhs);
  bool operator==(Coord rhs);
};
Coord&
Coord::operator=(Coord& rhs)
{
 //Проверка на присваивание самому себе
 if (this == &rhs)
     return *this;
 x = rhs.x;
 y = rhs.y;
 return *this;//Возвратить левую часть
}
 bool
Coord::operator==(Coord rhs)
{
 return (x == rhs.x && y == rhs.y);
}
int main()
{
 Coord Pt1(10,20), Pt2(10,25), Pt3(10,20);
 Pt2 = Pt1;
  if (Pt1==Pt2)
 cout << "Pt1 равна Pt2\n";
 else
       cout << "Pt1 не равна Pt2\n";
 Pt3 = Pt2 = Pt1;
 if (Pt1==Pt3)
 cout << "Pt1 равна Pt3\n";
 else
       cout << "Pt1 не равна Pt3\n";
  return 0;
}
```

При выполнении программа выводит на экран:

```
Pt1 равна Pt2
Pt1 равна Pt3
```

Заметим, что при реализации перегруженного оператора необходимо проводить проверку на присваивание самому себе. Если в коде программы встретится строка вида

```
ob = ob;
```

или, что гораздо более вероятно, нечто вроде следующего:

```
Coord ob, *pOb;
...
ob = *pOb;
```

произойдет присваивание самому себе, связанное с ошибкой в программе.

Если не предусмотреть защиту от такой ситуации, то произойдет следующее: оператор присваивания сначала очистит все ячейки памяти объекта слева, а затем попытается присвоить этому объекту значение объекта справа, который уже не содержит реальных значений.

В приведенном выше примере функция operator=() содержит инструкцию

```
if (this == &rhs)
   return *this;
```

Здесь использовано широко распространенное обозначение операнда, расположенного в правой части оператора присваивания – rhs (от англ. *right hand side*).

Проверку на присваивание самому себе иногда осуществляют и так:

```
if (*this == rhs)
   return *this;
```

Но в этом случае соответствующий класс должен содержать перегруженный оператор сравнения, так как в операторе if теперь фигурирует сравнение объектов, а не встроенного типа «указатель».

Тема 15.5
Перегрузка унарных операторов

Унарным операторам необходим только один операнд. При перегрузке унарного оператора с помощью функции-члена этот единственный операнд представляет собой объект, для которого и выполняется операция. Поэтому, когда операторная функция унарного оператора объявляется как нестатическая функция-член, она должна быть объявлена в виде:

```
<возвр_тип> operatorX();
```

где <возвр_тип> – тип возвращаемого функцией значения, X – перегружаемый унарный оператор.

Когда операторная функция унарного оператора объявляется как глобальная, она должна быть объявлена в виде:

```
<возвр_тип> operatorX(<тип_пар> par);
```

где <возвр_тип> и X – те же, что и выше, <тип_пар> – тип параметра, а par – параметр.

Хотя в эту категорию попадают операторы инкремента и декремента, мы рассмотрим их отдельно. А здесь мы рассмотрим в качестве примера перегрузку унарного оператора «плюс» и унарного оператора «минус».

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(){x = 0; y = 0;}
  Coord(int _x, int _y){x = _x; y = _y;}
  void GetCoord(int& _x, int& _y)
{_x = x; _y = y;}
  Coord operator+();
  Coord operator-();
};
Coord
Coord::operator+()
{
 x = +x;
 y = +y;
 return *this;
}
Coord
Coord::operator-()
{
 x = -x;
 y = -y;
 return *this;
}
main()
{
 int x,y;
 Coord PtA(-10,20);
 //Вызов перегруженного унарного оператора «плюс»
 PtA = +PtA;
 PtA.GetCoord(x,y);
```

```
 cout << "PtA.x =" << x << " PtA.y =" << y << endl;
 //Вызов перегруженного унарного оператора «минус»
 PtA = -PtA;
 PtA.GetCoord(x,y);
 cout << "PtA.x =" << x << " PtA.y =" << y << endl;
return 0;
}
```

При выполнении программа выводит на экран

```
PtA.x=-10 PtA.y=20
PtA.x=10 PtA.y=-20
```

Те же операторы могут быть перегружены с помощью глобальных функций, используя следующие прототипы функций, объявленных дружественными для класса:

```
friend Coord operator+(Coord& ob);
friend Coord operator-(Coord& ob);
```

Для рассматриваемого примера реализация этих функций может быть следующей:

```
Coord operator+(Coord& ob)
{
 ob.x = +ob.x;
 ob.y = +ob.y;
 return ob;
}
Coord operator-(Coord& ob)
{
 ob.x = -ob.x;
 ob.y = -ob.y;
 return ob;
}
```

Результат работы программы с перегруженными таким образом операторами будет тем же.

Тема 15.6
Перегрузка операторов инкремента и декремента

Операторы инкремента и декремента попадают в отдельную категорию, поскольку каждый из них имеет два варианта: префиксный и постфиксный. Чтобы различить префиксный и постфиксный варианты этих операторов, используется следующее правило: *префиксная* форма объявляется в точности так же, как

любой другой унарный оператор; *постфиксная* форма принимает дополнительный аргумент типа int. На самом деле, этот аргумент обычно не используется и при передаче в функцию равен нулю (однако при желании он может быть использован). Этот параметр просто указывает компилятору, что речь идет о постфиксной форме оператора. Следующий пример показывает, как определить префиксную и постфиксную формы операторов инкремента и декремента для класса Coord:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(){x = 0; y = 0;}
  Coord(int _x, int _y){x = _x; y = _y;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
   //Префиксная форма оператора инкремента
   Coord& operator++();
   //Постфиксная форма оператора инкремента
   Coord operator++(int);
   //Префиксная форма оператора декремента
   Coord& operator--();
   //Постфиксная форма оператора декремента
   Coord operator--(int);
};
Coord&
Coord::operator++()
{
 x++;
 y++;
 return *this;
}
Coord
Coord::operator++(int)
{
 Coord temp= *this;
 ++*this;
 return temp;
}
Coord&
Coord::operator--()
{
 x--;
 y--;
 return *this;
}
```

```
Coord
Coord::operator--(int)
{
 Coord temp= *this;
 --*this;
 return temp;
}
main()
{
 int x,y;
 Coord PtA(10,20),PtB(15,25),PtC;
 //Вызов префиксной формы оператора инкремента
 ++PtA;
 PtA.GetCoord(x,y);
 cout << "(++PtA).x =" << x << " (++PtA.y =" << y << endl;
 //Вызов постфиксной формы оператора инкремента
 PtA++;
 PtA.GetCoord(x,y);
 cout << "(PtA++).x =" << x << " (PtA++).y =" << y << endl;
 //Вызов префиксной формы оператора декремента
 --PtA;
 PtA.GetCoord(x,y);
 cout << "(--PtA).x =" << x << " (--PtA).y =" << y << endl;
 //Вызов постфиксной формы оператора декремента
 PtA--;
 PtA.GetCoord(x,y);
 Cout << "(PtA--).x =" << x << " (PtA--).y =" << y << endl;
 //Проверка постфиксной формы оператора инкремента
 PtC = PtB++;
 PtC.GetCoord(x,y);
 cout << "PtC.x =" << x << " PtC.y =" << y << endl;
 PtB.GetCoord(x,y);
 cout << "PtB.x =" << x << " PtB.y =" << y << endl;
 return 0;
}
```

При выполнении программа выводит на экран:

```
(++PtA).x=11 (++PtA).y=21
(PtA++).x=12 (PtA++).y=22
(--PtA).x=11 (--PtA).y=21
(PtA--).x=10 (PtA--).y=20
PtC.x=15 PtC.y=25
PtB.x=16 PtB.y=26
```

Обратите внимание, что постфиксная форма операторной функции возвращает не ссылку, а сам объект, поскольку из функции нельзя возвращать ссылку на временный объект.

Рассмотрим теперь перегрузку операторов инкремента и декремента с помощью дружественных функций. Заметим, что при

использовании дружественной функции для перегрузки этих операторов в операторную функцию нужно передавать параметр в виде ссылки, поскольку он должен быть изменен в функции, и это измененное значение объекта должно быть возвращено.

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(){x = 0; y = 0;}
  Coord(int _x, int _y){x = _x; y = _y;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  //Префиксная форма оператора инкремента
  friend Coord& operator++(Coord&);
  //Постфиксная форма оператора инкремента
  friend Coord operator++(Coord&, int);
  //Префиксная форма оператора декремента
  friend Coord& operator--(Coord&);
  //Постфиксная форма оператора декремента
  friend Coord operator--(Coord&, int);
};
Coord& operator++(Coord& ob)
{
 ob.x++;
 ob.y++;
 return ob;
}
Coord operator++(Coord& ob, int)
{
 Coord temp = ob;
 ++ob;
 return temp;
}
Coord& operator--(Coord& ob)
{
 ob.x--;
 ob.y--;
 return ob;
}
Coord operator--(Coord& ob, int)
{
 Coord temp = ob;
 --ob;
 return temp;
}
```

```
main()
{
 int x,y;
 Coord PtA(10,20),PtB(15,25),PtC;
 //Вызов префиксной формы оператора инкремента
 ++PtA;
 PtA.GetCoord(x,y);
 cout << "(++PtA).x =" << x << " (++PtA.y =" << y << endl;
 //Вызов постфиксной формы оператора инкремента
 PtA++;
 PtA.GetCoord(x,y);
 cout << "(PtA++).x =" << x << " (PtA++).y =" << y << endl;
 //Вызов префиксной формы оператора декремента
 --PtA;
 PtA.GetCoord(x,y);
 cout << "(--PtA).x =" << x << " (--PtA).y =" << y << endl;
 //Вызов постфиксной формы оператора декремента
 PtA--;
 PtA.GetCoord(x,y);
 cout << "(PtA--).x =" << x << " (PtA--).y =" << y << endl;
 //Проверка постфиксной формы оператора инкремента
 PtC = PtB++;
 PtC.GetCoord(x,y);
 cout << "PtC.x =" << x << " PtC.y =" << y << endl;
 PtB.GetCoord(x,y);
 cout << "PtB.x =" << x << " PtB.y =" << y << endl;
 return 0;
 }
```

Эта программа выводит на экран те же результаты, что и предыдущая.

Аргумент типа int, который обозначает постфиксную форму оператора, обычно не используется для передачи аргументов, однако он может быть использован, например, следующим образом:

```
class Coord
{
  int x, y;
public:
  Coord(){x = 0; y = 0;}
  Coord(int _x, int _y){x = _x; y = _y;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  //Постфиксная форма оператора инкремента
Coord& operator++(int);
};
Coord&
Coord::operator++(int n)
{
 if (n != 0)
```

```
    {
     //Если аргумент передается
     x += n; y += n;
    }
   else
    {
     x++; y++;
    }
   return *this;
}
main()
{
 Coord Pt(10,20);
 Pt.operator++(25); //Инкремент на 25
 return 0;
}
```

Таким образом, в этом случае постфиксную форму оператора инкремента приходится вызывать, указывая непосредственно имя соответствующей операторной функции. Так приходится поступать потому, что компилятор принимает выражение

```
Pt++(25);
```

за вызов функции с именем Pt++.

Тема 15.7
Перегрузка оператора индексирования

Оператор индексирования, записываемый в виде квадратных скобок [ ], рассматривается как бинарный оператор. При перегрузке он должен быть объявлен как нестатическая функция-член класса, которая принимает один аргумент. Этот аргумент может иметь любой тип. Сам аргумент при этом трактуется как индекс в массиве объектов класса. Следующий пример это иллюстрирует:

```
#include <iostream.h>
class IntArray
{
  int MaxCount;
  int* pInt;
 public:
   IntArray(int count);
  ~IntArray(){delete pInt;}
  int& operator[](int index);
};
IntArray:: IntArray(int count)
{
```

```
 pInt = new int[count];
 MaxCount = count;
}
int&
IntArray::operator[](int index)
{
 static int iErr=-1;
 if (index >= 0 && index < MaxCount)
    return pInt[index];
 else
  {
   cout << "Ошибка: выход за " << "границы массива!" << endl;
   return iErr;
  }
}
main()
{
 IntArray iArr(5);
 for(int i=0; i<5; i++)
   iArr[i] = i;
 for (int i=0; i<=5; i++)
    cout << "iArr[" << i <<"]=" << iArr[i] << endl;
 return 0;
}
```

Здесь реализуется целочисленный массив, в котором производится проверка границ. Если заданный индекс массива выходит за пределы допустимых, выдается сообщение об ошибке. При выполнении программа выводит на экран:

```
iArr[0]=0
iArr[1]=1
iArr[2]=2
iArr[3]=3
iArr[4]=4
Ошибка: выход за границы массива!
iArr[5]=-1
```

Заметим, что функция operator[] возвращает ссылку. Это позволяет использовать индексированные выражения с любой стороны от знака присваивания.

Тема 15.8
Перегрузка оператора вызова функции

Оператор вызова функции, вызываемый с помощью круглых скобок, трактуется как бинарный. Синтаксис вызова функции следующий:

<выражение>(<список_выражений>)

Здесь <выражение> – это первый операнд, а <список_выражений> – это второй (необязательный) операнд. Оператор вызова функции должен быть объявлен как нестатическая функция-член класса. К перегрузке этого оператора прибегают в тех случаях, когда нужна операция, требующая многих параметров (например, индекс в многомерном массиве). Использовать в этом случае оператор индексирования нельзя, поскольку он допускает только один индекс.

Пусть Вас не вводит в заблуждение название этого оператора. Когда оператор вызова функции перегружен, он модифицирует интерпретацию применения круглых скобок к *объектам* того класса, в котором он объявлен, а не то, как функции вызываются.

Рассмотрим пример перегрузки оператора вызова функции:

```
#include <iostream.h>
class Coord
  int x, y;
public:
  Coord(int _x, int _y){x = _x; y = _y;}
  Coord(){x = 0; y = 0;}
  void GetCoord(int& _x, int& _y){_x = x; _y = y;}
  Coord& operator()(int dx, int dy){x +=dx; y +=dy; return *this;}
};
main()
{
 int x, y;
 Coord Pt1, Pt2;
 Pt2 = Pt1(3,2);
 Pt2.GetCoord(x,y);
 cout << "Pt2.x =" << x << " Pt2.y =" << y << endl;
 return 0;
}
```

При выполнении программа выводит на экран:

```
Pt2.x = 3 Pt2.y = 2
```

Не следует путать выражение

```
Coord Pt(3,2);
```

с выражением

```
Pt(3,2);
```

Первое из них вызовет обращение компилятора к конструктору с параметрами, второе – к перегруженному оператору вызова

функции. Этот пример показывает, что к объекту класса, в котором определен перегруженный оператор вызова функции, можно обращаться, используя синтаксис вызова функции.

Тема 15.9
Перегрузка операторов доступа к членам класса

Доступ к членам класса может управляться путем перегрузки оператора выбора члена (->). Этот оператор рассматривается как унарный. Соответствующая перегруженная операторная функция должна быть объявлена как нестатическая функция-член класса. Поэтому объявление такой функции имеет вид:

```
T* operator->()
```

где T – это имя класса, которому принадлежит данный оператор. Перегрузка этого оператора часто используется для реализации «умных» указателей, которые, например, проверяют указатели на правильность до их использования.

Замечание: оператор выбора члена, обозначаемый точкой, не может быть перегружен.

Приведем пример перегрузки оператора доступа к члену класса:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
 int GetX(){return x;}
 int GetY(){return y;}
 Coord* operator->();
};
Coord*
Coord::operator->()
{
 cout << "Доступ к члену: ";
 return this;
}
main()
{
 Coord Pt(1,5);
 cout << "Pt->x = " << Pt->GetX() << endl;
 cout << "Pt->y = " << Pt->GetY() << endl;
 return 0;
}
```

При выполнении программа выводит на экран:

```
Доступ к члену: Pt->x = 1
Доступ к члену: Pt->y = 5
```

Обратите внимание, что в случае перегрузки оператора доступа к членам класса доступ к этим членам осуществляется путем указания перед стрелкой самого объекта, а не указателя на него. Почему это так, вам станет ясно, если записать выражения

```
Pt->GetX()
Pt->GetY()
```

так, как их трактует компилятор:

```
(Pt.operator->())->GetX()
(Pt.operator->())->GetY()
```

Тема 15.10
Перегрузка операторов new и delete

C++ поддерживает динамическое распределение и очистку памяти с помощью операторов new и delete. Оператор new вызывает специальную функцию operator new() (или operator new[]() в варианте, применяемом для массивов), а оператор delete – функцию operator delete() (соответственно, operator delete[]() – для массивов). В дальнейшем мы не будем оговаривать особо, что изложенное относится и к вариантам рассматриваемых функций для массивов, кроме тех случаев, когда имеются существенные отличия в их использовании. Имеются два варианта этих функций с различными областями видимости:

::operator new() – глобальная (стандартная);
::operator delete() – глобальная (стандартная);
<имя_класса>::operator new() – функция класса;
<имя_класса>::operator delete() – функция класса.

Определенная в классе функция operator new() является статической функцией-членом, которая скрывает глобальную функцию ::operator new() для объектов типа, задаваемого этим классом.

По умолчанию, если нет перегруженной версии оператора new, запрос на динамическое распределение памяти приводит к вызову глобальной функции ::operator new(). Запрос на выделение памяти для массива приводит к вызову функции ::operator new[]().

Для объектов некоторого класса может быть определен специальный оператор <тип_класса>::operator new() или <тип_класса>::operator new[](). Когда оператор new применяется к объектам некоторого класса, он вызывает соответствующую операторную функцию <тип_класса>::operator new(), если она объявлена в классе; в противном случае он вызывает глобальную функцию ::operator new(). Глобальные функции ::operator new() и ::operator new[]() могут быть перегружены, причем каждый перегруженный экземпляр должен иметь уникальный прототип. Поэтому в одной программе могут сосуществовать несколько экземпляров глобального оператора new. Операторы распределения памяти для объектов некоторого класса также могут быть перегружены.

Определенный пользователем оператор new должен возвращать void* и принимать в качестве первого параметра size_t. Тип size_t определен в файле <stddef.h> как unsigned int. Чтобы перегрузить оператор new, используйте следующие прототипы:

```
//Для объектов
void* operator new(size_t size);
//Для массивов объктов
void* operator new[](size_t size);
```

Эти прототипы объявлены в заголовочном файле new.h. Поэтому, если Вы собираетесь использовать перегрузку операторов new и delete, не забудьте подключить этот файл к программе.

Величину size (длину объекта в байтах) для оператора new вычисляет и предоставляет операторной функции сам компилятор, так что Вам не требуется применять выражения вида sizeof(size) для вычисления длины объектов заданного типа (как при использовании функции malloc). Поводом для включения этого аргумента в объявление функции является то, что производные классы наследуют operator new() и operator new[](), но величина объектов производных классов может отличаться от величины объектов базового класса.

Можно также использовать *помещающую форму* оператора new, если перегруженная версия этого оператора содержит дополнительный параметр:

```
void*
operator new(size_t size, void* p);
//Для объектов
void* operator new[](size_t Type_size, void* p);
//Для массивов объектов
```

Вы можете применить помещающий синтаксис, когда хотите использовать память, которая уже была однажды распределена в начале программы. Обычно помещающая форма применяется для глобальных объектов, которые должны быть инициализированы после того, как будут выполнены определенные операции. Другими словами, речь идет о размещении объекта по абсолютному адресу (и необязательно в пуле свободной памяти), причем место для него было ранее зарезервировано. В этом случае за удаление распределенной памяти ответственность несет программист: уже нельзя полагаться на то, что глобальный оператор delete корректно выполнит очистку памяти. Чтобы очистить память, делается *явный* вызов деструктора. Чтобы явно вызвать деструктор для объекта, скажем, Ob типа T, используйте следующий синтаксис:

```
Ob.T::~T();
pOb->T::~T();
```

где pOb – указатель на объект этого типа. Явный вызов деструктора может быть использован безотносительно к тому, определен ли в соответствующем классе деструктор. Если деструктор в классе не определен, просто ничего не делается.

Этот метод очистки памяти следует использовать с особой осторожностью. Если вы сделаете вызов деструктора прежде, чем объект, созданный в стеке, выйдет из области видимости, деструктор будет вызван снова при очистке стека для блока.

Синтаксис вызова перегруженного оператора new такой:

```
<::> new <placement> type_name <(initializer)>
<::> new <placement> (type_name) <(initializer)>
```

Здесь угловые скобки обозначают необязательные элементы синтаксиса, placement – дополнительные параметры операторной функции operator new() (первый параметр типа size_t, содержащий информацию о длине типа, предоставляет сам компилятор), type_name – имя типа данных, для которого выделяется память, initializer – инициализирующие значения, которые передаются конструктору объекта с именем типа type_name. Не путайте эти инициализирующие значения конструктора класса с дополнительными параметрами операторной функции operator new(), которые на самом деле передаются этой функции элементом синтаксиса placement. Обязательными элементами синтаксиса являются ключевое слово new и имя типа данных type_name.

Таким образом, при вызове оператора new можно задать значения, которые будут использованы для инициализации объекта.

При использовании этого оператора для массивов задать инициализирующие значения нельзя (то есть элемент синтаксиса вызова initializer в этом случае недопустим). Именно по этой причине, как мы уже упоминали ранее, массив (любой, а не только объектов), память для которого выделяется динамически, не может быть инициализирован при создании. В отсутствие инициализирующих значений объект, созданный оператором new, содержит непредсказуемые данные. Напомним, что массивы объектов класса инициализируются с помощью конструктора по умолчанию. Вызов оператора new для распределения памяти одному объекту приводит к вызову функции operator new(). Любой запрос на выделение памяти массиву приводит к вызову функции operator new[]().

Соответствующая функция пытается создать объект типа T путем распределения (если это возможно) sizeof(T) байтов памяти из пула свободной памяти (который называется также кучей). В случае успешного выделения памяти для размещения объекта (объектов) возвращается указатель типа T*, который уже не нуждается в явном приведении типа (в отличие от функции malloc).

Рассмотрим пример переопределения и перегрузки глобальных операторов new и delete.

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

//Переопределение глобального оператора new
void* operator new(size_t size)
{
 printf("Запрос %u байтов памяти\n", size);
 return malloc(size);
}

void operator delete(void* p)
{
 printf("Освобождение памяти\n");
 free(p);
}

//Перегрузка глобального оператора new
void* operator new
(size_t size, char* fname, int line)
{
 printf("Запрос %u байтов памяти из %s в
↳строке %d\n", size, fname, line);
 return malloc(size);
}

int main()
```

```
{
 //Вызов глобального оператора new
 long* pInt = new long;
 //Вызов глобального оператора delete
 delete pInt;
 //Вызов перегруженного оператора new
 pInt = new(__FILE__, __LINE__) long;
 //Вызов глобального оператора delete
 delete pInt;
 return 0;
}
```

Здесь переопределенные операторы new и delete вызывают функции malloc и free соответственно. А перегруженная версия оператора new дополнительно выводит имя файла и строку в нем, из которой он был вызван.

При выполнении программа выводит на экран:

```
Запрос 4 байтов памяти
Освобождение памяти
Запрос 4 байтов пямяти из ex15.cpp в строке 158
Освобождение памяти
```

Рассмотрим пример перегрузки глобального оператора new с целью использования помещающего синтаксиса:

```
#include <iostream.h>
//Перегрузка глобального оператора new
//с целью использования помещающего синтаксиса
inline void* operator new
(size_t size, void* ptr)
{
 cout <<"Помещение объекта по "<< "заданному адресу\n";
 return ptr;
}
void CheckSystemConf()
{
 cout <<"Проверка конфигурации системы\n";
}
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y)
  {
   x = _x; y = _y;
   cout <<"Инициализируем объект "<<"заданными величинами\n"
<< "x = " << x << " y = " << y << "\n";
  }
```

```
  Coord()
  {
   x = 0; y = 0;
   cout << "Инициализируем объект по " << "умолчанию\n"
<< "x = " << x << " y = " << y << "\n";
  }
  ~Coord()
     {cout << "Coord::~Coord()\n";}
};
//Глобальный объект
Coord  gOb;
main()
{
//Выполнение какой-то обработки
CheckSystemConf();
 //Теперь можно инициализировать объект
 Coord* ptr = new(&gOb) Coord(10,20);
 //Явный вызов деструктора
 ptr->Coord::~Coord();
 return 0;
}
```

При выполнении программа выводит на экран:

```
Инициализируем объект по умолчанию
x = 0 y = 0
Проверка конфигурации системы
Помещение объекта по заданному адресу
Инициализация объекта заданными величинами
x = 10 y = 20
Coord::~Coord
Coord::~Coord
```

Рассмотрим теперь пример перегрузки оператора new в классе с целью использования помещающего синтаксиса:

```
#include <iostream.h>
class String
{
  union
  {
    char  ch;
    char  buf[81];
  };
public:
  String(char c = '\0'): ch(c)
  {
   cout << "Символьный конструктор "<< "класса String" << endl;
```

```
    }
    String(char *s)
    {
     cout << "Строковый конструктор "<< "класса String" << endl;
     strcpy(buf,s);
    }
    ~String()
       {cout << "String::~String()" << endl;}
    //Перегрузка  оператора new класса
    //с целью использования помещающего синтаксиса
    void* operator new(size_t, void* buffer)
                {return buffer;}
};
char str[sizeof(String)];
void main()
{
 //Помещаем строку в буфер
 String* ptr = new(str) String("c++");
 cout << "str = " << str << endl;
 //Явный вызов деструктора
 ptr->String::~String();
 //Помещаем 'c' в начало строки str
 ptr = new(str) String ('C');
 cout << "str[0] = " << str[0] << endl;
 cout << "Новое содержимое буфера:\n";
 cout << "str = " << str << endl;
 //Явный вызов деструктора
 ptr->String::~String();
}
```

В рассматриваемом примере программа вначале создает глобальный объект строку str, затем в эту строку помещает объект класса String, который инициализирует ее значением "c++". После этого программа осуществляет явный вызов деструктора, который в данном случае просто выводит на экран строку "String::~String()". Затем в строке str меняется первый символ "c" на "C" и снова явно вызывается деструктор. При выполнении программа выводит на экран:

```
Строковый конструктор класса String
str = c++
String::~String()
Символьный конструктор класса String()
str[0] = C
Новое содержимое буфера:
str = C++
String::~String()
```

Динамически распределенная оператором new память может быть освобождена оператором delete. Оператор delete вызывает функцию operator delete(), которая освобождает память, возвращая ее в пул свободной памяти (кучу). Использование оператора delete также приводит к вызову деструктора класса (если он имеется).

Глобальные операторы ::operator delete() и ::operator delete[]() не могут быть перегружены, однако Вы можете *переопределить* их. В программе может существовать только один экземпляр каждой глобальной функции delete. Определенный пользователем оператор delete должен возвращать тип void и принимать void* в качестве своего первого параметра. Второй аргумент типа size_t является необязательным. В классе T может быть определена только одна версия каждой операторной функции T::operator delete() и T::operator delete[](). Если она определена, то скрывает глобальную функцию ::operator delete(). Чтобы перегрузить операторы delete в классе, используйте следующие прототипы:

```
void operator delete(void *ptr, [size_t size]);  //Для объектов
void operator delete[](void *ptr, [size_t size]);  //Для массивов
 //объектов
```

В классе может присутствовать только один из этих двух вариантов. Первая форма оператора delete работает так же, как описанная выше глобальная форма. Вторая форма принимает два аргумента, первый из которых – это указатель на блок памяти, который подлежит возврату в пул свободной памяти, второй – число возвращаемых байтов. Вторая форма оператора особенно полезна, когда наследуемый оператор delete используется для удаления объекта производного класса. Рассмотрим пример перегрузки оператора delete класса:

```
#include <stdlib.h>
class T {
  ...
public:
  void* operator new(size_t size)
        { return malloc(size);}
  void operator delete(void* p)
    { free(p); }
  T() { /* какая-то инициализация */ }
  T(char ch)
{ /* какая-то инициализация */ }
  ~T() { /* очистка памяти */ }
  ...
};
```

Аргумент size определяет величину создаваемого объекта.

Стандартные (глобальные) операторы new и delete все еще могут использоваться в пределах области видимости класса с помощью явного вызова ::operator new(), ::operator new[](), ::operator delete() и ::operator delete[]() или неявно при создании и удалении объектов других классов, не являющихся производными класса, в котором эти операторы перегружены. Например, Вы можете использовать стандартные операторы new и delete для определения своих перегруженных версий:

```
void* T::operator new(size_t size)
{
//Вызов стандартного оператора new
void* ptr = new char[size];
   ...
  return ptr;
}
void T::operator delete(void* ptr)
{
   ...
 //Вызов стандартного оператора delete
 delete (void*) ptr;
}
```

Заметим, что, определив функцию по имени operator new в некотором классе, Вы скрываете стандартную форму оператора new. Часто это является нежелательным. Избежать неприятностей можно, дополнительно определив в классе операторную функцию, поддерживающую стандартный синтаксис

```
void* operator new (size_t size)
  {return ::operator new(size);}
```

или, если для каждого дополнительного параметра оператора new задать значение по умолчанию:

```
void* operator new (size_t size, void* = 0);
```

Напомним, что оператор new наследуется производными классами. Поэтому, если Вы строите иерархию классов и в базовом классе перегружаете оператор new, в нем следует предусмотреть проверку длины объекта или его типа; в зависимости от полученных результатов должна выполняться различная обработка. Например, можно поручить выполнять распределение памяти для объектов производных классов стандартному оператору new. В частности, проверка длины объекта позволяет переадресовать

запросы на выделение нулевого количества байтов стандартному оператору new. То же самое относится и к оператору delete. Кроме того, следует предусмотреть возможность безопасного удаления нулевого объекта. Следующий пример демонстрирует перечисленные правила:

```
class Base
{
 public:
   void* operator new(size_t size);
   void  operator delete
       (void* pMem, size_t size);
};
void*
Base::operator new(size_t size)
{
 if (size != sizeof(Base)
 return ::operator new(size);
 //А здесь выполняем распределение памяти
 //объекту базового класса
}
void
Base::operator delete
       (void* pMem, size_t size)
{
 if (pMem == 0) return;
 if (size != sizeof(Base))
   {
    ::operator delete(pMem);
    return;
   }
 //Удаляем объект базового класса
}
class Derived: public Base
{
 //Класс Derived не объявляет operator new и operator delete
};
main()
{
 //Вызов Base::operator new()
 Derived* ptr = new Derived;
 //... Выполнение какой-то обработки
 //Вызов Base::operator delete()
 delete ptr;
}
```

Тема 15.11
Функции преобразования типа

Часто встречается задача преобразования объекта одного типа в объект другого типа. Мы уже рассмотрели ранее возможность такого преобразования с помощью конструкторов. Однако конструкторы преобразования имеют одно существенное ограничение: они позволяют преобразовать встроенный тип данных в объект, но не наоборот. *Функции (или операторы) преобразования типа* предоставляют возможность обратного преобразования объекта некоторого класса к любому из встроенных типов. Они имеют следующий вид:

```
operator <имя_нового_типа>();
```

Функции преобразования подчиняются следующим правилам:

- функция преобразования не имеет параметров;
- в функциях преобразования типа не задается тип возвращаемого значения (подразумевается, что она возвращает тип, указанный после ключевого слова operator);
- функция преобразования типа наследуется.

Следующий пример показывает, как используются функции преобразования типа:

```
#include <iostream.h>
class Coord
{
  int x, y;
 public:
  Coord(int _x, int _y){x = _x; y = _y;}
  //Функция преобразования типа
  operator int() {return x*y;}
};
int main()
{
 Coord Pt(10,20);
 int n;
 //Неявный вызов функции приведения типа
 n = Pt + 30;
 //Явный вызов функции приведения типа
 cout << "int(Pt) = " <<  int(Pt) <<endl;
 cout << "n = Pt + 30 = " << n;
}
```

При выполнении программа выводит на экран:

```
int(Pt) = 200
```

```
n = Pt + 30 = 230
```

Таким образом, использование функции преобразования типа в данном случае обеспечивает возможность участия объекта Pt типа Coord в неявных преобразованиях типа в выражении

```
n = Pt + 30;
```

и, с другой стороны, позволяет осуществлять явное преобразование типа, как это сделано в выражении int(Pt).

Практикум «Перегрузка операторов»

Упражнение 15.1
Перегрузка оператора умножения

Модифицируйте приведенный ниже пример таким образом, чтобы перегрузка оператора умножения действовала как для правого, так и для левого операнда.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
   int x;
public:
   MyClass(){x = 0;}
   MyClass(int _x){x = _x;}
   MyClass operator*(MyClass& ob);
   MyClass operator*(int _mult);
   int GetVal(){return x;}
   void SetVal(int _x){x = _x;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include <iostream.h>
#include "header.h"
MyClass
MyClass::operator*(MyClass& ob)
{
   MyClass tmpOb;
```

```
    tmpOb.x = x * ob.x;
    return tmpOb;
}
MyClass
MyClass::operator*(int _mult)
{
    MyClass tmpOb;
    tmpOb.x = x * _mult;
    return tmpOb;
}
int main()
{
    MyClass obj;
    obj.SetVal(10);
    obj = obj * 3;
    cout << obj.GetVal() << endl;
    return 0;
}
```

Упражнение 15.2

Перегрузка операторов < и >

Дополните приведенный ниже класс MyClass перегруженными операторами < и > для аргументов типа MyClass.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
    int x;
public:
    MyClass(){x = 0;}
    int GetX(){return x;}
    void SetX(int _x){x = _x;}
    bool operator<(int _x){return x < _x;}
    bool operator>(int _x){return x > _x;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include <iostream.h>
#include "header.h"
int main()
{
    MyClass big, small;
```

```
    big.SetX(10);
    small.SetX(3);
    if(big > small.GetX())
       cout << big.GetX() << endl;
    return 0;
}
```

Упражнение 15.3

Использование перегрузки оператора присваивания

Проанализируйте предлагаемый ниже пример перегрузки оператора присвоения.

Имеется некоторый класс MyClass, содержащий закрытый целочисленный член x и интерфейсные функции для работы с ним. В описание класса также включены описания и реализация операторов присвоения для случая, когда правый операнд является экземпляром этого же класса (MyClass& operator=(MyClass& rhs)), а также для целочисленного аргумента (MyClass& operator=(int _x)).

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
   int x;
public:
   MyClass(){x = 0;}
   int GetX(){return x;}
   void SetX(int _x){x = _x;}
   MyClass& operator=(MyClass& rhs)
   {
      if(this == &rhs)
         return *this;
      x = rhs.x;
      return *this;
   }
   MyClass& operator=(int _x)
   {
      x = _x;
      return *this;
   }
};
#endif//_HEADER_H_
// Основной модуль программы
```

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

int main()
{
   MyClass First, Second;
   First.SetX(15);
   Second = First;
   cout << Second.GetX() << endl;
   Second = 17;
   cout << Second.GetX() << endl;
   return 0;
}
```

Упражнение 15.4
Перегрузка префикса и постфикса

Дополните приведенный ниже класс MyClass префиксным и постфиксным операторами инкремента и декремента.

```
// Заголовочный файл

#ifndef _HEADER_H_
#define _HEADER_H_

class MyClass
{
   int x;
public:
   MyClass(){x = 0;}
   int GetX(){return x;}
   void SetX(int _x){x = _x;}
};

#endif//_HEADER_H_
```

Раздел 16
Полиморфизм и виртуальные функции

Тема 16.1
Раннее и позднее связывание. Динамический полиморфизм

В C++ полиморфизм поддерживается двумя способами. Во-первых, при компиляции он поддерживается посредством перегрузки функций и операторов. Такой вид полиморфизма называется *статическим полиморфизмом*, поскольку он реализуется еще до выполнения программы, путем *раннего связывания* идентификаторов функций с физическими адресами на стадии компиляции и компоновки. Во-вторых, во время выполнения программы он поддерживается посредством виртуальных функций. Встретив в коде программы вызов виртуальной функции, компилятор (а точнее, компоновщик) только обозначает этот вызов, оставляя связывание идентификатора функции с ее адресом до стадии выполнения. Такой процесс называется *поздним связыванием*. Виртуальная функция – это функция, вызов которой (и выполняемые при этом действия) зависит от типа объекта, для которого она вызвана. Объект определяет, какую функцию нужно вызвать уже во время выполнения программы. Этот вид полиморфизма называется *динамическим полиморфизмом*. Основой динамического полиморфизма является предоставляемая C++ возможность определить указатель на базовый класс, который реально будет указывать не только на объект этого класса, но и на любой объект производного класса. Эта возможность возникает благодаря наследованию, поскольку объект производного класса всегда является объектом базового класса. Во время компиляции еще не известно, объект какого класса захочет создать пользователь, располагая указателем на объект базового класса. Такой указатель связывается со своим объектом только во время

выполнения программы, то есть динамически. Класс, содержащий хоть одну виртуальную функцию, называется полиморфным.

Для каждого полиморфного типа данных компилятор создает таблицу виртуальных функций и встраивает в каждый объект такого класса скрытый указатель на эту таблицу. Она содержит адреса виртуальных функций соответствующего объекта. Имя указателя на таблицу виртуальных функций и название таблицы зависят от реализации в конкретном компиляторе. Например, в Visual C++ 6.0 этот указатель имеет имя vfptr, а таблица называется vftable (от англ. Virtual Function Table). Компилятор автоматически встраивает в начало конструктора полиморфного класса фрагмент кода, который инициализирует указатель на таблицу виртуальных функций. Если вызывается виртуальная функция, код, сгенерированный компилятором, находит указатель на таблицу виртуальных функций, затем просматривает эту таблицу и извлекает из нее адрес соответствующей функции. После этого производится переход на указанный адрес и вызов функции.

Напомним, что при создании объекта производного класса вначале вызывается конструктор его базового класса. На этой стадии создается и таблица виртуальных функций, а также указатель на нее. После вызова конструктора производного класса указатель таблицу виртуальных функций настраивается таким образом, чтобы он ссылался на переопределенный вариант виртуальной функции (если такой есть), существующий для объекта этого класса.

В связи с этим нужно отдавать себе отчет о той цене, которую приходится платить за возможность использования позднего связывания.

Поскольку объекты с виртуальными функциями должны поддерживать и таблицу виртуальных функций, то их использование всегда ведет к некоторому повышению затрат памяти и снижению быстродействия программы. Если Вы работаете с небольшим классом, который не собираетесь делать базовым для других классов, то в этом случае нет никакого смысла в использовании виртуальных функций.

Тема 16.2
Виртуальные функции

Функции, у которых известен интерфейс вызова (то есть прототип), но реализация не может быть задана в общем случае, а может быть определена только для конкретных случаев, называ-

ются *виртуальными* (термин, означающий, что функция может быть переопределена в производном классе).

Виртуальные функции – это функции, которые гарантируют, что будет вызвана правильная функция для объекта безотносительно к тому, какое выражение используется для осуществления вызова.

Предположим, что базовый класс содержит функцию, объявленную виртуальной, и производный класс определяет ту же самую функцию. В этом случае функция из производного класса вызывается для объектов производного класса, даже если она вызывается с использованием указателя или ссылки на базовый класс. Пример:

```
#include <iostream.h>
class Base
{
 public:
   Base(int _x){x = _x;}
   virtual int GetX(){return x;}
   virtual void PrintX();
 private:
   int x;
};
void Base:: PrintX()
{
 cout << "Ошибка. Эта функция "< "недоступна для типа Base.\n";
}
class Derived1: public Base
{
 public:
   Derived1(int _x): Base(_x){}
   void PrintX();
};

void
Derived1::PrintX()
{
 cout <<"Derived1::x = " << GetX() << "\n";
}
class Derived2: public Base
{
 public:
   Derived2(int _x):Base(_x){}
   void PrintX();
};
```

```
void
Derived2:: PrintX()
{
 cout << "Derived2::x = " << GetX() << "\n";
}
void main()
{
 Derived1 *pDer1 = new Derived1(10);
 Derived2 *pDer2 = new Derived2(20);
 Base* pBase = pDer1;
 pBase->PrintX();
 pBase = pDer2;
 pBase-> PrintX();
}
```

Функция PrintX() в производных классах является виртуальной, так как она объявлена виртуальной в базовом классе Base. Для вызова виртуальных функций, таких как PrintX(), может быть использован следующий код:

```
Base* pBase = pDer1;
pBase->PrintX();
pBase = pDer2;
pBase->PrintX();
```

Так как PrintX() является виртуальной, в предыдущем коде для каждого объекта вызывается своя версия функции. Функция PrintX() в производных классах Derived1 и Derived2 переопределяет функцию базового класса. Если объявлен класс, который не предоставляет переопределенной реализации функции PrintX(), используется реализация по умолчанию из базового класса. В результате предыдущая программа при выполнении выводит на экран:

```
Derived1::x = 10
Derived2::x = 20
```

При вызове функции с помощью указателей и ссылок применяются следующее правила:

- вызов виртуальной функции разрешается в соответствии с тем типом объекта, для которого она вызывается;
- вызов невиртуальной функции разрешается в соответствии с типом указателя или ссылки.

Так как виртуальные функции вызываются только для объектов, принадлежащих некоторому классу, нельзя объявить глобальную или статическую функцию виртуальной. Ключевое слово vir-

tual может быть использовано при объявлении переопределяемой функции в производном классе, однако оно не является обязательным: *переопределения виртуальных функций сами являются виртуальными функциями.*

Виртуальные функции, объявленные в базовом классе, должны быть в нем определены, если они не объявлены как чисто виртуальные. Функция, объявленная в производном классе, переопределяет виртуальную функцию в базовом классе только тогда, когда имеет то же имя и работает с тем же количеством и типами параметров, что и виртуальная функция базового класса. Если они отличаются типом хоть одного параметра, то функция в производном классе считается совершенно новой и переопределения не происходит. Функция в производном классе не может отличаться от виртуальной функции в базовом классе только своим возвращаемым значением, должен также отличаться список аргументов. Следующий пример показывает, что виртуальная функция переопределяет только виртуальную функцию базового класса с тем же прототипом.

```
#include <iostream.h>
class Base
{
  int x;
 public:
   virtual void SetValue(int _x)
   {
    x = _x;
    cout << "Base::x = " << x << "\n";
   }
   virtual void Show(Base* pOb)
     {SetValue(10);}
};
class Derived: public Base
{
  int x, y;
 public:
  virtual void SetValue(int _x, int _y)
  {
   x = _x; y = _y;
   cout << "Derived::x= " << x << " Derived::y= " << y << "\n";
  }
  void Show(Base* pOb) {SetValue(15, 20);}
};
int main()
{
```

```
Base* pOb1 = new Base;
Base* pOb2 = new Derived;
//Вызов виртуальной функции Show() из Base
pOb1->Show(pOb1);
//Вызов виртуальной функции Show() из Derived
pOb2->Show(pOb1);
//Вызов виртуальной функции Show() из Derived
pOb2->Show(pOb2);
return 0;
}
```

При выполнении программа выводит на экран:

```
Base::x = 10
Derived::x= 15 Derived::y = 20
Derived::x= 15 Derived::y = 20
```

В рассматриваемом примере базовый и производный классы имеют по две виртуальные функции с одинаковыми именами. Однако компилятор трактует их различным образом. Поскольку прототип функции SetValue в производном классе изменен, она рассматривается как совершенно новая виртуальная функция. С другой стороны, виртуальная функция Show() производного класса рассматривается как переопределение соответствующей виртуальной функции базового класса.

Тема 16.3
Виртуальные и невиртуальные функции

Следующий пример показывает, как ведут себя виртуальные и невиртуальные функции, когда они вызываются через указатели:

```
#include <iostream.h>
class Base
{
 public:
   virtual void VirtFunc();
   void InvokingClass();
};

void
Base:: VirtFunc()
{
 cout << "Base:: VirtFunc()\n";
}

void
Base::InvokingClass()
```

```
{
 cout << "Эта функция вызвана " << "классом Base\n";
}
class Derived: public Base
{
 public:
   //Виртуальная функция
   void VirtFunc();
   //Невиртуальная функция
   void InvokingClass();
};
void
Derived:: VirtFunc()
{
 cout << "Derived:: VirtFunc()\n";
}
void
Derived::InvokingClass()
{
 cout << "Эта функция вызвана " << "классом Derived\n";
}
void main()
{
 Derived aDerived;
 Derived *pDerived = &aDerived;
 Base    *pBase = &aDerived;
 //Вызов виртуальной функции
 pBase-> VirtFunc();
 //Вызов невиртуальной функции
 pBase->InvokingClass();
 //Вызов виртуальной функции
 pDerived-> VirtFunc();
 //Вызов невиртуальной функции
 pDerived->InvokingClass();
}
```

При выполнении программа выводит на экран:

```
Derived:: VirtFunc()
Эта функция вызвана классом Base
Derived:: VirtFunc()
Эта функция вызвана классом Derived
```

Заметим, что безотносительно к тому, вызывается ли функция VirtFunc() через указатель на класс Base или на класс Derived, она вызывает функцию из класса Derived. Так происходит потому, что VirtFunc() является виртуальной функцией и оба указателя pBase и pDerived указывают на объект типа Derived.

Следующий пример демонстрирует разницу между виртуальными и невиртуальными функциями.

```
#include <iostream.h>
class Base
{
 public:
   virtual void VirtFunc()
   {cout << "Мы в Base::VirtFunc\n";}
   void NonVirtFunc()
   {cout << "Мы в Base::NonVirtFunc\n";}
};
//Производный класс переопределяет обе
//функции класса Base
class Derived: public Base
{
 public:
   virtual void VirtFunc()
       {cout << "Мы в Derived::VirtFunc\n";}
   void NonVirtFunc()
    {cout << "Мы в Derived::NonVirtFunc\n";}
};
int main()
{
 Base* pBase = new Derived;
 //Вызов виртуальной функции из Derived
 pBase->VirtFunc();
 //Вызов невиртуальной функции из Base
 pBase->NonVirtFunc();
 return 0;
}
```

При выполнении программа выводит на экран:

```
Мы в Derived::VirtFunc
Мы в Base::NonVirtFunc
```

Таким образом, использование указателя pBase, который реально указывает на объект производного класса с невиртуальными функциями, приводит к вызову соответствующей функции из *базового* класса.

Вы можете выключить механизм позднего связывания с помощью явной квалификации имени вызываемой функции путем использования оператора расширения области видимости (::). Например:

```
#include <iostream.h>
class Base
{
```

```
 public:
   virtual void VirtFunc()
   {cout << "Мы в Base::VirtFunc\n";}
};
class Derived: public Base
{
 public:
   virtual void VirtFunc()
   {cout << "Мы в Derived::VirtFunc\n";}
};
int main()
{
 Base* pBase = new Derived;
 //Вызов виртуальной функции из Derived
 pBase->VirtFunc();
 //Вызов виртуальной функции из Base.
 //Механизм позднего связывания выключен
 pBase->Base::VirtFunc();
 return 0;
}
```

В процессе выполнения программа выводит на экран:

```
Мы в Derived::VirtFunc
Мы в Base::VirtFunc
```

Таким образом, инструкция

```
pBase->Base::VirtFunc();
```

приводит к выключению механизма позднего связывания.

Отметим также, что в C++ можно объявлять невиртуальные перегруженные функции, совпадающие по имени с виртуальными функциями. Однако невиртуальные функции, имена которых совпадают с виртуальными функциями, оказываются скрытыми. Рассмотрим пример, иллюстрирующий сказанное:

```
#include <iostream.h>
#include <string.h>
class Base
{
 public:
   Base(char* name) {strcpy(Name,name);}
   virtual void Func(char c)
 {
 cout << "virtual " << Name << "::Func "
 << " принимает параметр " << c << endl;
 }
 protected:
```

```
  char Name[20];
};
class Derived1: public Base
{
 public:
  Derived1(char* name):Base(name){}
  void Func(const char* s)
{
 cout << Name << "::Func "  << " принимает параметр "
      << s << endl;
}
        void Func(int n)
{
 cout << Name << "::Func " << " принимает параметр "
      << n << endl;
}
        virtual void Func(char c)
{
 cout << "virtual "< Name << "::Func "<<" принимает параметр "
      << c << endl;
}
};
class Derived2: public Derived1
{
 public:
  Derived2(char* name):Derived1(name){}
  void Func(const char* s)
     {
        cout << Name << "::Func "<< " принимает параметр "
             << s << endl;
}
  void Func(double d)
     {
      cout     << Name << "::Func "<<" принимает параметр "
               << d << endl;
     }
  virtual void Func(char c)
     {
          cout << "virtual "
               << Name << "::Func "<< " принимает параметр "
               << c << endl;
     }
};
int main()
{
 Base base("Base");
 Derived1 der1("Derived1");
```

```
Derived2 der2("Derived2");
base.Func('X');
der1.Func('Y');
der1.Func(10);
der1.Func("Der1");
der2.Func('Z');
der2.Func(10.1234);
der2.Func("Der2");
return 0;
}
```

Этот пример содержит три класса, которые образуют линейную иерархию наследования. В классе Base объявлена виртуальная функция Func(char). Класс Derived1 объявляет свою версию виртуальной функции Func(char) и две невиртуальные перегруженные функции Func(const char*) и Func(int). Класс Derived2 объявляет свою версию виртуальной функции Func(char) и две невиртуальные перегруженные функции Func(const char*) и Func(double). Заметим, что в классе Derived2 приходится вновь объявлять функцию Func(const char*), хотя она имеет тело, полностью совпадающее со своим аналогом из класса Derived1, поскольку она оказывается скрытой из-за наличия в классе Derived1 виртуальной и невиртуальной функций с тем же именем. Кроме того, в классах Derived1 и Derived2 приходится заново объявлять и виртуальную функцию Func(char), так как она также оказывается скрытой из-за наличия в этих классах перегруженных функций с теми же именами. При выполнении программа выводит на экран:

```
virtual Base::Func принимает параметр X
virtual Derived1::Func принимает параметр Y
Derived1::Func принимает параметр 10
Derived1::Func принимает параметр Der1
virtual Derived2::Func принимает параметр Z
Derived2::Func принимает параметр 10.1234
Derived2::Func принимает параметр Der2
```

Если Вы закомментируете объявления виртуальной функции в производных классах, вызовы этой функции для соответствующих объектов будут трактоваться как вызовы функции Func(int), объявленной в классе Derived1. Как объяснялось раньше, в этом случае можно получить доступ к функции из класса Base явной квалификацией имени:

```
der1.Base::Func('Y');
```

Тема 16.4
Применение динамического полиморфизма

Теперь, когда Вы изучили, как используются виртуальные функции, рассмотрим менее тривиальный пример. Динамический полиморфизм предоставляет исключительные возможности для гибкого управления процессом выполнения программы во время ее работы. Представленная ниже программа реализует два вида связанных списков для хранения целых значений: очередь и стек. Напомним, что очередь реализует принцип: первым пришел – первым ушел, а стек: первым пришел – последним ушел. Очередь и стек могут быть реализованы различными способами: с использованием массивов, связного списка и другими. Здесь мы рассмотрим реализацию этих структур данных с использованием связного списка. В связи с этим мы будем рассматривать иерархию классов, в основе которой лежит класс, реализующий связный список, а очередь и стек будем рассматривать как виды связного списка. Хотя реализация этих двух видов списков совершенно различна, для доступа к ним будет использоваться один и тот же интерфейс.

```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
//Абстрактный класс для списков
class List
{
 public:
  List(){Head = Tail = Next = 0;}
  virtual void Put(int n) = 0;
  virtual bool Get(int& n) = 0;
  void SetValue(int n){Num = n;}
  int GetValue(){return Num;}
  List* Head;     //Указатель на начало списка
  List* Tail;     //Указатель на конец списка
  List* Next;     //Указатель на следующий элемент списка
 private:
  int Num;
};
//Список типа "очередь"
class Queue: public List
{
 public:
  void Put(int n);
  bool Get(int& n);
};
```

```
void
Queue::Put(int n)
{
 List* Item;
 //Создаем новый элемент очереди
 Item = new Queue;
 //Устанавливаем хранимое значение
 Item->SetValue(n);
 //Добавляем в конец очереди
 if (Tail) Tail->Next = Item;
 Tail = Item;
 Item->Next = NULL;
 if (!Head) Head = Tail;
}

bool
Queue::Get(int& n)
{
 List* Item;
 if (!Head)
    {
     n = 0;
     return false;
    }
 //Восстанавливаем значение первого элемента очереди
 n = Head->GetValue();
 //Устанавливаем начало очереди на следующий элемент
 Item = Head;
 Head = Head->Next;
 //Удаляем первый элемент очереди
 delete Item;
 return true;
}
//Список типа "стек"
class Stack: public List
{
 public:
   void Put(int n);
   bool Get(int& n);
};
void
Stack::Put(int n)
{
 List* Item;
 //Создаем новый элемент стека
 Item = new Stack;
 //Устанавливаем хранимое значение
 Item->SetValue(n);
```

```
//Добавляем новый элемент в начало стека
if (Head) Item->Next = Head;
Head = Item;
if (!Tail) Tail = Head;
}
bool
Stack::Get(int& n)
{
List* Item;
if (!Head)
  {
   n = 0;
   return false;
  }
//Восстанавливаем значение первого элемента стека
n = Head->GetValue();
//Устанавливаем начало стека на следующий элемент
Item = Head;
Head = Head->Next;
//Удаляем первый элемент стека
delete Item;
return true;
}
int main()
{
List* Item;
int saved;
Queue QueOb;
Stack StOb;
char symbol;
for (int i=0; i< 10; i++)
{
 cout <<"Куда поместить значение?"
     <<" В очередь или стек?(o/c): ";
cin >> symbol;
 symbol = tolower(symbol);
 if (symbol == 'o') Item = &QueOb;
 else if (symbol == 'c')Item = &StOb;
 else break;
 Item->Put(i);
}
cout << "Откуда извлечь значение?(o/c)\n"
<< "Нажмите <к>, чтобы завершить работу\n";
for(;;)
  {
   cin >> symbol;
   symbol = tolower(symbol);
```

```
    if (symbol == 'к')
        break;
    if (symbol == 'о') Item = &QueOb;
    else if (symbol == 'с') Item = &StOb;
    else break;
    if (Item->Get(saved))
        cout << saved << "\n";
    else
        cout << "Список пуст\n";
  }
 return 0;
}
```

Программа считывает вводимые с экрана символы и помещает в очередь или стек значение параметра цикла i. Затем она задает вопрос, откуда следует извлечь значение: из очереди или из стека, и в зависимости от ответа выводит на экран соответствующее значение.

Тема 16.5
Виртуальные деструкторы

Конструкторы не могут быть виртуальными, однако деструкторы могут быть виртуальными, и часто ими являются. В том случае, когда ожидается указатель на объект базового класса, вполне допустима и часто используется передача указателя на объект производного класса. При удалении объекта производного класса, на который ссылается указатель базового класса, если деструктор объявлен как виртуальный, то будет вызван деструктор соответствующего производного класса. Затем деструктор производного класса вызовет деструктор базового класса и объект будет правильно удален (удален целиком). В противном случае будет вызван только деструктор базового класса и произойдет утечка памяти. Рассмотрим пример:

```
class Base
{
  int x;
 public:
  Base(_x){x = _x;};
  ~Base()
  {
//Освобождение памяти
  }
};
```

```
class Derived: public Base
{
public:
  ~Derived()
  {
     //Освобождение памяти
  }
};
int main()
{
 Base* pBase = new Derived;
 //... Выполнение какой-то обработки
 delete pBase;
 return 0;
}
```

Так как объявленный в классе Derived деструктор не является виртуальным, инструкция

```
delete pBase;
```

вызовет удаление только памяти, принадлежащей классу Base, поскольку она будет квалифицирована как вызов Base::~Base(). Это может привести к утечке памяти. Если же сделать деструктор базового класса виртуальным, то и деструктор производного класса будет виртуальным. Значит, использование предыдущей инструкции будет квалифицировано как вызов

```
Derived::~Derived(),
```

и удаление объекта произойдет правильно.

В связи с этим можно сформулировать следующие правила:

- используйте виртуальный деструктор, если в базовом классе имеются виртуальные функции;
- используйте виртуальные функции только в том случае, если программа содержит и базовый, и производный классы;
- не пытайтесь создать виртуальный конструктор.

Тема 16.6
Абстрактные классы и чисто виртуальные функции

Классы могут быть созданы для того, чтобы предписать протокол взаимодействия с объектами, принадлежащими соответствующему типу. Эти классы называются «абстрактными», потому что никакой объект этого класса не может быть создан. Они существуют исключительно для создания производных классов.

Абстрактный класс – это класс, содержащий по меньшей мере одну виртуальную функцию. Производные от него классы должны реализовывать эти чисто виртуальные функции, иначе, если останется хоть одна нереализованная чисто виртуальная функция, такой класс также будет абстрактным. *Чисто виртуальная функция* объявляется с помощью следующего синтаксиса:

```
virtual <имя_функции>
(<список_параметров>) = 0;
```

Рассмотрим пример. Предположим, проектируется иерархия классов, причем базовый класс этой иерархии с именем Base должен обеспечить общие функциональные возможности, но объекты типа Base являются слишком общими, чтобы быть полезными. В этом случае класс Base является хорошим кандидатом в абстрактные классы:

```
class Base
{
 public:
 Base (int _x){x = _x;}
   virtual int GetX()
{ return  x;}
   virtual void PrintX() = 0;   //Чисто виртуальная функция
 private:
   int x;
};
```

Путем задания чисто виртуальной функции PrintX() при проектировании класса Base разработчик класса добивается того, что любой неабстрактный класс, являющийся производным класса Base, не может быть реализован без функции PrintX().

Существуют определенные ограничения на использование абстрактных классов. Они не могут использоваться в качестве:

- переменных, являющихся членами некоторых других классов;
- типов передаваемых в функцию аргументов;
- типов возвращаемых функцией значений;
- типов явных преобразований.

Другое ограничение состоит в том, что если конструктор абстрактного класса вызывает чисто виртуальную функцию, прямо или косвенно, результат непредсказуем.

Чисто виртуальные функции могут быть не только объявлены, но и определены в абстрактном классе. Они могут быть непосредственно вызваны только с использованием следующего синтаксиса:

```
<имя_абстр_класса>::<имя_вирт_функции>
(<список_параметров>)
```

Этот синтаксис используется, например, при разработке иерархии классов, базовые классы которой содержат *чисто виртуальные деструкторы*. Рассмотрим следующий пример:

```
#include <iostream.h>
class Base
{
 public:
   Base(){}
   //Чисто виртуальный деструктор
   virtual ~Base() = 0;
};
//Определение деструктора
Base::~Base()
{
}
class Derived: public Base
{
 public:
   Derived() {}
   ~Derived(){}
};
void main()
{
 Derived* pDerived = new Derived;
 delete  pDerived;
}
```

Напомним, что деструктор базового класса всегда вызывается в процессе разрушения объекта. Когда объект, на который указывает pDerived, удаляется, – вызывается деструктор класса Derived, а затем деструктор базового класса. Пустая реализация для чисто виртуальной функции (в данном случае – деструктора) гарантирует, что для этой функции существует хоть какая-то реализация.

К абстрактным классам применимы следующие правила:

- абстрактный класс не может использоваться в качестве типа аргумента, передаваемого функции;
- абстрактный класс не может использоваться в качестве типа возвращаемого значения функции;
- нельзя осуществлять явное преобразование типа объекта к типу абстрактного класса;
- нельзя объявить представитель абстрактного класса;
- можно объявить указатель или ссылку на абстрактный класс.

Практикум «Полиморфизм и виртуальные функции»

Упражнение 16.1
Создание виртуальных функций

Проанализируйте пример создания виртуальных функций.

Пусть имеется некоторый базовый класс MyClass, в котором объявлена виртуальная функция SetX(int _x). В производном классе MyDerived эта функция переопределяется, благодаря чему в основном модуле программы указателем на объект базового класса производится вызов реализации функции наследуемого класса.

```
// Заголовочный файл
#include <iostream.h>
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
public:
   int x;
   MyClass(){x = 0;}
   virtual void SetX(int _x){x = _x;}
   int GetX(){return x;}
};
class MyDerived : public MyClass
{
public:
   MyDerived(){};
   void SetX(int _x){x = _x; x++; cout << x << endl;}
   int GetX(){cout << x << endl; return x;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
int main()
{
   MyClass *pOb = new MyDerived;
   pOb->SetX(10);
   delete pOb;
   return 0;
}
```

Упражнение 16.2
Применение виртуального деструктора

Проанализируйте приведенный ниже пример.

Имеется некоторый базовый класс MyClass, при создании экземпляра которого выделяется область памяти под закрытую целочисленную переменную. В конструкторе наследуемого класса MyDerived также происходит выделение области памяти. Поскольку деструктор базового класса объявлен как виртуальный, после удаления объекта производного класса будет высвобождена область памяти, отведенная под базовый объект.

```
// Заголовочный файл
#include <iostream.h>
#ifndef _HEADER_H_
#define _HEADER_H_
class MyClass
{
   int *x;
public:
   MyClass(){x = new int; *x = 0;}
   virtual ~MyClass(){delete x;};
   void SetX(int _x){*x = _x;}
   int GetX(){return *x;}
};
class MyDerived : public MyClass
{
   int *y;
public:
   MyDerived(){y = new int; *y = 0;};
   ~MyDerived(){delete y;}
   void SetY(int _y){*y = _y;}
   int GetY(){return *y;}
};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
int main()
{
   MyDerived *pOb = new MyDerived;
   pOb->SetX(10);
   pOb->SetY(12);
   delete pOb;
   return 0;
}
```

Упражнение 16.3
Использование абстрактных классов

Проанализируйте пример использования абстрактных классов.

Пусть имеется некоторый класс MyPoint, содержащий закрытый целочисленный член-данные, интерфейсные функции для работы с этими данными, а также чисто виртуальную функцию void Out(). Таким образом, класс MyPoint является абстрактным и чисто виртуальная функция должна быть определена в классе-наследнике с тем, чтобы можно было создать объект наследуемого класса.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
#include <iostream.h>
class MyPoint
{
      int x;
public:
       MyPoint(){x=0;}
   virtual void SetX(int _x){x = _x;}
   virtual int GetX(){return x;}
   virtual void Out()=0;
};
class MyDer : public MyPoint
{
public:
   MyDer(){};
   MyDer(int _x):MyPoint(){SetX(_x);}
   virtual void Out()
   {
      cout << GetX() << endl;
   }
};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
int main()
{
   MyDer *pOb = new MyDer(10);
   pOb->Out();
   delete pOb;
   return 0;
}
```

Раздел 17
Исключения и информация о типе времени выполнения

Тема 17.1
Обработка исключительных ситуаций

Исключение – это некоторое событие, которое является неожиданным или прерывает нормальный процесс выполнения программы. Стандарт языка C++ предусматривает встроенный механизм обработки исключений. Кроме того, компиляторы Visual C++ 6.0 и Borland C++ Builder 5 предусматривают еще один механизм обработки исключений, который реализуется в тесном взаимодействии этих компиляторов с операционными системами Windows и Windows NT и называется *структурированной обработкой исключений* (SEH – от англ. Structured Exception Handling). Этот последний механизм был реализован еще в компиляторах языка C, в связи с чем часто называется *C-исключениями*. Хотя структурированная обработка исключений может быть использована в C++, для программ, написанных на этом языке, рекомендуется использовать более новый механизм обработки исключений C++ (C++ EH от англ. C++ Exception Handling). В дальнейшем всюду в этом разделе мы будем рассматривать обработку исключений C++.

Механизм обработки исключений C++ реализует так называемую *окончательную модель* управления: после того как исключение произошло, обработчик исключения не может потребовать, чтобы исполнение было продолжено с того места в коде, которое вызвало исключение. Исключения C++ также не поддерживают аппаратных (или асинхронных) исключений: перехватываются только исключения, выброшенные некоторой функцией. В контексте управления исключениями в C++ используются три ключевых слова: try, catch и throw.

Ключевое слово try служит для обозначения блока кода, который может генерировать исключение. При этом соответствующий блок заключается в фигурные скобки и называется *защищенным*, или try-*блоком*:

```
try
{
 //Защищенный блок кода
}
```

Тело всякой функции, вызываемой из try-блока, также принадлежит try-блоку. Предполагается, что одна или несколько функций или инструкций из защищенного блока могут *выбрасывать* (или *генерировать*) исключение. Если выброшено исключение, выполнение соответствующей функции или инструкции приостанавливается, все оставшиеся инструкции try-блока игнорируются и управление передается вне блока.

Ключевое слово catch следует непосредственно за try-блоком и обозначает секцию кода, в которую передается управление, если произойдет исключение. За ключевым словом следует *описание исключения,* состоящее из имени типа исключения и необязательной переменной, заключенное в круглые скобки:

```
catch(<тип_искл> <пер_искл>)
{
 //Обработчик исключения
}
```

Имя типа исключения идентифицирует обслуживаемый тип исключений. Блок кода, обрабатывающего исключение, заключается в фигурные скобки и называется catch-*блоком*, или *обработчиком исключения.* При этом говорят, что данный catch-блок *перехватывает* исключения описанного в нем типа. Если исключение перехвачено, переменная <пер_искл> получает его значение. Если Вам не нужен доступ к самому исключению, указывать эту переменную не обязательно. Переменная исключения может иметь любой тип данных, включая созданные пользователем типы классов.

За одним try-блоком могут следовать несколько catch-блоков. Оператор catch, с указанным вместо типа исключения многоточием, перехватывает исключения любого типа и должен быть последним из операторов catch, следующих за try-блоком.

Рассмотрим простой пример обработки исключения:

```
#include <exception>
#include <iostream>
using std::cout;
void func()
{
 //Функция генерирует исключение
}

int main()
{
 try
 {
  //Пример использования механизма
  //обработки исключений C++
  func();
  return 0;
 }
 catch (const char*)
 {
  //Обработчик исключения типа const char*
  cout << "Было выброшено исключение "
    << "типа const char*\n";
  return 1;
 }
 catch (...)
 {
  //Обработчик всех необработанных исключений
  cout << "Было выброшено исключение "
    << "другого типа\n";
  return 1;
 }
}
```

В рассматриваемом примере функция func() может генерировать исключения. В связи с этим она включена в защищенный блок. Перехват исключений осуществляется в двух catch-блоках. Первый из них перехватывает исключения, имеющие тип const char, второй – любого другого типа.

Тема 17.2
Генерирование исключений

Для обозначения в программе места и типа выбрасываемого исключения служит ключевое слово throw. Синтаксис его использования следующий:

```
throw <выраж>;
```

где <выраж> – это выражение, вычисленное значение которого инициализирует временный объект типа, определяемого типом этого выражения (то есть throw(T arg)). Вследствие этого полезно определить конструктор копирования для объекта типа исключения, если он содержит подобъекты. Местоположение инструкции throw определяет *точку выброса* исключения.

Операнд в инструкции throw не является обязательным. Инструкция throw без операнда используется для повторного выбрасывания исключения того же типа, которое обрабатывается в данный момент. Следовательно, она может использоваться только в catch-блоках.

Рассмотрим пример, демонстрирующий выбрасывание исключений:

```
#include <stdio.h>
bool test;
class SomeException{};
void Func(bool bvar)
{
 if (bvar) throw SomeException();
}
int main()
{
  try
  {
   test = true;
   Func(true);
  }
  catch(SomeException& e)
  {
   test = false;
  }
  return test ? (puts("test = true.\n"),1) :
                (puts("test = false.\n"),0);
}
```

При выполнении программа выводит на экран:

```
test = false
```

Следующий пример показывает, как может использоваться инструкция без параметра для повторного выбрасывания того же исключения:

```
#include <iostream>
using std::cout;
class SomeException{};
```

```
int main()
{
 try
 {
  try
  {
   throw SomeException();
  }
  catch(...)// Обрабатывать все исключения
  {
   //Частичная обработка исключения
   //Повторное выбрасывание того же исключения
   throw;
  }
 }
 catch(SomeException& e)
 {
  cout << " SomeException обработано.\n";
 }
 return 0;
}
```

Ключевое слово throw может использоваться не только как инструкция компилятору выбросить исключение, но и для спецификации исключений, которые может выбрасывать некоторая функция. *Спецификация исключений* имеет следующий формат:

```
throw(<тип1>, <тип2>, ...)
```

Она указывается после списка параметров функции. Если для функции заданы типы исключений, которые она может выбрасывать, это еще не означает, что она не может выбрасывать исключения других типов. Если такая функция все же выбрасывает исключение не предусмотренного в спецификации типа, это рассматривается как *неожиданное исключение* (unexpected exception) – особый вид исключений, который также может быть обработан. Этот вид исключений мы рассмотрим позже. Чтобы запретить функции выбрасывать какие-либо исключения, используется спецификация исключений без типа. С другой стороны, функции без спецификации могут выбрасывать исключения любого типа.

Замечание. Компилятор Visual C++ не поддерживает спецификации исключений для функций, как это предписано стандартом языка.

Рассмотрим пример:

```
#include <stdio.h>
bool test;
class ExClass{};
//Эта функция может выбрасывать исключения
//только типа ExClass
void ExThrowFunc(bool bvar) throw(ExClass)
{
 if(bvar) throw ExClass();
}
//Эта функция не может выбрасывать исключения
void TestFunc() throw()
{
 try
 {
  ExThrowFunc(true);
 }
 catch(ExClass& e)
 {
  test = true;
 }
}
int main()
{
 test = false;
 TestFunc();
 return test ?
 (puts("Функция TestFunc() обработала исключение."),1):
 (puts("Никаких исключений не произошло."),0);
}
```

При выполнении программа выводит на экран:

```
Функция TestFunc() обработала исключение.
```

Этот пример наглядно демонстрирует, что спецификация исключений без типа для некоторой функции запрещает распространение исключений за пределы данной функции. Однако ничто не мешает Вам задать исключение в самой этой функции, которое должно быть в ней же и обработано. Если функция ExThrowFunc() генерирует исключение, отличное от типа ExClass, оно рассматривается как неожиданное исключение.

Тема 17.3
Перехватывание исключений

Если при работе программы в некоторой точке выбрасывается исключение, поток выполнения программы прерывается и происходит следующее:

- если выброшена переменная встроенного типа или объект класса по значению, создается копия выброшенной переменной (причем в последнем случае используется конструктор копирования); если выброшена переменная по ссылке, копирование не производится;
- осуществляется поиск ближайшего обработчика, принимающего параметр, совместимый по типу с выброшенной переменной;
- если обработчик найден, стек раскручивается до этой точки; при этом вызываются деструкторы локальных объектов, которые выходят из области видимости;
- управление программой передается найденному обработчику;
- если обработчик не найден, Вы можете вызвать set_terminate(), предоставив обработчик завершения; в противном случае программа вызывает функцию завершения.

Если исключение не выброшено, программа выполняется обычным образом.

При поиске соответствующего типа исключения компилятор пользуется следующими правилами. Обработчик считается найденным, если:

- тип выброшенной переменной совпадает с типом, ожидаемым обработчиком; другими словами, если тип выброшенной переменной есть T, соответствующими ему признаются обработчики, перехватывающие исключение типа T, const T, T& и const T&;
- тип выброшенной переменной представляет собой указатель, тип которого может быть преобразован к типу указателя, перехватываемого обработчиком;
- если выброшенная переменная представляет собой объект некоторого класса, а тип перехватываемого исключения является базовым классом, наследуемым этим объектом как public.

Еще раз подчеркнем, что компилятор ищет *ближайший* обработчик, принимающий параметр, совместимый по типу с выброшенной переменной. Этот момент не следует забывать и внимательно следить за порядком, в котором располагаются в программе обработчики исключений для данного try-блока. Обработчик, ожидающий исключение базового класса, скрывает обработчик производного класса. Точно так же обработчик для

указателя типа void* скрывает обработчик для указателя любого типа. Рассмотрим пример:

```
#include <iostream.h>
class Base
{
};

class Derived : public Base
{
};
void FuncExDerived()
{
 Derived derOb;
 throw derOb;
 }
void FuncExErr()
{
 throw "Ошибка в функции FuncExErr()";
}
int main()
{
 try
 {
  FuncExDerived();
   }
 catch(Derived&)
 {
  cout << "Исключение перехвачено " << "в Derived&" << endl;
 }
 //Этот обработчик может скрывать предыдущий,
 //если будет помещен перед ним.
 catch(Base&)
 {
  cout << "Исключение перехвачено " < "в Base&" << endl;
 }
 try
 {
  FuncExErr();
 }
 catch(const char*)
 {
  cout << "Исключение перехвачено " << "в const char*" << endl;
 }

 //Этот обработчик может скрывать
 //предыдущий, если будет помещен перед ним.
```

```
 catch(void*)
 {
 cout << "Исключение перехвачено " << "в void*" << endl;
 }
 return 0;
}
```

При выполнении программа выводит на экран:

```
Исключение перехвачено в Derived&
Исключение перехвачено в const char*
```

Если переставить местами обработчики прерываний, результат будет совсем иной.

Тема 17.4
Использование вложенных блоков try/catch

Иногда возникает необходимость использования вложенных блоков try/catch. Для этого в языке C++ нет никаких препятствий, однако должно соблюдаться единственное требование: за каждым try-блоком обязательно должен стоять catch-блок. Рассмотрим пример:

```
#include <iostream.h>
class Base
{
};
class Derived : public Base
{
};
void FuncExDerived()
{
 Derived derOb;
 throw derOb;
}
int main()
{
 try
 {
  // ... Выполнение какой-то обработки
  try
  {
   FuncExDerived();
  }
  catch(Derived&)
  {
```

```
    cout << "Исключение перехвачено " << "в Derived&" << endl;
   }
   Base baseOb;
   throw baseOb;
  }
  catch(Base&)
  {
   cout << "Исключение перехвачено " << "в Base&" << endl;
  }
  return 0;
}
```

При выполнении программа выводит на экран:

```
Исключение перехвачено  в Derived&
Исключение перехвачено  в Base&
```

Однако гораздо чаще вложенные try/catch-блоки возникают неявно, в результате создания такого блока в функции, которая сама находится в защищенном блоке, как в следующем примере:

```
#include <iostream>
using namespace std;
void OtherFunc()
{
throw "Ошибка в OtherFunc()\n";
}
void FuncThrownEx()
{
 char* errStr = "Ошибка в FuncThrownEx()\n";
  try
  {
      throw errStr;
  }
  catch(char* s)
  {
   cout << s;
  }
  OherFunc();
}
void main()
{
  try
  {
   FuncThrownEx();
  }
  catch(char* s)
  {
   cout << s;
```

```
  }
}
```

При выполнении программа выводит на экран:

```
Ошибка в FuncThrownEx()
Ошибка в OtherFunc()
```

К вложенным try/catch-блокам приходится прибегать потому, что какая-то функция может совершенно непредвиденно для Вас привести к исключению. В этом случае простейший выход – заключить весь код в функции main() в такой блок, причем для перехвата исключения использовать обработчик вида catch(...). Затем можно использовать средства, предоставляемые компилятором для определения места в программе, вызвавшего появление исключения. Например, компилятор Borland C++ Builder 5 предоставляет для этого глобальные переменные __throwFileName, __throwLineNumber и __throwExceptionName.

Тема 17.5
Неожиданные исключения и обработка завершения

Если функция во время выполнения программы выбрасывает не специфицированное исключение (то есть отсутствующее в ее спецификации исключений), то говорят, что имеет место *неожиданное* (или *непредвиденное*) *исключение*. При этом вызывается функция unexpected(). Эта функция вызывает текущий обработчик неожиданных исключений, которым по умолчанию является функция terminate(). Вызов функции terminate() приводит в конечном итоге к завершению работы программы. Функция unexpected() имеет следующий прототип:

```
void unexpected();
```

Однако с помощью функции set_unexpected() Вы можете установить собственный обработчик, который будет вызываться, если функция выбрасывает исключение, отсутствующее в ее спецификации исключений.

Эта функция имеет следующий прототип:

```
unexpected_handler
set_unexpected(unexpected_handler ph) throw();
```

Здесь тип unexpected_handler описывает указатель на обработчик неожиданных исключений:

```
typedef void (*unexpected_handler)();
```

Функция set_unexpected() устанавливает в качестве нового обработчика неожиданных исключений функцию, на которую указывает параметр ph, и возвращает адрес предыдущего обработчика. Таким образом, ph не может быть равен NULL. Согласно объявлению, обработчик неожиданных исключений должен представлять собой функцию, которая не принимает параметров и ничего не возвращает. Эта функция не должна возвращать управление вызвавшему ее коду. Она может завершить выполнение программы одним из следующих способов:

- выбросив исключение типа, предусмотренного в спецификации исключений функции, вызвавшей появление неожиданного исключения;
- выбросив исключение типа bad_exception;
- вызвав одну из функций terminate(), abort() или exit().

Как уже упоминалось, обработчик по умолчанию вызывает функцию terminate().

Упомянутый выше класс bad_exception описывает исключения, которые могут быть выброшены из unexpected_handler, и определяется следующим образом:

```
class bad_exception: public exception{};
```

Таким образом, он, в свою очередь, является производным от базового класса exception всех исключений, выброшенных некоторыми выражениями и стандартной библиотекой C++. Класс exception объявлен следующим образом:

```
class exception
{
public:
  exception() throw();
  exception(const exception& rhs) throw();
  exception& operator = (const exception& rhs) throw();
  virtual ~exception() throw();
  virtual const char *what() const throw();
};
```

Значение строки, возвращаемой функцией-членом what() оставлено свободным для реализации. Поэтому в разных компиляторах оно различно.

Ниже приведен пример обработки неожиданного исключения:

```
#include <exception>
```

```
#include <iostream>
using namespace std;
// Функция не должна выбрасывать исключения
void f() throw()
{
 // Неожиданный выброс исключения
 //приводит к вызову  unexpected()
 throw "bad";
}
 void My_unexp_handler()
{
 exit(-1);
}
int main()
{
 set_unexpected(My_unexp_handler);
   f();
 cout << "Эта строка кода " << "никогда не достигается.";
 return 0;
}
```

Теперь рассмотрим, что происходит, если для выброшенного исключения не найден подходящий обработчик. В этом случае вызывается функция terminate(). По умолчанию функция terminate() вызывает функцию abort(), приводящую к аварийному завершению программы. Это поведение можно изменить, установив собственный обработчик завершения программы с помощью функции set_terminate(). Эта функция имеет следующий прототип:

```
terminate_function
set_terminate(terminate_function term_func);
```

Здесь тип terminate_function описывает указатель на функцию завершения:

```
typedef void (*terminate_function)();
```

Функция set_terminate() принимает единственный параметр – term_func – адрес предоставленной Вами функции завершения, а возвращает адрес предыдущей функции завершения. Вы можете вызвать функцию set_terminate() в любой точке программы перед тем, как выброшено исключение. Этой функции можно передать в качестве параметра NULL, чтобы восстановить поведение по умолчанию.

Функция terminate() имеет следующий прототип:

```
void terminate();
```

Предоставленная Вами функция завершения должна иметь такой же прототип. После выполнения любых желаемых действий по очистке памяти, она должна вызвать функцию exit() для выхода из программы (то есть она не должна возвращать управление вызвавшему ее коду или выбрасывать исключения). Если она этого не сделает, вызывается функция abort().

Рассмотрим пример:

```
#include <exception>
#include <iostream>
using std::cout;
void My_term_func();
void main()
{
 int i = 10, j = 0, res;
 set_terminate(My_term_func);
 try
 {
  if (j == 0)
    throw "Деление на нуль!";
  else
    res = i/j;
 }
 catch(int)
 {
  cout << "Перехватывает некоторое исключение типа integer.\n";
 }
 cout << "Эта строка никогда не будет " << "напечатана\n";
}
void My_term_func()
{
 cout << "My_term_func() была вызвана функцией terminate().\n";
 //Выполнение очистки памяти и других действий
 exit(-1);
}
```

При выполнении программа выводит на экран:

```
My_term_func() была вызвана функцией terminate().
```

Тема 17.6
Исключения и конструкторы

Конструктор класса может выбрасывать исключение, если он не может полностью сконструировать объект. В этом случае деструкторы вызываются только для полностью сконструирован-

ных локальных объектов и базовых классов. Процесс вызова деструкторов для объектов, созданных между началом защищенного блока и точкой выброса исключения, называется *раскруткой стека*. Если деструктор приводит к возникновению исключения в ходе раскрутки стека и не перехватывает его, вызывается функция terminate().

```
#include <iostream>
#include <exception>
using std::cout;
class Data
{
 public:
  Data(){cout << "Data::Data()\n" << "Throw exception\n";
                                  throw "Error.";}
  ~Data(){cout << "Data::~Data()\n";}
};
class Base
{
 public:
      Base(){cout << "Base::Base\n";}
      ~Base(){cout << "Base::~Base\n";}
};
class Derived: public Base
{
   Data data;
 public:
  Derived(){cout << "Derived::Derived()\n";}
  ~Derived(){cout << "Derived::~Derived\n";}
};
int main()
{
 try
 {
  Derived derOb;
  ...
 }
 catch(const char* s)
 {
  cout << s <<"\n";
 }
 return 0;
}
```

При выполнении эта программа выводит на экран:

```
Base::Base()
Data::Data()
```

```
Throw exception
Base::~Base()
Error.
```

Отсюда видно, что программа не вызвала деструкторы для не полностью сконструированных объектов Data и Derived.

Как отмечено выше, раскрутка стека ограничивается вызовом деструкторов для полностью сконструированных локальных объектов. Однако, если исключение выбрасывается из конструктора объекта, созданного динамически (то есть с помощью оператора new), память автоматически освобождается. Чтобы продемонстрировать это, модифицируем предыдущий пример:

```
#include <iostream>
#include <exception>
using std::cout;
class Data
{
 public:
   Data(){cout << "Data::Data()\n" << "Throw exception\n";
                                    throw "Error.";}
  ~Data(){cout << "Data::~Data()\n";}
};
class Base
{
 public:
     Base(){cout << "Base::Base\n";}
     ~Base(){cout << "Base::~Base\n";}
};

class Derived: public Base
{
   Data data;
 public:
   Derived(){cout << "Derived::Derived()\n";}
   ~Derived(){cout << "Derived::~Derived\n";}
 void* operator new(size_t size)
   {cout   << "Вызов оператора new для Derived\n";
   return ::operator new(size);}
 void operator delete(void* p) {cout << "Derived::delete\n";
::delete p;}
};
int main()
{
 try
 {
  Derived * pderOb = new Derived;
  delete pderOb;
 }
```

```
  catch(const char* s)
  {
   cout << s <<"\n";
  }
  return 0;
}
```

В рассматриваемом варианте программы объект Derived создается динамически. Кроме того, чтобы убедиться, что память для соответствующего объекта будет освобождена, мы перегрузили операторы new и delete. Теперь программа при выполнении выводит на экран:

```
Вызов оператора new для Derived
Base::Base()
Data::Data()
Throw exception
Base::~Base()
Derived::delete
Error.
```

Таким образом, для не полностью сконструированного объекта в этом случае автоматически был вызван оператор delete, что привело к освобождению памяти, распределенной для объекта pderOb.

Если Вы выполняете в конструкторе какие-то операции, которые могут привести к выбросу исключения, лучше поместить эти операции в защищенный блок специально предназначенной для этого функции, а не выбрасывать исключение в самом конструкторе. С другой стороны, язык C++ не позволяет конструктору или деструктору возвращать значение. Однако Вы можете использовать исключения, чтобы сообщить о возникшей ошибке.

Тема 17.7
Обработка отказа в выделении свободной памяти

Здесь мы рассмотрим, как обрабатываются отказы в выделении свободной памяти, запрошенной с помощью оператора new. В соответствии с требованиями языка Standard C++ по умолчанию оператор new должен генерировать исключение при невозможности удовлетворить запрос на выделение памяти, а в качестве альтернативной возможности допускается возвращение нулевого указателя. Если программа не обрабатывает это исключение, ее выполнение прекращается. К сожалению, требования к оператору new менялись за последние годы несколько раз, поэтому да-

же в современных компиляторах его поведение может отличаться от предписанного стандартом.

В более ранних версиях компилятора Visual C++ при невозможности удовлетворить запрос на выделение памяти оператор new возвращал нулевой указатель, а компилятор Borland C++ генерировал исключение xalloc. Сейчас стандарт языка предусматривает генерирование исключения bad_alloc. Для того чтобы выбрасывание исключения стало возможным, в программу нужно подключить заголовочный файл <new>. Класс bad_alloc объявлен следующим образом:

```
class bad_alloc: public exception
{
 public:
   bad_alloc(): exception() {}
   bad_alloc(const bad_alloc&){}
   bad_alloc& operator= (const bad_alloc&)
{return *this;}
   virtual ~bad_alloc();
   virtual const char* what() const
 {return __RWSTD::__rw_stdexcept_BadAllocException;}
};
```

Как уже отмечалось, стандарт языка допускает, чтобы оператор new возвращал нуль, а не выбрасывал исключение. Для этого предусмотрена следующая форма оператора new:

```
void* operator new
(size_t n, const nothrow&);
```

Такая форма оператора new может оказаться особенно полезной при компиляции устаревших программ.

Обработчик исключения bad_alloc, установленный по умолчанию, приводит к вызову функции terminate() и, как следствие, завершению работы программы. Приложение может заменить этот обработчик другим, который может попытаться освободить часть памяти в куче. Это можно сделать с помощью вызова функции set_new_handler(), которая объявлена в файле <new.h> следующим образом:

```
new_handler set_new_handler
(new_handler my_handler);
```

Эта функция устанавливает обработчик, переданный ей в качестве параметра, и возвращает указатель на предыдущий обработчик. К сожалению, тип new_handler объявляется в компилято-

рах различным образом. Компилятор Visual C++ требует, чтобы он объявлял указатель на функцию, которая принимает аргумент типа size_t и возвращает значение типа int:

```
typedef int (*new_handler)(size_t);
```

Этот тип определен в компиляторе с именем _PNH.

С другой стороны, компилятор Borland C++ Builder 5 требует, чтобы обработчик не принимал аргументов и возвращал void:

```
typedef void (* new_handler)();
```

Если Вы хотите, чтобы оператор new возвращал нуль в случае нехватки памяти, отмените обработчик по умолчанию, вызвав функцию set_new_handler() следующим образом:

```
set_new_handler(0);
```

Определенный пользователем обработчик my_handler должен выполнять одно из следующего:

- осуществлять возврат управления вызвавшему его коду после освобождения памяти в куче;
- выбрасывать исключение bad_alloc или производного от него типа;
- вызывать функцию abort() или exit().

Если обработчик возвращает управление вызвавшему его коду, то оператор new будет снова пытаться распределить память.

Рассмотрим следующий пример программы, обрабатывающей нехватку памяти:

```
#include <new>
#include <iostream>
using namespace std;
class BigClass
{
 public:
   BigClass() {};
   ~BigClass(){}
   long BigArray[100000000];
};
int main()
{
 try
 {
  BigClass* p = new BigClass;
 }
 catch(bad_alloc a)
```

```
{
 const char* temp = a.what();
 cout << temp << endl;
 cout << "Выброшено исключение bad_alloc"   << endl;
}
BigClass* q = new(nothrow) BigClass;
if ( q == NULL )
 cout << "Возвращен указатель NULL" << endl;
try
{
 BigClass * arr = new BigClass[3];
}
catch(bad_alloc a)
{
 const char * temp = a.what();
 cout << temp << endl;
 cout << "Выброшено исключение bad_alloc" << endl;
}

return 0;
}
```

Если Вы откомпилируете ее с помощью Borland C++ Builder 5, то при выполнении эта программа выведет на экран в полном соответствии с требованиями стандарта языка:

```
bad alloc exception thrown
Выброшено исключение bad_alloc
Возвращен указатель NULL
bad alloc exception thrown
Выброшено исключение bad_alloc
```

Попытка выполнить эту же программу в Visual C++ 6.0 приводит к неожиданному результату. Этот компилятор не выбрасывает исключения в случае нехватки памяти, а по-прежнему возвращает нулевой указатель. Чтобы добиться ожидаемого поведения, Вам придется написать что-нибудь вроде следующего:

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

using namespace std;

int my_new_handler(size_t)
{
 throw  bad_alloc();
 return 0;
}

class BigClass
{
```

```
  public:
    BigClass() {};
    ~BigClass(){}
    long BigArray[100000000];
};
int main()
{
 _PNH _old_new_handler;
 _old_new_handler =
 _set_new_handler(my_new_handler);
 try
 {
  BigClass* p = new BigClass;
 }
 catch(bad_alloc a)
 {
  const char * temp = a.what();
  cout << temp << endl;
  cout << "Выброшено исключение bad_alloc" << endl;
 }
 BigClass* q = new(nothrow) BigClass;
 if (q == NULL)
    cout << "Возвращен указатель NULL" << endl;
 try
 {
  BigClass* arr = new BigClass[3];
 }
 catch(bad_alloc a)
 {
  const char * temp = a.what();
  cout << temp << endl;
  cout << "Выброшено исключение bad_alloc" << endl;
 }
 set_new_handler(_old_new_handler);
 return 0;
}
```

При выполнении эта программа выводит на экран:

```
bad allocation
Выброшено исключение bad_alloc
Возвращен указатель NULL
bad allocation
Выброшено исключение bad_alloc
```

С другой стороны, тот же код на Borland C++ Builder 5 должен выглядеть несколько иначе:

```
void my_new_handler()
```

```
{
 throw std::bad_alloc();
}
...
new_handler _old_new_handler;
_old_new_handler =
set_new_handler(my_new_handler);
...
set_new_handler(_old_new_handler);
```

Тема 17.8
Информация о типе времени выполнения

Язык C++ предоставляет Вам широкие возможности по использованию указателей, которые реально указывают на объекты базовых классов. В связи с этим возникает задача идентификации типа объекта во время выполнения программы. *Информация о типе времени выполнения* (RTTI – Run-Time Type Information) предоставляется оператором typeid в виде ссылки на объект типа typeinfo. Синтаксис применения этого оператора следующий:

```
const typeinfo& id = typeid(typeName);
```

или

```
const typeinfo& id = typeid(objName);
```

Здесь typeName – имя типа объекта, а objName – имя объекта. Если оператору передается выражение, представляющее собой указатель или ссылку на *полиморфный* тип, он возвращает определенный во время выполнения программы (динамический) тип переданного ему объекта. Если же оператору передается указатель или ссылка на *не полиморфный* тип, он возвращает статический тип выражения. Под этим подразумевается, что оператор анализирует само переданное выражение и на этом основании приходит к заключению о его типе. В частности, если оператору передается переменная или выражение, значением которого является встроенный тип данных, он возвращает его тип.

Класс typeinfo определен в заголовочном файле typeinfo.h. В этом классе определен только private-конструктор, так что Вы не можете создавать объекты этого класса. Кроме того, в классе определены две функции, имеющие доступ public: name() и before(). Функция name() имеет следующий прототип:

```
const char* name() const;
```

Она возвращает строку, которая идентифицирует имя типа аргумента оператора typeid(). Функция before() имеет следующий прототип:

```
int before(const type_info&);
```

Эта функция используется для сравнения лексического порядка типов. Например, чтобы сравнить лексический порядок типов T1 и T2, используется следующая конструкция:

```
typeid(T1).before(typeid(T2));
```

Функция before() возвращает 0 или 1. Она редко используется в практике программирования.

Кроме этих двух функций общего доступа, класс содержит два оператора сравнения: равно (==) и не равно (!=). Они имеют следующие прототипы:

```
int operator==(const type_info&) const;
int operator!=(const type_info&) const;
```

Эти два оператора позволяют сравнивать два объекта типа type_info.

Следующая программа демонстрирует использование оператора typeid и функций name() и before()класса typeinfo.

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

class Base {};
class Derived: public Base {};

void main()
{
 char C;
 float X;
 Derived derOb;
 Base* pBase = &derOb;

 if (typeid(C) == typeid(X))
 cout << "C и X имеют одинаковый тип."<< endl;
 else cout << "C и X имеют разный тип."<< endl;
 cout << typeid(int).name();
 cout << " before " << typeid(double).name() << ": "
<< (typeid(int).before(typeid(double)) ? true : false) << endl;
 cout << typeid(double).name();
 cout << " before " << typeid(int).name() << ": "
<< (typeid(double).before(typeid(int)) ? true : false) << endl;
 if (typeid(*pBase) == typeid(Derived))
      cout << "Base и Derived имеют " << "одинаковый тип.\n";
 else
```

```
 cout << "Base и Derived имеют " <<"различный тип.\n";
 cout << typeid(Base).name();
 cout << " before " << typeid(Derived).name() << ": "
   << typeid(Base).before(typeid(Derived)) ? true : false) << endl;
}
```

При выполнении программа выводит на экран:

```
C и X имеют разный тип.
int before double: 0
double before int: 1
Base и Derived имеют различный тип.
Base before Derived: 1
```

В предыдущем примере оператор typeid() возвращал статическую информацию о типе. Рассмотрим теперь пример, когда он возвращает динамическую информацию о типе:

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

//Полиморфный класс
class PMorClass
{
 virtual void vFunc(){};
};

class Derived: public PMorClass {};

int main()
{
 Derived derOb;
 PMorClass* pOb;
 //Инициализируем указатель ссылкой
 //на объект производного класса
 pOb = &derOb;

 if (typeid(*pOb) == typeid(Derived))
   cout << "Имя типа есть " << typeid(*pOb).name();

 if (typeid(*pOb) != typeid(PMorClass))
   cout << "\nУказатель не-PMorClass-типа.";
 return 0;
}
```

При выполнении программа выводит на экран:

```
Имя типа есть Derived
Указатель не-PMorClass-типа.
```

Чаще всего оператор typeid() применяют к разыменованному указателю. В тех случаях, когда разыменовывается указатель NULL, этот оператор выбрасывает исключение bad_typeid.

Тема 17.9
Операторы приведения типа

Назначение RTTI значительно шире, чем только для предоставления возможности сравнения двух типов. Она также используется *для безопасного приведения типов*. Под безопасным приведением типа подразумевается, что приведение типа к заданному осуществляется только в том случае, когда эти типы связаны отношением наследования. В противном случае выбрасывается исключение или возвращается указатель NULL. В самом деле, обычное приведение типа выражения

```
(type_cast)expression
```

может приводить к непредсказуемым результатам. Рассмотрим следующий фрагмент кода:

```
struct One
{
  int x;
};
struct Two
{
  char Name[20];
};
void Func(One* p)
{
  Two* ptr = (Two*) p;
  cout << ptr.Name;
}
```

Приведение типа в функции Func() будет слепо произведено компилятором, однако при выполнении возникнут проблемы.

Оператор dynamic_cast предназначен для безопасного приведения типа указателя или ссылки в другой тип. Это осуществляется с использованием информации о типе времени исполнения. Если dynamic_cast не может осуществить приведение указателя к заданному типу, результатом является указатель NULL. Когда dynamic_cast не может осуществить приведение типа к ссылке или исходный ука-

затель есть NULL, он выбрасывает исключение bad_cast. Синтаксис использования этого оператора следующий:

```
dynamic_cast<newType*>(origPtr);
```

или

```
dynamic_cast<newType>(origRef);
```

Здесь newType – новый тип данных, а origPtr и origRef – исходный указатель и ссылка соответственно.

Поскольку dynamic_cast использует RTTI, то в качестве операндов могут задаваться только указатели или ссылки на полиморфные объекты. Такое приведение типов в основном применяется для нисходящего приведения типов, то есть для преобразования типа указателя (ссылки) на базовый класс в указатель (ссылку) на тип производного. Обратите внимание на использование в синтаксисе оператора угловых скобок вместо общепринятых круглых скобок.

Рассмотрим пример:

```
#include <iostream.h>
#include <typeinfo.h>
//Полиморфный класс
class Base1
{
 public:
   virtual void vFunc()
 {cout << "Вызвана vFunc()\n";};
};
//Не полиморфный класс
class Base2 {};
class Derived: public Base1,
public Base2 {};
int main()
{
  try
  {
   Derived derOb, *pderOb;
   Base1* pBase1;
   //Инициализируем указатель ссылкой
   //на объект производного класса
   pBase1 = &derOb;
   //Осуществляем нисходящее приведение типа
   //в старом (не безопасном стиле)
   pderOb = (Derived*)pBase1;
   //Если pBase1 не указывает на самом
   //деле на объект типа Derived, следующий вызов
```

```
    //закончится аварийным завершением
    pderOb->vFunc();
    //Осуществляем нисходящее приведение типа
    //и проверку осуществимости преобразования
    if ((pderOb = dynamic_cast<Derived*>(pBase1)) != 0)
      {
       cout << "Результирующий указатель " << "имеет тип "
            << typeid(pderOb).name() << endl;
    //Теперь вызов безопасен
    pDerOb->vFunc();
      }
  else throw bad_cast();
  Base2* pBase2;
  //Осуществляем нисходящее приведение типа с последующим
  //восходящим приведением типа к другому базовому классу
  //pBase1 = NULL;
  if ((pBase2 = dynamic_cast<Base2*>(pBase1)) != 0)
   {
    cout << "Результирующий указатель " << "имеет тип "
         << typeid(pBase2).name() << endl;
   }
    else throw bad_cast();
   }
   catch(bad_cast)
   {
    cout << "Отказ в dynamic_cast!" << endl;
    return 1;
   }
   return 0;
}
```

При выполнении программа выводит на экран:

```
Вызвана vFunc()
Результирующий указатель имеет тип Derived *
Вызвана vFunc()
Результирующий указатель имеет тип Base2 *
```

Этот пример содержит закомментированную строку:

```
pBase1 = NULL;
```

Откомментируйте ее и запустите программу снова. На этот раз будет выброшено исключение, и программа выведет на экран:

```
Вызвана vFunc()
Результирующий указатель имеет тип Derived *
Вызвана vFunc()
Отказ в dynamic_cast!
```

Оператор static_cast осуществляет приведение типов на основе статической информации (то есть на основе синтаксического анализа выражения). Синтаксис его использования следующий:

```
static_cast<newType>(varName);
```

Здесь newType – новый тип данных, а varName – имя переменной, приводимой к новому типу данных. Аргумент newType должен быть указателем, ссылкой, арифметического типа или enum–типа. Если тип varName может быть преобразован в заданный тип newType с помощью некоторого встроенного в язык преобразования, то выполнение такого преобразования с помощью static_cast дает тот же результат. Целочисленный тип может быть преобразован в enum-тип. Если значение целочисленного типа не попадает в диапазон значений enum-типа данных, результат операции не определен. Указатель NULL преобразуется сам в себя.

В основном этот оператор используется для нисходящего приведения типов в том случае, когда типы операндов не являются полиморфными. Это преобразование осуществимо, если newType – недвусмысленный производный класс для класса, экземпляром которого является varName (это означает, что существует однозначное преобразование newType в тип класса, объектом которого является varName).

Объект может быть явно преобразован к типу ссылки T&, если указатель на этот объект может быть явно преобразован в T*. Конструкторы и функции преобразования при этом не вызываются. Указатель на член класса может быть явно преобразован в указатель на другой член класса, только если они оба являются указателями на члены одного класса или указателями на члены классов, один из которых является недвусмысленным производным другого.

Рассмотрим пример:

```
#include <iostream.h>
#include <typeinfo.h>
//Не полиморфный класс
class Base {};
class Derived: public Base {};
int main()
{
  Derived derOb, *pderOb;
  Base* pBase;
  //Инициализируем указатель ссылкой
  //на объект производного класса
  pBase = &derOb;
  //Осуществляем нисходящее приведение типа
  if ((pderOb = static_cast<Derived*>(pBase)) != 0)
    {
     cout << "Результирующий указатель "<< "имеет тип "
```

```
    << typeid(pderOb).name() << endl;
  }
  return 0;
}
```

При выполнении программа выводит на экран:

```
Результирующий указатель имеет тип Derived *
```

Этот оператор может также использоваться для приведения встроенных типов данных:

```
#include <iostream>
using namespace std;
int main()
{
  int n;
  double d;
  d = 209.999;
  n = static_cast<int>(d);
  cout << "Значение n = " << n << endl;
  int* pi;
  pi = static_cast<int*>(NULL);
  cout << "Значение pi = " << pi << endl;
  int day;
  enum Weekend {Saturday, Sunday};
  Weekend we;
  day = static_cast<int>(Sunday);
  //Без предыдущего прebразования
  //следующая операция невозможна
  day += 3;
  cout << "Значение day = " << day;
  return 0;
}
```

При выполнении программа выводит на экран:

```
Значение n = 209
Значение pi = 0x00000000
Значение day = 4
```

Оператор const_cast используется, когда необходимо добавить или удалить квалификаторы const или volatile из типа объекта. Преобразование типа производится во время компиляции. Синтаксис его использования следующий:

```
const_cast<newType>(varName);
```

Здесь newType и varName – те же, что и выше.

Рассмотрим пример применения оператора const_cast.

```
void Func(const char* s)
{
```

```
 char* str;
 str = const_cast<char*>(s);
 strcpy(str, "А это - другая строка.");
}
void main()
{
 char* source = "Это - исходная строка.";
 cout << "До вызова Func() source = " << source << endl;
 Func(source);
 cout << "После вызова Func() source = " << source<< endl;
}
```

При выполнении программа выводит на экран:

```
До вызова Func() source = Это - исходная строка.
После вызова Func() source = А это - другая строка.
```

Оператор reinterpret_cast используется во всех случаях, не охватываемых предыдущими операторами преобразования. Синтаксис его использования следующий:

```
reinterpret_cast<newType>(varName);
```

Здесь newType и varName – те же, что и выше. Аргумент newType должен быть указателем, ссылкой, арифметического типа, enum-типа, указателем на функцию или указателем на член класса. Этот оператор позволяет преобразовывать указатель одного типа в указатель совершенно другого типа, а также преобразовывать указатель в целое число и наоборот. Преобразование указателя в целочисленный тип и обратно в тот же указатель дает исходное значение. Указатель на функцию может быть преобразован в указатель на объект некоторого класса при условии, что этот указатель имеет достаточно битов памяти для хранения исходного указателя на функцию. Для обратного преобразования также должно выполняться аналогичное требование.

Рассмотрим пример использования оператора reinterpret_cast:

```
#include <iostream>
using namespace std;
//Использование reinterpret_cast<Type> (expr) для замены
//приведения типа (Type)expr в случае небезопасных
//преобразований или преобразований, зависящих от реализации
void func(void* pv)
{
 //Приведение типа указателя к целому числу
 int i = reinterpret_cast<int>(pv);
 cout << "i = " << i << endl;
}
```

```
void main()
{
 int n;
 char* str = "строка";
 //Приведение типа указателя к целому числу и обратно
 n = reinterpret_cast<int>(str);
 str = reinterpret_cast<char*>(n);
 cout << "str = "<< str << endl;
 //Приведение типа целого числа к типу указателя
 func(reinterpret_cast<void *>(1024));
 //Приведение типа указателя на функцию одного
 //типа к указателю на функцию другого типа
 typedef void (* PFV)();
 PFV pfunc = reinterpret_cast<PFV>(func);
 pfunc();
}
```

При выполнении программа выводит на экран:

```
str = строка
i = 1024
i = 0
```

Заметим, что осуществляемое этим оператором преобразование может быть так же опасно, как и обычное приведение типа.

Практикум «Исключения и информация о типе времени выполнения»

Упражнение 17.1
Генерирование исключительных ситуаций

Проанализируйте приведенный ниже пример генерирования исключительной ситуации.

В начале программы объявлен некоторый класс MyException, а в функции main() осуществляется выбрасывание исключения типа MyException, а также его перехват в блоке catch(MyException& e).

```
// Основной модуль программы
#include <iostream.h>
class MyException{};
int main()
```

```
{
   try
   {
      throw MyException();
      return 0;
   }
   catch(MyException& e)
   {
      cout << "Handled by MyException.\n";
   }
}
```

Упражнение 17.2
Пример перехвата исключения

В данной самостоятельной работе Вам предлагается исправить приведенный код так, чтобы в примере работал перехват исключения MyDer&.

```
// Заголовочный файл
#ifndef _HEADER_H_
#define _HEADER_H_
#include <iostream.h>
class MyClass{};
class MyDer : public MyClass{};
#endif//_HEADER_H_
// Основной модуль программы
#include "header.h"
void FncThrow()
{
   MyDer dObject;
   throw dObject;
}
int main()
{
   try
   {
      FncThrow();
      return 0;
   }
   catch(MyClass&)
   {
      cout << "Handled by MyClass.\n";
   }
   catch(MyDer&)
   {
```

```
        cout << "Handled by MyDer.\n";
    }
}
```

Упражнение 17.3
Вложенные блоки try/catch

Проанализируйте следующий пример.

Имеется пустой класс MyException. В блоке try функции FncThrow() осуществляется генерирование исключительной ситуации типа MyException, а также перехват исключений любого типа с выводом соответствующего сообщения с последующим выбросом исключения типа MyException.

Вызов FncThrow() в функции main() приводит к тому, что блоки try/catch выполняются как вложенные.

```
// Основной модуль программы
#include <iostream.h>
class MyException{};
void FncThrow()
{
    try
    {
        throw MyException();
    }
    catch(...)
    {
        cout << "FncThrow() raised exception\n";
        throw MyException();
    }
}
int main()
{
    try
    {
        cout << "Begin main...\n";
        FncThrow();
    }
    catch(MyException& e)
    {
        cout << "Handled in main!\n";
    }
}
```

Раздел 18
Потоковый ввод-вывод

Тема 18.1
Предопределенные потоки

До сих пор в этой книге мы регулярно пользовались стилем ввода-вывода C++, не вникая в детали его реализации. Теперь самое время заняться изучением реализации потоков ввода-вывода. Наряду с функциями библиотеки времени выполнения, предназначенными для ввода-вывода, в C++ включен дополнительный набор объектно-ориентированных подпрограмм ввода-вывода. Главным преимуществом системы ввода-вывода C++ является то, что она может перегружаться для создаваемых Вами классов. В настоящее время используются две версии библиотеки ввода-вывода C++: старая, или традиционная, основанная на предварительном варианте стандарта C++, и новая, определенная международным стандартом Standard C++. Внешне эти версии почти не различаются. Однако внутренняя их реализация совершенно различна. Новая версия библиотеки широко использует шаблоны классов. Поскольку мы еще не рассмотрели использование шаблонов классов, эти детали реализации мы будем опускать.

Поток ввода-вывода – это логическое устройство, предназначенное для приема и выдачи информации пользователю. Поток связан с физическим устройством с помощью системы ввода-вывода C++. Тем самым поток обеспечивает пользователю единый интерфейс при работе с системой ввода-вывода. Это означает, что, например, для вывода информации на экран монитора и для записи ее в файл используется одна и та же функция. Когда программа на C++ начинает выполняться, автоматически создаются восемь предопределенных стандартных потоков. Эти стандартные потоки связаны со стандартными файлами stdin, stdout и stderr языка C показанным в табл. 18.1 образом.

Таблица 18.1
Предопределенные стандартные потоки и связанные с ними стандартные файлы языка С

Поток для «узких» символов	Поток для «широких» символов	Стандартный файл языка С
cin	wcin	stdin
cout	wcout	stdout
cerr	wcerr	stderr
clog	wclog	stderr

Как и стандартные файлы языка С, все эти потоки по умолчанию связаны с терминалом. Объект cin управляет вводом из буфера потока, связанного с объектом stdin, объявленным в <cstdio>. По умолчанию эти два потока синхронизированы. Объект cout управляет буфером потока, связанным с объектом stdout, объявленным в <cstdio>. По умолчанию эти два потока также синхронизированы. Поток clog – это просто буферизованная версия потока cerr. В буферизованной версии потока запись на реальное внешнее устройство делается, только когда буфер полон. Поэтому clog является более эффективным для перенаправления вывода в файл, в то время как cerr используется, главным образом, для вывода на экран терминала.

Система ввода-вывода содержит две иерархии классов: одну, предназначенную для работы с ASCII-символами, имеющими длину 8 бит, и другую, предназначенную для работы с UNICODE-символами, имеющими длину 16 бит. Символы первого набора называются «узкими», а второго – «широкими». Подробнее об UNICODE-символах Вы узнаете из Приложения. Стандартные потоки, перечисленные в столбце 2 табл. 18.1, предназначены для работы с «широкими» символами. На самом деле ситуация несколько сложнее, поскольку шаблонные классы представляют собой просто классы, содержащие типы данных в качестве параметров. Поэтому имена классов у каждой из иерархий одинаковы, но предназначены они для использования с разными наборами символов, которые им передаются в качестве параметра. В дальнейшем мы будем говорить о классах этих иерархий, употребляя только их имена. В основе иерархии потоковых классов лежит класс basic_ios, который в качестве своего подобъекта содержит абстрактный класс basic_streambuf. Последний является базовым

классом для создания буфера потока, который управляет передачей элементов в поток и из него для специализированных видов потоков.

Класс basic_ios является базовым для нескольких производных классов, среди которых классы basic_istream, basic_ostream и basic_iostream. Он содержит наиболее общие функции, необходимые для всех потоков, и обслуживает информацию о состоянии, которая отражает целостность потока и буфер потока. Этот класс также обслуживает связь потоковых классов с классами буферов потоков с помощью функции-члена rdbuf(). Классы, производные от basic_ios, специализируют операции ввода-вывода. В свою очередь, basic_ios использует класс ios_base, который также является базовым классом для всех потоковых классов. Он не зависит от типа символов и инкапсулирует информацию, необходимую для всех потоков. Эта информация включает в себя:

- управляющую информацию для синтаксического анализа и форматирования;
- дополнительную информацию для нужд пользователя (которая предоставляет путь для расширения потоков, как мы увидим позже);
- наполнение потока региональными (или локальными) символами.

Кроме того, этот класс определяет несколько типов данных, используемых всеми потоковыми классами, такие как флаги форматирования, биты состояния, режимы открытия файлов и т.д.

С целью обеспечения совместимости с традиционной версией библиотеки ввода-вывода C++ введены синонимы для имен потоковых классов, приведенные в табл. 18.2. Имена синонимов в точности соответствуют именам потоковых классов в традиционной версии библиотеки ввода-вывода C++. Далее при изложении потокового ввода-вывода и мы будем пользоваться этими именами, поскольку как раз эти имена следует указывать в программах.

Библиотека ввода-вывода содержит два параллельных семейства классов: одно – производное от streambuf и второе – производное от ios. Оба эти класса являются низкоуровневыми, выполняющими различный вид задач. Все потоковые классы имеют по крайней мере один из этих классов в качестве базового. Доступ из ios-производных классов к streambuf-производным осуществляется через указатель. Класс streambuf обеспечивает интерфейс с памятью и физическими устройствами. Функции-члены семей-

ства классов streambuf используются ios-производными классами. В табл. 18.3 приведено назначение классов потокового ввода-вывода.

Таблица 18.2
Потоковые классы и их синонимы

Шаблон класса	Синоним
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_ifstream	ifstream
basic_ofstream	ofstream
basic_fstream	fstream
basic_streambuf	streambuf

Таблица 18.3
Назначение классов потокового ввода-вывода. Абстрактный потоковый базовый класс

ios	Потоковый базовый класс
Потоковые классы ввода	
istream	Потоковый класс общего назначения для ввода, являющийся базовым классом для других потоков ввода
ifstream	Потоковый класс для ввода из файла
istream_with assign	Потоковый класс ввода для cin
istrstream	Потоковый класс для ввода строк
Потоковые классы вывода	
ostream	Потоковый класс общего назначения для вывода, являющийся базовым классом для других потоков вывода

ios	Потоковый базовый класс
ofstream	Потоковый класс для вывода в файл
ostream_with assign	Потоковый класс ввода для cout, cerr и clog
ostrstream	Потоковый класс для вывода строк
Потоковые классы ввода-вывода	
iostream	Потоковый класс общего назначения для ввода-вывода, являющийся базовым классом для других потоков ввода-вывода
fstream	Потоковый класс для ввода-вывода в файл
strstream	Потоковый класс для ввода-вывода строк
stdiostream	Класс для ввода-вывода в стандартные файлы ввода-вывода
Классы буферов для потоков	
streambuf	Абстрактный базовый класс буфера потока
filebuf	Класс буфера потока для дисковых файлов
strstreambuf	Класс буфера потока для строк
stdiobuf	Класс буфера потока для стандартных файлов ввода-вывода

Назначение почти всех классов следует из их названия. Классы группы _withassign являются производными соответствующих потоковых классов без этого окончания. Они перегружают оператор присваивания, что позволяет изменять указатель на используемый классом буфер. Если подключен заголовочный файл <iostream>, программы, написанные на языке C++, начинают выполняться с четырьмя открытыми предопределенными потоками, объявленными как объекты классов группы _withassign следующим образом:

```
//Соответствует stdin;
istream_withassign cin;
//Соответствует stdout;
ostream_withassign cout;
//Соответствует stderr;
ostream_withassign cerr;
//Буферизованный cerr;
ostream_withassign clog;
```

Замечание. В Microsoft Visual C++ предопределенные потоки cin, cout, cerr и clog инициализирует специально для этого предназначенный статический класс iostream_init.

Библиотека потоков C++ предоставляет несколько преимуществ в сравнении с функциями ввода-вывода библиотеки времени выполнения.

Безопасность типов. Давайте сравним вызов функций stdio с использованием стандартных потоков. Вызов stdio для чтения выглядит следующим образом:

```
int i = 25;
char name[50] = "Простая строка";
fprintf(stdout, "%d %s", i, name);
```

Он правильно напечатает:

```
25 Простая строка
```

Однако, если Вы по невнимательности поменяете местами аргументы для fprintf(), ошибка обнаружится только во время исполнения программы. Может произойти что угодно: от странного вывода до краха системы. Этого не может случиться в случае использования стандартных потоков:

```
cout << i << ' ' << name << '\n';
```

Так как имеются перегруженные версии оператора сдвига operator<<(), правый оператор всегда будет вызван. Функция cout << i вызывает operator<<(int), а cout << name вызывает operator<<(const char*). Следовательно, использование стандартных потоков является безопасным по типам данных.

Расширяемость для новых типов. Другим преимуществом стандартных потоков является то, что определенные пользователем типы данных могут быть без труда в них встроены. Рассмотрим тип Pair, который мы хотим напечатать:

```
struct Pair {int x; string y;}
```

Все, что нам нужно сделать, – это перегрузить оператор operator<<() для этого нового типа Pair, и мы сможем осуществлять вывод следующим образом:

```
Pair p(25, "December");
cout << p;
```

Соответствующий оператор operator<<() может быть реализован так:

```
ostream<char>&
operator<<(ostream<char>& o, const Pair& p)
{return o << p.x << ' ' << p.y;}
```

Простота и последовательность. Библиотека потоков поддерживает единообразный интерфейс ввода-вывода благодаря широкому использованию перегруженных функций и операторов. Это приводит к более простому и интуитивно понятному синтаксису.

Тема 18.2
Операции помещения и извлечения из потока

Вывод в поток выполняется с помощью *оператора вставки* (в поток), которым является перегруженный оператор сдвига влево <<. Левым его операндом является объект потока вывода. Правым его операндом может являться любая переменная, для которой определен вывод в поток (т.е. переменная любого встроенного типа или любого определенного пользователем типа, для которого она перегружена). Например:

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

приводит к выводу в предопределенный поток cout строки "Hello!".

Оператор << возвращает ссылку на объект ostream, для которого он вызван. Это позволяет строить цепочки вызовов оператора вставки в поток, которые выполняются слева направо:

```
int i = 5;
double d = 2.08;
cout << "i = " << i << ", d = " << d << '\n';
```

Эта инструкция приведет к выводу на экран следующей строки:

```
i = 5, d = 2.08
```

Оператор вставки в поток поддерживает следующие встроенные типы данных: bool, char, short, int, long, char* (рассматриваемый как строка), float, double, long double, and void*:

```
ostream_type& operator<<(bool n);
ostream_type& operator<<(short n);
ostream_type& operator<<(unsigned short n);
ostream_type& operator<<(int n);
ostream_type& operator<<(unsigned int n);
ostream_type& operator<<(long n);
ostream_type& operator<<(unsigned long n);
ostream_type& operator<<(float f);
ostream_type& operator<<(double f);
ostream_type& operator<<(long double f);
ostream_type& operator<<(const void *p);
```

Целочисленные типы форматируются в соответствии с правилами, принятыми по умолчанию, для функции printf() (если они не изменены путем установки различных флагов форматирования). Например, следующие две инструкции дают одинаковый результат:

```
int i;
long l;
cout << i << " " << l;
printf("%d %ld", i, l);
```

Тип void* используется для отображения адреса указателя:

```
int i;
//Отобразить адрес указателя в 16-ричной форме.
cout << &i;
```

Для ввода информации из потока используется *оператор извлечения*, которым является перегруженный оператор сдвига вправо >>. Левым операндом оператора >> является объект класса istream. Это позволяет строить цепочки инструкций извлечения из потока, выполняемых слева направо. Правым операндом может быть любой тип данных, для которого определен поток ввода.

```
istream_type& operator>>(bool& n);
istream_type& operator>>(short& n);
istream_type& operator>>(unsigned short& n);
istream_type& operator>>(int& n);
istream_type& operator>>(unsigned int& n);
istream_type& operator>>(long& n);
istream_type& operator>>(unsigned long& n);
istream_type& operator>>(float& f);
istream_type& operator>>(double& f);
istream_type& operator>>(long double& f);
istream_type& operator>>(void*& p);
```

По умолчанию оператор >> пропускает символы-заполнители, затем считывает символы, соответствующие типу заданной переменной. Пропуск ведущих символов-заполнителей устанавливается специально для этого предназначенным флагом форматирования. Рассмотрим следующий пример:

```
int i;
double d;
cin >> i >> d;
```

Последняя инструкция приводит к тому, что программа пропускает ведущие символы-заполнители и считывает целое число i. Затем она игнорирует любые символы-заполнители, следующие за целым числом, и считывает переменную с плавающей точкой d.

Для переменной типа char* (рассматриваемого как строка) оператор >> пропускает символы-заполнители и сохраняет следующие за ними символы, пока не появится следующий символ-заполнитель. Затем в указанную переменную добавляется нуль-символ.

Тема 18.3
Форматирование потока

До сих пор мы использовали для вывода информации во всех примерах форматы, заданные в C++ по умолчанию. Для управления форматированием библиотека ввода-вывода предусматривает три вида средств: *форматирующие функции, флаги* и *манипуляторы*. Все эти средства являются членами класса basic_ios и потому доступны для всех потоков.

Рассмотрим вначале *форматирующие функции-члены*. Их всего три: width(), precision() и fill().

По умолчанию при выводе любого значения оно занимает столько позиций, сколько символов выводится. Функция width() позволяет задать минимальную ширину поля для вывода значения. При вводе она задает максимальное число читаемых символов. Если выводимое значение имеет меньше символов, чем заданная ширина поля, то оно дополняется символами-заполнителями до заданной ширины (по умолчанию – пробелами). Однако если выводимое значение имеет больше символов, чем ширина отведенного ему поля, то поле будет расширено до нужного размера. Эта функция имеет следующие прототипы:

```
streamsize width(streamsize wide);
streamsize width() const;
```

Тип streamsize определен в заголовочном файле <iostream> как целочисленный. Функция с первым прототипом задает ширину поля wide, а возвращает предыдущее значение ширины поля. Функция со вторым прототипом возвращает текущее значение ширины поля. По умолчанию она равна нулю, то есть вывод не дополняется и не обрезается. В ряде компиляторов после выполнения каждой операции вывода значение ширины поля возвращается к значению, заданному по умолчанию.

Функция precision() позволяет узнать или задать точность (число выводимых цифр после запятой), с которой выводятся числа с плавающей точкой. По умолчанию числа с плавающей точкой выводятся с точностью, равной шести цифрам. Функция precision() имеет следующие прототипы:

```
streamsize precision(streamsize prec);
streamsize precision() const;
```

Функция с первым прототипом устанавливает точность равной prec и возвращает предыдущую точность. Функция со вторым прототипом возвращает текущую точность.

Замечание. Если не установлен флаг scientific или fixed (оба эти флага рассматриваются далее), то precision() задает общее число цифр.

Функция fill() позволяет прочесть или установить символ-заполнитель. Она имеет следующие прототипы:

```
char_type fill(char_type ch);
char_type fill() const;
```

Функция с первым прототипом устанавливает ch в качестве текущего символа-заполнителя и возвращает предыдущий символ-заполнитель. Функция со вторым прототипом возвращает текущий символ-заполнитель. По умолчанию в качестве символа-заполнителя используется пробел. Тип данных char_type является параметром класса basic_ios и может обозначать набор «узких» или «широких» символов.

Рассмотрим пример программы, в котором используются форматирующие функции:

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
```

```
{
 double x;
 cout.precision(4);
    cout.fill('0');
 cout << "  x          sqrt(x)     x^2\n\n";
 for (x=1.0; x<=6.0; x++)
  {
      cout.width(7);
      cout << x << " ";
      cout.width(7);
      cout << sqrt(x) << " ";
      cout.width(7);
      cout << x*x << '\n';
  }
 return 0;
}
```

Эта программа выводит на экран небольшую таблицу значений переменной x, ее квадратного корня и квадрата:

```
  x          sqrt(x)     x^2
0000001 00000001 0000001
0000002 0001.414 0000004
0000003 0001.732 0000009
0000004 0002.000 0000016
0000005 0002.236 0000025
0000006 0002.449 0000036
```

С каждым потоком связан набор *флагов*, которые управляют форматированием потока. Они представляют собой битовые маски, которые определены в классе ios как данные enum-типа fmt_flags. Сами флаги принадлежат типу fmtflags, который определен следующим образом:

```
typedef int     fmtflags;
```

Флаги форматирования и их назначение приведены в табл. 18.4.

Таблица 18.4

Флаги форматирования и их назначение

Флаг	Назначение
boolalpha	Значения булева типа вставляются и извлекаются в виде слов true и false
Hex	Значения целого типа преобразуются к основанию 16 (как шестнадцатеричные)

Флаг	Назначение
Dec	Значения целого типа преобразуются к основанию 10
Oct	Значения целого типа преобразуются к основанию 8 (как восьмеричные)
Fixed	Числа с плавающей точкой выводятся в формате с фиксированной точкой (т.е. nnn.ddd)
scientific	Числа с плавающей точкой выводятся в научной записи (т.е. n.xxxEyy)
showbase	Выводится основание системы счисления в виде префикса к целому числовому значению (например, число 1FE выводится как 0x1FE)
showpoint	При выводе значений с плавающей точкой выводится десятичная точка и последующие нули
showpos	При выводе положительных числовых значений выводится знак плюс
uppercase	Заменяет определенные символы нижнего регистра на символы верхнего регистра (символ "e" при выводе чисел в научной нотации на "E" и символ "x" при выводе 16-ричных чисел на "X")
Left	Данные при выводе выравниваются по левому краю поля
Right	Данные при выводе выравниваются по правому краю поля
internal	Добавляются символы-заполнители между всеми цифрами и знаками числа для заполнения поля вывода
skipws	Ведущие символы-заполнители (знаки пробела, табуляции и перевода на новую строку) отбрасываются
unitbuf	Выходной буфер очищается после каждой операции вставки в поток
adjustfield	= left | right | internal

Флаг	Назначение
basefield	= dec | oct | hex
floatfield	= scientific | fixed

Флаги left и right взаимно исключают друг друга. Флаги dec, oct и hex также взаимно исключают друг друга.

Прочесть текущие установки флагов позволяет функция-член flags() класса ios. Для этого используется следующий прототип этой функции:

```
fmtflags flags() const;
```

Функция flags() имеет и вторую форму, которая может применяться для установки значений флагов. Для этого используется следующий прототип этой функции:

```
fmtflags flags(fmtflags fmtfl);
```

В этом случае битовый шаблон копируется fmtfl в переменную, предназначенную для хранения флагов форматирования. Функция возвращает предыдущие значения флагов. Поскольку эта форма функции меняет весь набор флагов, она применяется редко. Вместо нее используется функция-член setf() класса ios, которая позволяет установить значение одного или нескольких флагов. Она имеет следующие прототипы:

```
fmtflags setf(fmtflags mask);
fmtflags setf
(fmtflags fmtfl, fmtflags mask);
```

Первая функция-член неявно вызывает функцию flags(mask | flags()) для установки битов, указанных параметром mask, и возвращает предыдущие значения флагов. Второй вариант функции присваивает битам, указанным параметром mask, значения битов параметра fmtfl, а затем возвращает предыдущие значения флагов. Например, следующий вызов функции setf() устанавливает для потока cout флаги hex и uppercase:

```
cout.setf(ios::hex | ios::uppercase);
```

Сбросить установленные флаги можно с помощью функции-члена unsetf() класса ios, имеющей следующий прототип:

```
void unsetf(fmtflags mask);
```

Она сбрасывает флаги, заданные параметром mask.

Следующий пример демонстрирует некоторые флаги:

```
#include <iostream>
using namespace std;
int main()
{
 double d = 1.321e9;
 int    n = 1024;
 //Вывести значения
 cout << "d = " << d << '\n';
 cout << "n = " << n << '\n';
 //Изменить флаги и вывести значения снова
 cout.setf(ios::hex | ios::uppercase);
 cout.setf(ios::showpos);
 cout << "d = " << d << '\n';
 cout << "n = " << n << '\n';
 return 0;
}
```

При выполнении программа выводит на экран:

```
d = 1.321e+09
n = 1024
d = +1.321E+09
n = +1024
```

Система ввода-вывода C++ предусматривает еще один способ форматирования потока. Этот способ основан на использовании *манипуляторов ввода-вывода*. Список манипуляторов и их назначение приведены в табл. 18.5. Манипуляторы ввода-вывода представляют собой просто вид функций-членов класса ios, которые, в отличие от обычных функций-членов, могут располагаться *внутри* инструкций ввода-вывода. В связи с этим ими пользоваться обычно удобнее.

Таблица 18.5

Манипуляторы ввода-вывода и их назначение

Манипулятор	Использование	Назначение
boolalpha	ввод-вывод	Устанавливает флаг boolalpha
dec	ввод-вывод	Устанавливает флаг dec
endl	вывод	Вставляет символ новой строки и очищает буфер

Манипулятор	Использование	Назначение
ends	вывод	Вставляет символ конца строки
fixed	вывод	Устанавливает флаг fixed
flush	вывод	Очищает буфер потока
hex	ввод-вывод	Устанавливает флаг hex
internal	вывод	Устанавливает флаг internal
left	вывод	Устанавливает флаг left
noboolalpha	ввод-вывод	Сбрасывает флаг boolalpha
noshowbase	вывод	Сбрасывает флаг noshowbase
noshowpoint	вывод	Сбрасывает флаг noshowpoint
noshowpos	вывод	Сбрасывает флаг noshowpos
noskipws	ввод	Сбрасывает флаг noskipws
nounitbuf	вывод	Сбрасывает флаг nounitbuf
nouppercase		Сбрасывает флаг
oct	ввод-вывод	Устанавливает флаг oct
Resetiosflags (ios_base:: fmtflags mask)	ввод-вывод	Сбрасывает ios-флаги в соответствии с mask
right	вывод	Устанавливает флаг right
scientific		Устанавливает флаг scientific
Setbase (int base)	ввод-вывод	Задает основание системы счисления для целых
Setfill (charT c)	ввод-вывод	Устанавливает символ-заполнитель
Setiosflags (ios_base:: fmtflags mask)	ввод-вывод	Устанавливает ios-флаги в соответствии с mask
Setprecision (int n)	ввод-вывод	Устанавливает точность чисел с плавающей точкой

Манипулятор	Использование	Назначение
setw(int n)	ввод-вывод	Устанавливает минимальную ширину поля
showbase	вывод	Устанавливает флаг showbase
showpoint	вывод	Устанавливает флаг showpoint
showpos	вывод	Устанавливает флаг showpos
skipws	ввод	Устанавливает флаг skipws
unitbuf	вывод	Устанавливает флаг unitbuf
uppercase	вывод	Устанавливает флаг uppercase
ws	ввод	Устанавливает пропуск символов-заполнителей

За исключением setw(), все изменения в потоке, внесенные манипулятором, сохраняются до следующей установки.

При внимательном изучении таблицы можно заметить, что манипуляторы охватывают функциональные возможности, предоставляемые обычными функциями-членами и флагами форматирования. В частности, манипулятор setiosflags() реализует те же функциональные возможности, что и функция-член setf(), а манипулятор resetiosflags() – те же, что и функция-член unsetf().

Для доступа к манипуляторам с параметрами необходимо включить в программу стандартный заголовочный файл <iomanip>. При использовании манипулятора без параметров скобки за ним не ставятся, так как на самом деле он представляет собой указатель на функцию-член, который передается перегруженному оператору <<.

Рассмотрим пример, демонстрирующий использование манипуляторов.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
int main()
{
 double x;
 cout << setprecision(4);
 cout << setfill('0');
```

```
 cout << "     x sqrt(x)    x^2\n\n";
 for (x=1.0; x<=6.0; x++)
   {
    cout << setw(7) << x << " ";
    cout << setw(7) << sqrt(x) << " ";
    cout << setw(7) << x*x << "\n";
   }
 return 0;
}
```

Этот пример функционально полностью эквивалентен приведенному ранее, но для управления форматом вывода использует манипуляторы, а не функции форматирования.

Тема 18.4
Файловый ввод-вывод с использованием потоков

Для осуществления операций с файлами библиотека ввода-вывода предусматривает три класса: ifstream, ofstream и fstream. Эти классы являются производными, соответственно, от классов istream, ostream и iostream. Поскольку эти последние классы, в свою очередь, являются производными от класса ios, классы файловых потоков наследуют все функциональные возможности своих родителей (перегруженные операции << и >> для встроенных типов, функции и флаги форматирования, манипуляторы и пр.). Для реализации файлового ввода-вывода нужно включить в программу заголовочный файл <fstream>.

Существует небольшое отличие между использованием предопределенных и файловых потоков. Файловый поток должен быть связан с файлом прежде, чем его можно будет использовать. С другой стороны, предопределенные потоки могут использоваться сразу после запуска программы, даже в конструкторах статических классов, которые выполняются даже раньше вызова функции main(). Вы можете позиционировать файловый поток в произвольную позицию в файле, в то время как для предопределенных потоков это обычно не имеет смысла.

Для создания файлового потока эти классы предусматривают следующие формы конструктора:

- создать поток, не связывая его с файлом:

```
explicit ifstream();
explicit ofstream();
explicit fstream();
```

- создать поток, открыть файл и связать поток с файлом:

```
explicit ifstream(const char *name, ios::openmode mode = ios::in);
explicit ofstream(const char * name,
ios::openmode mode = ios::out | ios::trunc);
explicit fstream(const char * name,
ios::openmode mode = ios::in | ios::out);
```

Чтобы открыть файл для ввода или вывода, можно использовать вторую форму нужного конструктора

```
fstream fs("FileName.dat");
```

или вначале создать поток с помощью первой формы конструктора, а затем открыть файл и связать поток с открытым файлом, вызвав функцию-член open(). Эта функция определена в каждом из классов потокового ввода-вывода и имеет следующие прототипы:

```
void ifstream::open(const char *name,
ios::openmode mode = ios::in);
void ofstream::open(const char * name,
ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char * name,
ios::openmode mode = ios::in | ios::out);
```

Здесь name – имя файла, mode – режим открытия файла. Параметр mode является перечислением и может принимать значения, указанные в табл. 18.6.

Таблица 18.6

Режимы открытия и их назначение

Режим открытия	Назначение
ios::in	Открыть файл для чтения
ios::out	Открыть файл для записи
ios::ate	Начало вывода устанавливается в конец файла
ios::app	Окрыть файл для добавления в конец
ios::trunc	Усечь файл, т.е. удалить его содержимое
ios::binary	Двоичный режим операций

Режимы открытия файла представляют собой битовые маски, поэтому Вы можете задавать два или более режима, объединяя их операцией ИЛИ. В следующем фрагменте кода файл открывается для вывода с помощью функции open():

```
ofstream ofs;
ofs.open("FileName.dat");
```

Обратите внимание, что по умолчанию режим открытия файла соответствует типу файлового потока. У потока ввода или вывода флаг режима всегда установлен неявно. Например, для потока вывода в режиме добавления файла можно вместо инструкции

```
ofstream
ofs("FName.txt",ios::out | ios::app);
```

написать следующую:

```
ofstream ofs("FName.txt", ios::app);
```

Между режимами открытия файла ios::app и ios::ate имеется небольшая разница. Если файл открывается в режиме добавления, весь вывод в файл будет осуществляться в конец файла, безотносительно к операциям позиционирования в файле. В режиме открытия ios::ate (от англ. "at end") Вы можете изменить позицию вывода в файл и осуществлять запись, начиная с нее. Для потоков вывода режим открытия эквивалентен ios::out | ios::trunc, то есть Вы можете опустить режим усечения файла. Однако для потоков ввода-вывода его нужно указывать явно. Файлы, которые открываются для вывода, создаются, если они еще не существуют.

Если открытие файла завершилось неудачей, объект, соответствующий потоку в булевом выражении, будет возвращать false:

```
if (!osf)
  {
   cout << "Файл не открыт\n";
  }
```

Проверить успешность открытия файла можно также с помощью функции-члена is_open(), имеющей следующий прототип:

```
bool is_open() const;
```

Функция возвращает true, если поток удалось связать с открытым файлом. Например:

```
if (!osf.is_open())
 {
  cout << "Файл не открыт\n";
 }
```

Если при открытии файла не указан режим ios::binary, файл открывается в текстовом режиме, и после того как файл был успешно открыт, для выполнения операций ввода-вывода можно использовать операторы извлечения и вставки в поток. Можно даже использовать функции ввода-вывода, принятые в языке С, такие как fprintf() и fscanf(). Для проверки, достигнут ли конец файла, можно использовать функцию eof() класса ios, имеющую следующий прототип:

```
bool eof() const;
```

Завершив операции ввода-вывода, необходимо закрыть файл, вызвав функцию-член close():

```
ofs.close();
```

Функция close() не имеет параметров и возвращаемого значения:

```
void close();
```

Закрытие файла происходит автоматически при выходе объекта потока из области видимости, когда вызывается деструктор потока.

Рассмотрим пример, демонстрирующий файловый ввод-вывод с использованием потоков:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  int n = 50;
  //Открываем файл для вывода
  ofstream ofs("Test.txt");
  if (!ofs)
  {
   cout << "Файл не открыт.\n";
   return 1;
  }
  ofs << "Hello!\n" << n;
  //Вывод другои информации в файл
  //...
  //Закрываем файл
```

```
    ofs.close();
    //открываем тот же файл для ввода
    ifstream file("Test.txt");
    if (!file)
    {
     cout << "Файл не открыт.\n";
     return 2;
    }
    char str[80];
    file >> str >> n;
    cout << str << " " << n << endl;
    //Закрываем файл
    file.close();
    return 0;
}
```

Следует иметь в виду, что стандартная библиотека ввода-вывода отличается от традиционной. Этот факт нужно учитывать при переносе старых программ в современные системы программирования. Во-первых, в традиционной библиотеке функция open() имела третий параметр, задававший режим защиты файла. Во-вторых, конструктор потока fstream и функция open() для него не предусматривали установки по умолчанию режимов открытия ios::in | ios::out. В-третьих, стандартная библиотека не поддерживает режимы открытия ios::noncreate и ios::noreplace, которые были в традиционной.

Тема 18.5
Неформатируемый ввод-вывод

Когда Вы открываете файл в текстовом режиме, происходит следующее:

- при вводе каждая пара символов '\r'+'\n' (возврат каретки + перевод строки) преобразуется в символ перевода строки ('\n');
- при выводе каждый символ перевода строки ('\n') преобразуется в пару '\r'+'\n' (возврат каретки + перевод строки).

Это не всегда удобно. Если Вы собираетесь использовать выводимый файл для последующего ввода в программу (возможно, другую), лишние байты информации ни к чему. С этой целью система ввода-вывода предоставляет Вам возможность осуществления неформатируемого ввода-вывода, то есть записи и чтения двоичной информации (иногда говорят – сырых данных). Для

осуществления ввода-вывода в двоичном режиме включите флаг ios::binary в параметр open_mode, передаваемый конструктору потока или функции open(). Чтение двоичной информации из файла осуществляется функцией read(), которая имеет следующий прототип:

```
istream_type&
read(char_type* s, streamsize n);
```

Здесь параметр s задает буфер для считывания данных, а параметр n – число читаемых символов.

Запись двоичных данных осуществляет функция-член write():

```
ostream_type&
write(const char_type* s, streamsize n);
```

Эта функция получает n символов из буфера, адрес которого задан параметром s, и вставляет их в поток вывода. Рассмотрим пример:

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 int x = 255;
 char str[80] = "Тестирование двоичного ввода-вывода.";
 //Открываем файл для вывода в двоичном режиме
 ofstream ofs("Test.dat");
 if (!ofs)
  {
   cout << "Файл не открыт.";
   return 1;
  }
 ofs.write((char*)&x, sizeof(int));
 ofs.write((char*)&str, sizeof(str));
 //Открываем файл для вывода в двоичном режиме
 ifstream ifs("Test.dat");
 if (!ifs)
  {
   cout << "Файл не открыт.";
   return 1;
  }
 ifs.read((char*)&x, sizeof(int));
 ifs.read((char*)&str, sizeof(str));
 cout << x << '\n' << str << '\n';
 return 0;
}
```

Тема 18.6
Часто применяемые функции

Помимо уже описанных функций, библиотека ввода-вывода C++ содержит широкий набор различных функций. Здесь мы приведем лишь некоторые, наиболее часто употребляемые из них. Большинство этих функций используется для неформатируемого ввода-вывода.

Для извлечения символа из потока можно использовать функцию-член get() потока istream. Она имеет много прототипов, однако чаще других используются следующие два:

```
int_type get();
istream_type& get(char_type& c);
```

Тип int_type определен как целочисленный. Приведем пример использования функции get():

```
#include <iostream>
using std::cout;
int main()
{
 char ch;
 cout << "Введите число. "
 << "Для завершения ввода нажмите <ENTER>:";
  while (cin.get(ch))
  {
   // Проверка на код клавиши <ENTER>
   if (ch == '\n') break;
  }
 return 0;
}
```

Для вставки символа в поток вывода используется функция put(), которая имеет следующий прототип:

```
ostream& put(char ch);
```

Функция get() может также использоваться для чтения строки символов. В этом случае используются ее варианты, определяемые следующими прототипами:

```
istream_type&
get(char_type* str, streamsize len,
char_type delim);
istream_type&
get(char_type* str, streamsize len);
```

Эта функция извлекает из входного потока символы в буфер str, пока не встретится символ-разграничитель delim (по умолчанию – перевод строки) или не будет прочитано len-1 символов

либо конец файла. Символ-разграничитель не извлекается из входного потока.

Ввиду того что функция get() не извлекает из входного потока символ-разграничитель, она используется редко. Гораздо чаще используется функция getline(), которая извлекает из входного потока символ-разграничитель, но не помещает его в буфер. Она имеет следующие прототипы:

```
istream_type& getline(char_type* str,
streamsize len, char_type delim);
istream_type& getline(char_type* str,
streamsize len);
```

Здесь параметры имеют то же назначение, что и в функции get().

Функция gcount() возвращает число символов, извлеченных из потока последней операцией неформатируемого ввода (т.е. функцией get(), getline() или read()). Она имеет следующий прототип:

```
streamsize gcount() const;
```

Рассмотрим пример, в котором используются две последние функции:

```
#include <iostream>
using namespace std;
void main()
{
 char *name;
 int len  = 100;
 int count = 0;
 name = new char[len];
 cout << "Введите свое имя:";
 cin.getline(name, len);
 count = cin.gcount();
 //Уменьшаем значение счетчика на 1, т.к.
 //getline() не помещает разграничитель в буфер
 cout << "\nЧисло прочитанных символов: " << count - 1;
}
```

Для того чтобы пропустить при вводе несколько символов, используется функция ignore():

```
istream_type& ignore(streamsize n=1,
int_type delim=traits::eof());
```

Эта функция игнорирует вплоть до n символов во входном потоке. Пропуск символов прекращается, если она встречает символ-разграничитель, которым по умолчанию является символ конца файла. Символ-разграничитель извлекается из входного потока.

Функция peek(), имеющая следующий прототип:

```
int_type peek();
```

позволяет «заглянуть» во входной поток и узнать следующий вводимый символ. При этом сам символ из потока не извлекается.

С помощью функции putback(), имеющей прототип:

```
istream_type& putback(char_type ch);
```

можно вернуть символ ch в поток ввода.

При выполнении вывода данные не сразу записываются в файл, а временно хранятся в связанном с потоком буфере, пока он не заполнится. Функция flush() позволяет вызвать принудительную запись в файл до заполнения буфера. Эта функция имеет следующий прототип:

```
ostream_type& flush();
```

Она неявно используется манипулятором endl. Этот манипулятор вставляет в поток символ перевода строки и очищает буфер. Таким образом, инструкция

```
cout << endl;
```

эквивалентна следующим:

```
cout << '\n'; cout.flush();
```

Функция rdbuf() позволяет получить указатель на связанный с потоком буфер. Эта функция имеет следующий прототип:

```
streambuf_type* rdbuf() const;
```

Наконец, функция setbuf() позволяет связать с потоком другой буфер. Она имеет такой прототип:

```
void setbuf(char* buf, streamsize n);
```

Здесь buf указатель на другой буфер длины n.

Тема 18.7
Файлы с произвольным доступом

Произвольный доступ в системе ввода-вывода реализуется с помощью функций seekg() и seekp(), используемых для позиционирования, соответственно, входного и выходного потоков. Каждая из них имеет по два прототипа:

```
istream& seekg(pos_type pos);
istream_type& seekg(off_type& offset,
ios_base::seekdir dir);
ostream& seekp(pos_type pos);
ostream_type& seekp(off_type offset, ios_base::seekdir dir);
```

Здесь параметр pos задает абсолютную позицию в файле относительно начала файла. Параметр offset задает смещение в файле, а параметр dir – направление смещения, которое может принимать значения:

- ios::beg – смещение от начала файла;
- ios::cur – смещение относительно текущей позиции;
- ios::end – смещение от конца файла.

С каждым потоком связан указатель позиционирования, который изменяет свое значение в результате операции ввода или вывода. Для выполнения операций произвольного доступа файл должен открываться в двоичном режиме.

Получить текущее значение позиции в потоке ввода или вывода можно с помощью функций tellg() и tellp() соответственно. Эти функции имеют следующие прототипы:

```
pos_type tellg();
pos_type tellp();
```

Следующий пример демонстрирует возможность позиционирования потока ввода информации:

```
#include <iostream>
#include <fstream>

using namespace std;

void main(int argc, char* argv[])
{
 int size = 0;
 if (argc > 1)
  {
   const char *FileName = argv[1];
   ifstream file;
   file.open(FileName, ios::in | ios::binary);
   if (file)
    {
      file.seekg(0, ios::end);
      size = file.tellg();
      if (size < 0)
       {
          cerr << FileName << " не найден.";
          return;
```

```
            }
            cout << FileName << " size = " << size;
        }
    }
    else cout << "Вы не задали имя файла.";
}
```

Программа выводит на экран длину заданного файла.

Тема 18.8
Опрос и установка состояния потока

Класс ios поддерживает информацию о состоянии потока после каждой операции ввода-вывода. Текущее состояние потока хранится в объекте типа iostate, который объявлен следующим образом:

```
typedef int iostate;
```

Состояния потока являются элементами перечислимого типа io_state, который может иметь значения, представленные в табл. 18.7.

Таблица 18.7
Состояния потока и их значения

Состояние	Значение
Goodbit	Ошибок нет
Eofbit	Достигнут конец файла
Failbit	Имеет место ошибка форматирования или преобразования
Badbit	Имеет место серьезная ошибка

Для опроса и установки состояния потока можно использовать функции-члены класса ios. Имеется два способа получения информации о состоянии операции ввода-вывода. Во-первых, можно вызвать функцию rdstate(), имеющую следующий прототип:

```
iostate rdstate() const;
```

Функция возвращает состояние операции ввода-вывода. Во-вторых, можно воспользоваться одной из следующих функций-членов:

```
bool good() const;
bool eof() const;
bool fail() const;
bool bad() const;
```

Каждая из этих функций возвращает true, если установлен соответствующий бит состояния (точнее, функция fail() возвращает true, если установлен бит failbit или badbit).

Если прочитано состояние, которое сигнализирует об ошибке, его можно сбросить с помощью функции clear():

```
void  clear(iostate state = ios::goodbit);
```

Установить нужное состояние можно с помощью функции setstate():

```
void setstate(iostate state);
```

Кроме перечисленных функций, класс ios содержит функцию приведения типа

```
operator void*() const;
```

(она возвращает NULL, если установлен бит badbit) и перегруженный оператор логического отрицания

```
bool operator!() const;
```

(он возвращает true, если установлен бит badbit). Это позволяет сравнивать выражения, в которые входит поток или его отрицание с нулем, то есть писать выражения вида:

```
while(!strmOb.eof())
{
 //...тело цикла while
}
```

Следующий пример иллюстрирует получение информации о состоянии ввода-вывода.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char* argv[])
{
  char c;
  if (argc > 1)
  {
   ifstream ifs(argv[1]);
   if (!ifs)
```

```
    {
     cout << "Файл не открыт\n";
     return 1;
    }
    while (!ifs.eof())
    {
     ifs.get(c);
     //Контроль состояния потока
     if (ifs.fail())
       {
        cout << "Ошибка!";
        break;
       }
     cout << c;
    }
    ifs.close();
   }
   return 0;
}
```

В этом примере осуществляется ввод символов из файла, заданного в командной строке при запуске программы. Если при извлечении символов встречается ошибка, чтение прекращается и выводится сообщение об этом.

Тема 18.9
Ошибки потоков

По умолчанию поток ввода-вывода не выбрасывает никаких исключений.

Вы должны явно активизировать исключение сами. Для этой цели поток содержит маску исключений. Каждый из флагов этой маски соответствует одному из флагов ошибки (состояния потока). Ключевую роль в выбрасывании исключений играют две функции-члена класса ios:

```
iostate exceptions() const;
```

Эта функция возвращает маску, которая определяет, какие флаги состояния, установленные в маске флагов состояния, будут вызывать выбрасывание исключения. Она неявно использует вызов функции rdstate(). Другая функция:

```
void exceptions(iostate except);
```

устанавливает маску исключений потока в значение, определяемое переменной except, а затем вызывает clear(rdstate()) для

сброса, возможно, установленных битов маски состояния потока. Например, если флаг badbit установлен в маске исключений, исключение будет выброшено, если этот флаг будет установлен в маске состояния потока. Следующий пример демонстрирует, как можно активизировать исключение в потоке ввода:

```
try
{
 ifs.exceptions(ios::badbit | ios::failbit);
 in >> x;
   ...
}
catch(ios::failure& e)
{
 cerr << e.what() << endl;
 throw;
}
```

При вызове функции exceptions() Вы указываете, какие флаги состояния потока будут вызывать выбрасывание исключения. Объекты, выброшенные потоковыми операциями ввода-вывода, являются производными от исключения типа ios::failure. Заметим, что любое изменение состояния потока будет приводить к выбрасыванию исключения. Это происходит потому, что функции setstate() и exception() приводят к выбрасыванию исключения, если в маске исключений установлен соответствующий бит. Следовательно, оператор catch, приведенный выше, будет ловить все исключения, связанные с потоками. Рекомендуется активизировать исключение по флагу badbit и подавить исключения по флагам eofbit и failbit, поскольку они, вообще говоря, не представляют исключительные состояния.

Тема 18.10
Перегрузка операторов извлечения и вставки

Одним из главных преимуществ потоков ввода-вывода является их расширяемость для новых типов данных. Вы можете реализовать операторы извлечения и вставки для своих собственных типов данных. Чтобы избежать неожиданностей, ввод-вывод для определенных пользователем типов данных должен следовать тем же соглашениям, которые используются операторами извлечения и вставки для встроенных типов данных. Рассмотрим пример перегрузки операторов извлечения и вставки в поток для оп-

ределенного пользователем типа данных, которым является следующий класс даты:

```
class Date
{
 public:
 Date(int d, int m, int y);
 Date(const tm& t);
 Date();
 private:
 tm tm_date;
};
```

Этот класс содержит закрытое данное-член типа tm, которое представляет собой структуру для хранения даты и времени, определенную в заголовочном файле <ctime>. Чтобы осуществить ввод-вывод пользовательского типа данных, какими являются объекты класса Date, нужно перегрузить операторы извлечения и вставки в поток для этого класса. Приведем соответствующее объявление класса Date:

```
class Date
{
 public:
   Date(int d, int m, int y);
   Date(tm t);
   Date();
 private:
   tm tm_date;
   friend istream& operator>>(istream& is, Date& dat);
   friend ostream& operator<<(ostream& os, const Date& dat);
};
```

Реализуем операторы извлечения и вставки для объектов класса Date.

Возвращаемым значением для операторов извлечения (и вставки) должна являться ссылка на поток, чтобы несколько операторов могли быть выполнены в одном выражении. Первым параметром должен быть поток, из которого будут извлекаться данные, вторым параметром – ссылка или указатель на объект определенного пользователем типа. Чтобы разрешить доступ к закрытым данным класса, оператор извлечения должен быть объявлен как дружественная функция класса. Ниже приведен оператор извлечения из потока для класса Date:

```
istream&
operator>>(istream& is, Date& dat)
```

```
{
  is >> dat.tm_date.tm_mday;
  is >> dat.tm_date.tm_mon;
  is >> dat.tm_date.tm_year;
  return is;
}
```

Те же самые замечания верны и для оператора вставки. Он может быть построен аналогично. Единственное отличие заключается в том, что Вы должны в него передать константную ссылку на объект типа Date, поскольку оператор вставки не должен модифицировать выводимые объекты. Ниже приведена его реализация для класса Date:

```
ostream&
operator<<(ostream& os,const Date& dat)
{
  os << dat.tm_date.tm_mon << '/';
  os << dat.tm_date.tm_mday << '/';
  os << dat.tm_date.tm_year ;
  return os;
}
```

Следуя соглашениям о вводе-выводе для потоков, мы можем теперь осуществлять извлечение и вставку объектов класса Date следующим образом:

```
Date birthday(24,10,1985);
cout << birthday << '\n';
```

или

```
Date date;
cout << "Пожалуйста, "
  << "введите дату (день, месяц, год)" << '\n';
cin  >> date;
cout << date << '\n';
```

Приведем теперь пример полностью:

```
#include <iostream>
#include <ctime>

using namespace std;

class Date
{
 public:
   Date(int d, int m, int y)
   {
    tm_mday = d;
```

```
    tm_mon = m;
    tm_year = y;
   };
   Date(tm t){tm_date = t;};
   Date()
   {
    tm_mday = 01;
    tm_mon =  00;
    tm_year = 00;
   }
 private:
   tm tm_date;
   friend istream& operator>>(istream& is, Date& dat);
   friend ostream& operator<<(ostream& os, const Date& dat);
};
istream&
operator>>(istream& is, Date& dat)
{
  is >> dat.tm_date.tm_mday;
  is >> dat.tm_date.tm_mon;
  is >> dat.tm_date.tm_year;
  return is;
}
ostream&
operator<<(ostream& os,const Date& dat)
{
  os << dat.tm_date.tm_mon << '/';
  os << dat.tm_date.tm_mday << '/';
  os << dat.tm_date.tm_year ;
  return os;
}
void main()
{
 Date date;
 cout << "Пожалуйста, "
      << "введите дату (день, месяц, год)" << '\n';
 cin  >> date;
 cout << date << '\n';
}
```

Приведем еще один пример, демонстрирующий перегрузку операторов извлечения и вставки в поток, на этот раз для структуры:

```
#include <iostream.h>
struct info {
  char *name;
  double val;
  char *units;
  };
```

```
//Вы можете перегрузить оператор << для вывода так:
ostream& operator << (ostream& s, info& m)
{
 s << m.name << " " << m.val << " " << m.units;
 return s;
};
//Вы можете перегрузить оператор >> для ввода так:
istream& operator >> (istream& s, info& m)
{
 s >> m.name >> m.val >> m.units;
 return s;
};
int main(void)
{
 info x;
 x.name = new char[15];
 x.units = new char[10];
 cout << "\nВведите наименование, "
   << "значение и единицы изм.:";
 cin >> x;
 cout << "\nВведено:" << x;
 return(0);
}
```

Тема 18.11
Переадресация ввода-вывода

Вы можете переназначить имена cin или cout файловым потокам. Это позволяет легко проводить отладку ввода-вывода, переадресовывая ввод-вывод вместо файла на экран. Следующий пример демонстрирует эту возможность:

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char* argv[])
{
 char str[80];
 //Создаем файловый поток
 ofstream ofs;
 //Если в командной строке задан аргумент
 cout << "Введите имя и фамилию:\n";
 cin.getline(str, sizeof(str));
 if (argc > 1)
 {
 //Открываем файл с заданным именем
```

```
    ofs.open(argv[1]);
    //Если файл успешно открыт
    if (ofs)
    //Переадресовываем вывод
    cout = ofs;
  }
  cout << "Привет, " << str << '!' << endl;
  return 0;
}
```

Если при запуске программы в командной строке задано имя файла, то вывод осуществляется в этот файл, в противном случае на экран терминала.

Тема 18.12
Резидентные в памяти потоки

Язык C++ поддерживает два вида ввода-вывода: файловый ввод-вывод и резидентные в памяти потоки. Файловый ввод-вывод предполагает передачу данных из внешнего устойства или в него. Напротив, резидентные потоки не предполагают работу с внешними устройствами. В этом случае не требуется преобразования кодов и выполнения операций с внешним устройством: осуществляется только форматирование информации. Результат такого форматирования сохраняется в памяти и может быть восстановлен в виде символьной строки. Класс istrstream предназначен для чтения символов из массива в памяти. Он использует private-объект strstreambuf для управления связанным с ним массивом. Поскольку он является производным от basic_istream, он может использовать функции для форматированного и неформатированного ввода. Класс ostrstream предназначен для записи массива в память. Он также использует private-объект strstreambuf для управления связанным с ним массивом. Поскольку он является производным от basic_ostream, он также может использовать функции для форматированного и неформатированного вывода. Класс strstream позволяет читать и записывать в массив символов в памяти и является производным от класса basic_iostream. Все эти классы определены в заголовочном файле <strstream>. В контексте операций ввода-вывода массивы символов в памяти часто называют *резидентными в памяти потоками* (строк).

Для работы с резидентным в памяти потоком нужно сначала связать этот поток с некоторой областью памяти (массивом сим-

волов). Затем весь ввод-вывод в этот массив можно выполнять с помощью операторов излечения и помещения в поток. Чтобы открыть массив для ввода, из него нужно создать объект класса istrstream и тем самым связать нужный поток с заданным массивом. Этот класс описывает объект, который управляет извлечением элементов из stream_buffer класса strstreambuf и объявлен следующим образом:

```
class istrstream: public istream
{
 public:
   explicit istrstream(const char *s);
   explicit istrstream(char *s);
   istrstream(const char *s, streamsize n);
   istrstream(char *s, streamsize n);
   strstreambuf *rdbuf() const;
   char *str();
};
```

Таким образом, для создания объекта класса istrstream можно использовать один из четырех объявленных конструкторов. Конструкторы

```
istrstream
<поток_ввода>(char *s, streamsize n);
```

и

```
istrstream
<поток_ввода>(const char *s, streamsize n);
```

используются, когда ввод осуществляется из буфера (которым служит массив), заданного своим указателем s, в том случае, если буфер не заканчивается нулевым символом. В этом случае параметр n задает размер буфера. Если буфер заканчивается нулевым символом, можно воспользоваться более простыми формами конструктора:

```
istrstream <поток_ввода>(const char *s);
```

и

```
istrstream <поток_ввода>(char *s);
```

Для вывода информации в символьный массив нужно создать объект класса ostrstream, который объявлен следующим образом:

```
class ostrstream: public ostream
{
 public:
```

```
    ostrstream();
    ostrstream(char *s, streamsize n, ios_base::openmode mode =
                                        ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz = true);
    char *str();
    streamsize pcount() const;
};
```

Этот класс описывает объект, который управляет вставкой элементов в stream_buffer класса strstreambuf.

Обычно для вывода информации в символьный массив используется конструктор вида:

```
ostrstream
<поток_вывода>(char *s, streamsize n,
                    ios_base::openmode mode = ios_base::out);
```

Здесь <поток_вывода> – это поток, который связывается с буфером s, заданным своим указателем (которым является символьный массив), n – размер буфера, mode – задает режим открытия потока вывода. По умолчанию поток открывается для вывода, поэтому этот параметр можно опускать. Функция-член str() возвращает указатель на буфер и «замораживает» массив. При использовании динамических объектов вызов str() делает динамический буфер Вашей собственностью. Это означает, что в дальнейшем Вы должны будете удалить этот буфер или вернуть его в собственность потока ostrstream, вызвав

```
<поток_вывода>->rdbuf()->freeze(0);
```

Отметим, что перед использованием функции str() нужно явным образом вывести в поток нулевой символ, иначе она возвратит указатель на строку, не ограниченную нулем. Другая функция-член – pcount() – возвращает число записанных в буфер байтов информации.

Чтобы открыть массив для ввода и вывода, создайте объект класса strstream:

```
strstream
<поток_вв>(char *s, streamsize n, ios_base::openmode mode =
    ios_base::in | ios_base::out);
```

Здесь <поток_вв> – это поток ввода-вывода, который для выполнения этих операций связывается с массивом s (который служит буфером) длиной n. После того как Вы создали нужный объект, Вы можете пользоваться всеми ранее описанными функция-

ми ввода-вывода, включая функции ввода-вывода для двоичных файлов и для файлов с произвольным доступом. Рассмотрим пример использования резидентного в памяти потока:

```
#include <iostream>
#include <strstream>

using namespace std;

int main()
{
 char buf[] = "Тест 9 50.32 *";
 char str[80];
 int n;
 float f;
 char c;
 //Открываем резидентный в памяти поток для ввода
 istrstream istr(buf);
 istr >> str >> n >> f >> c;
 cout << str << ' ' << n << ' ' << f << ' '  << c << endl;
 return 0;
}
```

В этом примере содержимое буфера извлекается в переменные, а затем значения переменных выводятся на экран.

Заметим, что, наряду с перечисленными тремя потоками, для работы с резидентными в памяти потоками могут использоваться три аналогичных им потока istringstream, ostringstream и stringstream. Эти потоки во всем аналогичны рассмотренным, за исключением того, что они используют буфер типа basic_stringbuf (то есть работают со строками типа string) и требуют для их использования подключения стандартного файла <sstream>.

Практикум
«Потоковый ввод-вывод»

Упражнение 18.1
Работа с форматирующими функциями

В предлагаемом Вашему вниманию упражнении необходимо составить текст программы, выводящей на консоль таблицу соответствия первых 16-ти чисел 16-ричного формата представления (от 0x00 до 0x0F включительно) цифрам 10-чного формата с использованием функций форматирования потоков и флагов.

После этого необходимо видоизменить программу таким образом, чтобы в форматировании потока участвовали манипуляторы ввода-вывода.

Упражнение 18.2
Использование потоков в файловом вводе-выводе

Проанализируйте следующий пример.

Пусть необходимо составить текст программы, выводящей в некоторый лог-файл App.log информацию обо всех печатных символах системы с последующим чтением этих данных и выводом их на консоль.

Для этой цели нам понадобятся две потоковые переменные, одна из которых (OutFile) будет связана с лог-файлом для записи в него информации, а вторая (InFile) – для последующего чтения записанных в него данных.

В начале программы производится открытие файла App.log для вывода, переменная OutFile проверяется на равенство нулю, определяя результат операции открытия. В случае неудачи программа завершается с кодом 1. При успешном создании лог-файла в него будет записано символическое представление чисел от 33 до 255, которые также будут выведены на консоль. Завершает операцию вывода в файл закрытие его функцией-членом close().

После этого необходимо открыть тот же файл уже для ввода информации из него. С этой целью создается объект InFile класса ifstream. После проверки корректности открытия лог-файла следует циклическое чтение записанной в него информации с последующим выводом ее на консоль. Завершает работу программы закрытие файла App.log.

```
#include <iostream.h>
#include <fstream.h>
int main()
{
  ofstream OutFile("App.log");
  if(!OutFile)
  {
    cout << "Файл не открыт!";
    return 1;
  }
  cout << "Out file:\n";
  for(int i=33; i<=255; i++)
```

```
    {
        cout << (char)i << ' ';
        OutFile << (char)i;
    }
    OutFile.close();
    cout << endl << endl;
    ifstream InFile("App.log");
    if(!InFile)
    {
        cout << " Файл не открыт!";
        return 1;
    }
    char ch;
    cout << "In file:\n";
    for(int i=33; i<=255; i++)
    {
        InFile >> ch;
        cout << ch << ' ';
    }
    InFile.close();
    return 0;
}
```

Упражнение 18.3
Неформатируемый ввод-вывод

Модифицируйте предыдущий пример таким образом, чтобы в программе использовался неформатируемый ввод-вывод.

Раздел 19
Стандартный класс string

Тема 19.1
Конструкторы строк

До сих пор мы работали со строками, которые представляют собой массивы символов, заканчивающихся символом '\0'. Однако Standard C++ предоставляет другую возможность для работы со строками – класс string стандартной библиотеки шаблонов (Standard Template Library). Эта библиотека стала частью языка. Более подробно мы с ней ознакомимся в следующей главе, а сейчас предметом нашего иссследования станет класс string. На самом деле, этот класс является специализацией базового класса basic_string для 8-разрядных символов типа char:

```
typedef basic_string <char> string;
```

Другим производным классом для него является класс wstring, который предназначен для работы с 16-разрядными символами:

```
typedef basic_string <wchar_t> wstring;
```

Поскольку обычно используются 8-разрядные символы, здесь мы рассмотрим только версию string базового класса basic_string. Чтобы получить доступ к классу string, к программе нужно подключить заголовочный файл <string>. Стандарт языка требует, чтобы функции работы с массивами символов (strcpy(), strcat() и пр.) были объявлены в заголовочном файле <cstring>. Однако ряд современных компиляторов по-прежнему объявляет их в заголовочном файле <string.h>. Не следует путать эти два файла.

В классе string поддерживается много различных конструкторов.

```
string();                                                  Форма 1
string(const string &s);                                  Форма 2
string(const string &s, size_t start, size_t n = npos);   Форма 3
string(const char* cp);                                   Форма 4
```

```
string(const char* cp, size_t start, size_t n = npos);   Форма 5
string(char c);                                          Форма 6
string(size_t n = npos, char c);                         Форма 7
string(signed char c);                                   Форма 8
string(size_t n = npos, signed char c);                  Форма 9
string(unsigned char c);                                 Форма 10
string(size_t n = npos,  unsigned char c);               Форма 11
```

Первая форма конструктора – это конструктор по умолчанию. Он создает строку нулевой длины. Вторая и третья формы – это конструкторы копирования. Четвертая форма создает копию символьной строки, на которую указывает параметр cp. Пятая форма конструктора аналогична предыдущей, но копируются только первые n символов. В шестой форме создается строка, содержащая один символ c. Седьмая форма создает строку, содержащую символ c, повторенный n раз. Следующие две пары конструкторов повторяют шестую и седьмую формы для случая, когда параметр имеет тип signed char или unsigned char. Ряд форм конструкторов содержит значение параметра n по умолчанию, равное npos. Это значение определено следующим образом:

```
const size_t npos = size_t(-1);
```

Эта константа задает максимально возможную длину строки.

Приведем несколько примеров объявления и инициализации строк типа string. Простейшая форма объявления переменной типа string состоит в указании ее имени и начального значения (если это нужно). Конструктор копирования позволяет объявить эту переменную, проинициализировав ее значением другой такой же переменной:

```
string s1;
string s2("строка");
string s3 = "начальное значение";
string s4(s3);
```

Память для этой переменной может быть также выделена динамически:

```
string* ps5 = new string;
string* ps6 = new string(("строка 2");
string& ps7 = *new string("ссылка на строку");
```

В этих случаях длина строки равна числу инициализирующих символов. Другая форма конструктора позволяет Вам установить длину строки и инициализировать ее копиями одного символа. Например:

```
string s7 (10, '\n');
```

Переменая s7 инициализируется как строка из 10 символов со значением '\n' (перевод строки).

Тема 19.2
Изменение величины строки и ее емкости

Текущее значение длины строки дает функция-член size(), а текущую емкость объекта-строки возвращает функция-член capacity(). Емкость объекта-строки может быть изменена функцией reserve().Функция-член max_size() возвращает максимальное значение строки, для которой может быть распределена память:

```
cout << s6.size() << endl;
cout << s6.capacity() << endl;
//Изменяем емкость строки до 200
s6.reserve(200);
cout << s6.capacity() << endl;
cout << s6.max_size() << endl;
```

Функция-член length() – это просто синоним для size(). Функция-член resize() изменяет величину строки либо отсекая символ в конце, либо вставляя новые. Необязательный второй аргумент функции resize() может использоваться для указания символа, который будет вставляться в новые позиции. Например, следующая инструкция

```
s7.resize(15, '\t');
```

вызывает изменение величины строки и заполнение новых позиций символами табуляции '\t'.

Функция-член empty() возвращает true, если строка пуста:

```
if (str.empty())
   cout << "Строка пуста" << endl;
```

Тема 19.3
Присваивание, добавление и обмен строк

Переменной типа string может быть присвоено значение другой такой же переменной, литеральной строки (символьного массива) или одного символа:

```
str1 = str2;
str2 = "Новое значение";
str3 = 'a';
```

С любой из этих форм аргумента можно использовать оператор += , который добавляет строку, записанную справа, в конец строки, записанной слева. Например, выполнение инструкции

```
str3 += "bc";
```

приводит к тому, что s3 теперь содержит строку "abc". Функции-члены более общего назначения assign() и append() позволяют Вам задать значение подстроки, которая будет использована для присваивания строке, вызвавшей ее, или для добавления к ней. Аргументы pos и n указывают, что в операции участвуют n символов, начиная с позиции pos:

```
//Присвоить первые три символа строки s2 строке s4
s4.assign (s2, 0, 3);
//Добавить к s4 три символа строки s2, начиная с позиции 2
s4.append (s2, 2, 3);
```

Оператор сложения + используется для конкатенации (объединения) строк. Этот оператор создает копию левого операнда, а затем добавляет к ней содержимое правого операнда:

```
cout << (s2 + s3) << endl;
```

Функция-член swap() позволяет поменять местами содержимое двух строк типа string:

```
//Меняет местами содержимое строк s4 and s5
s5.swap (s4);
```

Тема 19.4
Доступ к символам строки

Доступ к отдельным символам строки обеспечивается с использованием индекса этого символа в строке. Функция-член at() является почти синонимом этого оператора, за исключением того, что она выбрасывает исключение out_of_range, если затребованное значение индекса выходит за пределы строки или равно ее величине:

```
//Вывод символа из позиции 2
cout << s4[2] << endl;
//Эквивалентен предыдущему
cout << s4.at(2) << endl;
```

Функция-член c_str() возвращает указатель на завершающуюся нулем символьную строку, элементы которой совпадают с элементами, содержащимися в переменной типа string. Это по-

зволяет использовать string-строки с функциями, которые требуют указателя на символьный массив. Полученный указатель объявлен как константа. Это означает, что Вы не можете использовать c_str() для модификации строки. Приведем пример использования этой функции:

```
char d[256];
strcpy(d, s4.c_str());
```

Функция-член data() возвращает указатель на используемый string-объектом символьный буфер.

Тема 19.5
Копирование строк и подстроки

Функция-член copy() имеет следующий прототип:

```
size_t
copy(char* s, size_t n, size_t pos = 0)
const;
```

Она заменяет элементы символьного массива, указатель на который передан ей в качестве первого аргумента копиями элементов из строки. Второй аргумент задает длину копируемой строки. Третий аргумент задает позицию в исходной строке, начиная с которой выполняется копирование символов. По умолчанию копируются все символы строки. В символьный массив s завершающий символ нуль не дописывается. Если указанная позиция pos>size(), выбрасывается исключение out_of_range. Приведем пример использования этой функции:

```
s3.copy (s4, 2, 3);
```

Эта инструкция вызывает копирование в символьный массив s4 двух символов из строки s3, начиная с третьего.

Функция-член substr() возвращает строку, которая представляет собой часть исходной строки:

```
string
substr(size_t pos = 0, size_t  n = npos)
const;
```

Параметр pos задает позицию в исходной строке, начиная с которой создается подстрока. Параметр n задает число символов исходной строки, которые будут участвовать в образовании подстроки. По умолчанию подстрока создается, начиная с нулевой позиции. Например:

```
cout << s4.substr(3) << endl;
cout << s4.substr(3, 2) << endl;
```

Тема 19.6
Сравнение строк

Класс string содержит удобные методы сравнения благодаря перегруженным операторам:

```
bool operator==(const string& s1, const string& s2);
bool operator==(const string& s, const char* cp);
bool operator==(const char* cp, const string& s);
bool operator!= (const string& s1, const string& s2);
bool operator!= (const string& s, const char* cp);
bool operator!= (const char* cp, const string& s);
bool operator>(const string& s1, const string& s2);
bool operator>(const string& s, const char* cp);
bool operator>(const char* cp, const string& s);
bool operator<(const string& s1, const string& s2);
bool operator<(const string& s, const char* cp);
bool operator<(const char* cp, const string& s);
bool operator<=(const string& s1, const string& s2);
bool operator<=(const string& s, const char* cp);
bool operator<=(const char* cp, const string& s);
bool operator>=(const string& s1, const string& s2);
bool operator>=(const string& s, const char* cp);
bool operator>=(const char* cp, const string& s);
```

Как можно видеть, в сравнениях могут участвовать строки типа string или строка и массив символов.

Функция-член compare() используется для выполнения лексикографического сравнения двух строк:

```
int
compare (const string& str);
int
compare (size_t pos, size_t n,
const string& str) const;
```

Параметр pos задает начальную позицию в массиве символов, а n – число символов в массиве символов, участвующих в операции сравнения. Функция возвращает положительное значение, если вызывающая строка (лексикографически) больше, чем строка, заданная параметром s. Если они равны, она возвращает ноль. Наконец, она возвращает отрицательное значение, если вызывающая строка меньше.

Заметим, что функция используется достаточно редко ввиду наличия в классе удобных перегруженных операторов сравнения.

Тема 19.7
Операции поиска

Функция-член find() определяет первое появление строки, массива символов или символа, переданного ей в качестве первого аргумента, в текущей строке. Эта функция имеет следующие прототипы:

```
size_type
find (const basic_string& str, size_type pos = 0) const;
size_type
find (const char* s, size_type pos, size_type n) const;
size_type
find (const char* s, size_type pos = 0) const;
size_type
find (char c, size_type pos = 0) const;
```

Необязательный параметр pos позволяет задать начальную позицию для поиска. Если первое вхождение строки найдено, функция возвращает позицию первого символа в текущей строке, с которого начинается совпадение. В противном случае она возвращает значение npos. Например:

```
s1 = "mississippi";
//возвращает 2
cout << s1.find("ss") << endl;
//возвращает 5
cout << s1.find("ss", 3) << endl;
//возвращает 5
cout << s1.rfind("ss") << endl;
```

Функция rfind() аналогична, однако выполняет сканирование исходной строки справа налево. Она имеет следующие прототипы:

```
size_type
rfind (const string& str, size_type pos = npos) const;
size_type
rfind (const char* s, size_type pos, size_type n) const;
size_type
rfind(const char* s, size_type pos = npos) const;
size_type
rfind(char c, size_type pos = npos) const;
```

Функции find_first_of(), find_last_of(), find_first_not_of() и find_last_not_of() выполняют аналогичные операции поиска. Первый их аргумент задает строку как массив символов или символ для поиска. Другие (необязательные) параметры задают позицию и число символов исходной строки, участвующие в операции. Эти функции находят первый или последний символ, который присутствует или, наоборот, отсутствует в строке. В противном случае, они возвращает значение npos. Например:

```
//Найти первую гласную
i = s2.find_first_of("aeiou");
//Найти согласную
j = s2.find_first_not_of("aeiou", i);
```

Тема 19.8
Вставка символов в строку

Функция insert() дает возможность вставлять другую строку, массив символов или символ в исходную строку. Она имеет следующие прототипы:

```
string&
insert(size_type pos, const string& s);
string&
insert(size_type pos, const string& s, size_type pos2 = 0,
   size_type n = npos);
string&
insert(size_type pos, const char* s, size_type n);
string&
insert(size_type pos, const char* s);
string&
insert(size_type pos, size_type n, char c);
```

Первый вариант функции вставляет строку s в исходную строку, начиная с позиции pos. Второй вариант этой функции вставляет не все символы строки s, а лишь с позиции pos2 в этой строке. Количество вставляемых символов определяется меньшим из чисел n или s.size() – pos2. Если pos2 > s.size(), выбрасывается исключение out_of_range. Третий вариант функции вставляет в исходную строку n символов массива, на который указывает параметр s. Четвертый вариант функции вставляет в исходную строку символы массива, на который указывает параметр s, начиная с первого символа этого массива и до символа с

(но не включая его). Наконец, пятый вариант вставляет n копий символа c, начиная с позиции pos.

Все эти функции выбрасывают исключение out_of_range, если pos > size(), и исключение length_error, если полученная в результате вставки строка превышает максимально допустимый размер.

Тема 19.9
Замена и удаление символов из строки

Функция replace() заменяет символы в строке содержимым другой строки, массива символов или просто символом. Она имеет следующие прототипы:

```
string&
replace(size_type pos, size_type n1, const string& s);
string&
replace(size_type pos1, size_type n1, const string& str,
        size_type pos2, size_type n2);
string&
replace(size_type pos, size_type n1, const char* s, size_type n2);
string&
replace(size_type pos, size_type n1, const char* s);
string&
replace(size_type pos, size_type n1, size_type n2, char c);
```

Функция заменяет n1 символов исходной строки, начиная с позиции pos, всеми или выбранными символами строки или массива символов, заданных третьим параметром. В этом случае параметры n2 и pos2 задают, соответственно, число символов и позицию первого из символов, которые должны заменять символы исходной строки. Последняя форма функции предназначена для замены n1 символов исходной строки, начиная с позиции pos, n2 копиями символа c. Все эти функции выбрасывают исключение out_of_range, если pos1 > size(), и исключение length_error, если полученная в результате замены строка превышает максимально допустимый размер.

Функция erase () удаляет символы в строке:

```
string&
erase(size_type pos = 0, size_type n = npos);
```

Она удаляет символы, начиная с позиции pos. Количество удаляемых символов определяется меньшим из чисел n или size() – pos. Если pos > size(), выбрасывается исключение out_of_range.

Тема 19.10
Операции ввода-вывода строк

Для выполнения операций ввода-вывода в поток класс string перегружает операторы вставки и извлечения из потока, а также функцию-член getline() потока istream:

```
ostream&
operator<<(ostream& os, const string& str);
istream&
operator>>(istream& is, const string& str);
istream&
getline(istream& is, string& str, char delim);
```

Рассмотрим пример использования строк:

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
string str;
//Ввод строки длиной от 5 до 20 символов
while(str.empty() || str.size() <= 5)
{
  cout << "Введите строку длиной от 5 до 20 символов." << endl;
  cin >> str;
}
//Проверка доступа по индексу
cout <<"Изменить третий символ с '" << str[2] <<"' на '*' " <<endl;
str[2] = '*';
cout << "Теперь: " << endl << str
     << endl << endl;
//Проверка функции insert()
cout << "Найти средний символ: " << endl;
str.insert(str.size() / 2, "(середина здесь!)");
cout << str << endl << endl;
//Проверка функции replace()
cout << "Мне не нравится слово 'середина',"
"поэтому вместо него я скажу:" << endl;
str.replace(str.find("середина",0), 8, "центр");
cout << str << endl;
return 0;
}
```

При выполнении программа выводит на экран:

```
Введите строку длиной от 5 до 20 символов.
Программирование
Изменить третий символ с 'о' на '*'
Теперь: Пр*граммирование
```

```
Найти средний символ:
Пр*грамм(середина здесь!)ирование

Мне не нравится слово 'середина',
поэтому вместо него я скажу:
Пр*грамм(центр здесь!)ирование
```

Следует иметь в виду, что класс string содержит и много других, не упомянутых здесь функций.

Практикум
«Стандартный класс string»

Упражнение 19.1
Конструкторы строк. Изменение длины строк

В предлагаемом Вашему вниманию практическом задании необходимо сконструировать строку "Пример конструирования строки" всеми доступными конструкторами как статически, так и динамически.

Выведите на экран значение длины сконструированной строки. Увеличьте емкость строки на 50 символов, заполнив зарезервированное место символом "пробел". Очистите строку.

Упражнение 19.2
Работа с содержимым строк

Проанализируйте следующий пример. Пусть необходимо создать и вывести на печать динамическую строку, проинициализировав ее текстом "Hello!". Далее необходимо модифицировать эту строку таким образом, чтобы на экране была представлена фраза "Hello, world!".

Для решения поставленной задачи можно воспользоваться функциями-членами assign() и append().

```
int main()
{
   string *ps;
   ps = new string ("Hello!");
   cout << *ps << endl;
   ps->assign("Hello");
   string neo = ", world!";
   cout << ps->size() << endl;
```

```
    ps->append(neo, 0, 8);
    cout << *ps;
    return 0;
}
```

Модифицируйте приведенный пример таким образом, чтобы вместо использования функции-члена ps->assign() была применена функция доступа к отдельным символам ps->at(), заменяющая символ "!" исходной строки на ",", тогда как строка neo инициализировалась бы значением " world!".

Упражнение 19.3
Копирование строк. Подстроки

В данной части практического задания необходимо составить программу, копирующую в строку "Буратино! " строку "Это просто праздник какой-то!" с позиции 10, предварительно зарезервировав соответствующее количество памяти. Обоснуйте полученный результат.

Выведите на экран подстроку "праздник".

Упражнение 19.4
Операции поиска

Модифицируйте пример предыдущего задания таким образом, чтобы выводимая на экран подстрока "праздник" была найдена с помощью соответствующей функции-члена.

Дополните пример реверсивным поиском.

Упражнение 19.5
Вставка, замена и удаление символов строки

Проанализируйте приведенный ниже пример, демонстрирующий использование функции-члена insert() для вставки некоторой строки в исходную строку.

```
int main()
{
    string dance("Dance with SaragossaBand!");
    dance.insert(20, "-");
    cout << dance << endl;
```

```
    return 0;
}
```

Таким образом, на экран будет выведено:

```
Dance with Saragossa-Band!
```

Необходимо отметить, что операции вставки строк должна предварять проверка превышения максимально допустимого размера. Эту проверку Вам предлагается осуществить самостоятельно.

Модифицируем приведенный пример. Применив функцию replace() заменим подстроку "Saragossa" на "Electric Light Orchestra" и удалим символ "-" с помощью функции-члена erase().

```
int main()
{
   string dance("Dance with SaragossaBand!");
   dance.insert(20, "-");
   cout << dance << endl;
   dance.replace(11, 9, "Electric Light Orchestra ");
   dance.erase(36, 1);
   cout << dance << endl;
   return 0;
}
```

В результате на экран будет выведено:

```
Dance with Electric Light Orchestra Band!
```

Раздел 20
Шаблоны

Тема 20.1
Шаблоны функций

В этой главе мы познакомимся с двумя важными понятиями языка C++: шаблонами функций и шаблонами классов. *Шаблоны* позволяют давать обобщенные, в смысле произвольности используемых типов данных, определения функций и классов. Поэтому их часто называют *параметризованными функциями* и *параметризованными классами*. Часто также используются термины *шаблонные функции* и *шаблонные классы*.

Шаблон функции представляет собой обобщенное определение функции, из которого компилятор автоматически создает представитель функции для заданного пользователем типа (или типов) данных. Когда компилятор создает по шаблону функции конкретного ее представителя, то говорят, что он создал *порожденную функцию*. Синтаксис объявления шаблона функции имеет следующий вид:

```
template <class T1|T1 идент1,
class T2|T2 идент2,..., class Tn|Tn идентn>
возвр_тип имя_функции(список параметров)
{
 //тело функции
}
```

За ключевым словом template следуют один или несколько параметров, заключенных в угловые скобки, разделенные между собой запятыми. Каждый параметр является:

- либо ключевым словом class, за которым следует имя типа;
- либо именем типа, за которым следует идентификатор.

Для задания параметризованных типов данных вместо ключевого слова class может также использоваться ключевое слово typename. Параметры шаблона, следующие за ключевым словом

class или typename, называют *параметризованными типами*. Они информируют компилятор о том, что некоторый тип данных используется в шаблоне в качестве параметра. Параметры шаблона, состоящие из имени типа и следующего за ним идентификатора, информируют компилятор о том, что параметром шаблона является *константа* указанного типа. В связи с этим такие параметры называются *нетиповыми*.

Вы можете вызывать шаблонную функцию как обычную; никакого специального синтаксиса не требуется. Рассмотрим пример определения и вызова шаблонных функций:

```
#include <iostream>
using namespace std;
template <class T>
T Sqr(T x)
{
 return x*x;
}
template <class T>
T* Swap(T* t, int ind1, int ind2)
{
 T tmp = t[ind1];
 t[ind1] = t[ind2];
 t[ind2] = tmp;
 return t;
}
template <typename T1, typename T2>
void Display(T1 x, T2 y)
{
 cout.width(8);
 cout << x << ' ';
 cout.width(8);
 cout <<  y << endl;
}
template  <class T, int offset>
void GetAddres(T* obj, unsigned int *pAddr)
{
  *pAddr = (unsigned int)&obj[0] + offset*sizeof(T);
}
int main()
{
 int n = 10, sq_n, i = 2, j = 5;
 double d = 10.21, sq_d;
 char* str = "Шаблон";
 sq_n = Sqr(n);
```

```
    sq_d = Sqr(d);
    int Arr[100];
    unsigned int addr = 0;
    cout << "Значение n = " << n << endl
         << "Его квадрат = " << sq_n << endl;
    cout << "Значение d = " << d << endl
         << "Его квадрат = " << sq_d
    << endl;
    cout << "Исходная строка = '" << str << "'"
         << endl;
    cout << "Преобразованная строка = '"
         << Swap(str, i, j) << "'" << endl;
    cout << "Величина  Ее квадрат\n";
    Display (n, d);
    Display (sq_n, sq_d);
    GetAddres<int,5>(Arr, &addr);
    cout << "Адрес эл-та Arr[5] = " << hex
        << showbase << uppercase<<addr;
    return 0;
    }
```

При выполнении программа выводит на экран:

```
Значение n = 10
Его квадрат = 100
Значение d = 10.21
Его квадрат = 104.244
Исходная строка = 'Шаблон'
Преобразованная строка = 'Шанлоб'
Величина Ее квадрат
100
  10.21 104.244
Адрес эл-та Arr[5] = 0X12FDE8
```

Как и для обычных функций, Вы можете создать прототип шаблона функции в виде его предварительного объявления. Например:

```
template <class T>
T* Swap(T* t, int ind1, int ind2);
```

является предварительным объявлением шаблона функции Swap().

Имена параметров шаблонной функции в ее объявлении и определении могут не совпадать. Например,

```
//Объявление шаблонной функции
template <class T, class U>
void FuncName(T, U);
...
```

```
//Определение шаблонной функции
template <class S, class W>
void FuncName(S, W)
{
 //Тело шаблонной функции
}
```

Каждый типовый параметр шаблона должен появиться в списке формальных параметров функции. Например, следующее объявление приведет к ошибке во время компиляции:

```
template <class T, class U>
T FuncName(U);
```

Таким образом, при определении шаблонной функции нельзя вводить типовый параметр, нужный только для обозначения возвращаемого значения. С другой стороны, имя формального параметра в списке параметров шаблонной функции должно быть уникальным. Например, следующее объявление вызовет ошибку при компиляции:

```
template <class T, class T>
T FuncName(T,T);
```

Шаблонная функция может быть также объявлена внешней, статической или встраиваемой (inline). В этом случае соответствующий спецификатор располагается вслед за списком параметризованных типов шаблона, а не перед ключевым словом template:

```
//Объявление внешней шаблонной функции
template <class T> extern
T* Swap(T* t, int ind1, int ind2);
//Объявление статической шаблонной функции
template <class T> static
T* Swap(T* t, int ind1, int ind2);
//Объявление встроенной шаблонной функции
template <class T> inline
T* Swap(T* t, int ind1, int ind2);
```

Обратите внимание на вызовы шаблонных функций, сделанные в предыдущем примере, например:

```
sq_n = Sqr(n);
sq_d = Sqr(d);
```

Каждый такой вызов приводит к построению конкретной порожденной функции. В данном случае первый вызов приводит к построению компилятором функции Sqr() для целого типа дан-

ных, во втором – для типа double. В связи с этим процесс построения порожденной функции компилятором называют *конкретизацией* шаблонной функции.

Допускается явная спецификация типа шаблонных параметров, то есть вы можете потребовать от компилятора создать желаемую конкретизацию шаблонной функции. Например:

```
template <class T>
void FuncName(T) {...};
...
void AnotherFunc(char ch)
{
 //Требуем сгенерировать конкретизацию шаблонной функции
 FuncName<int>(ch);
}
```

Когда типовый шаблонный параметр задан явно, выполняется обычное неявное преобразование типов аргументов функции к типу соответствующих шаблонных параметров функции. В приведенном выше примере компилятор преобразует переменную ch типа char в тип int.

Тема 20.2
Перегрузка шаблонов функций

Как и обычные функции, шаблонные функции могут быть перегружены. Это означает, что в программе могут присутствовать несколько шаблонных функций с одним и тем же именем, если только они имеют различное число или типы параметров. Рассмотрим соответствующий пример:

```
#include <iostream>
using namespace std;
//Возвращает больший из двух параметров
template <class T>
const T& max(const T& a, const T& b)
{
 return a > b ? a : b;
}
//Возвращает наибольший элемент массива
template <class T>
const T max(T* a, size_t size)
{
 T* tmp = a;
 for (int i=0;i<size;i++)
```

```
  {
      if (a[i] > *tmp)
*tmp = a[i];
  }
return *tmp;
}

int main()
{
 int m = 9, n = 12;
 int arr[] = {3, 5, 7, 9};
 cout << "max int = " << max(m, n) << endl;
 cout << "max in arr = " << max(arr, sizeof(arr)/sizeof(int))
  << endl;
 return 0;
}
```

При выполнении программа выводит на экран:

```
max int = 12
max in arr = 9
```

Тема 20.3
Специализация шаблонов функций

Не всегда шаблонная функция может быть эффективно применена ко всем предусмотренным типам данных. В этом случае ничто не мешает Вам определить специализированный вариант этой функции для какого-то типа параметров. Например, рассмотренная выше шаблонная функция

```
template <class T>
const T& max(const T& a, const T& b)
```

непригодна для использования с массивами символов. В этом случае можно определить специализированную версию функции. Следующий пример это демонстрирует:

```
#include <iostream>
#include <cstring>

using namespace std;

//Возвращает больший из двух параметров
template <class T>
const T& max(const T& a, const T& b)
{
 return a > b ? a : b;
}
```

```
char* max(char* str1, char* str2)
{
 return strcmp(str1, str2) >= 0 ? str1 : str2;
}
int main()
{
 int m = 9, n = 12;
 char* str1 = "Слово и дело";
 char* str2 = "Слово и Дело";
 cout << "max int = " << max(m, n) << endl;
 cout << "max str = " << max(str1, str2) << endl;
 return 0;
}
```

Встретив вызов функции, компилятор использует следующий алгоритм для разрешения ссылки:

- найти обычную (нешаблонную) функцию, типы параметров которой в точности соответствуют указанным в вызове;
- если функция не найдена, найти *шаблонную* функцию, которая порождает функцию, типы параметров которой в точности соответствуют указанным в вызове;
- если такая шаблонная функция не найдена, снова начать поиск обычной функции, параметры которой можно преобразовать к заданным при вызове.

Заметим, что при рассмотрении шаблонных функций не рассматриваются порожденные функции со стороны возможного преобразования типов к указанным при вызове.

Тема 20.4
Шаблоны функций сортировки

Шаблоны функций представляют собой идеальное средство для реализации алгоритмов сортировки. Рассмотрим два примера таких функций. Самым простым (и самым неэффективным) является *алгоритм пузырьковой сортировки*. Этот алгоритм основан на перестановках соседних сортируемых элементов. Название алгоритма связано с его подобием всплывающим пузырькам воздуха в резервуаре с водой: каждый из переставляемых элементов, как и пузырек воздуха, достигает своего уровня, соответствующего порядку этого элемента в массиве.

```
#include <iostream>
#include <cstring>
```

```
using namespace std;
template <class T>
void bubbleSort(T* item, int count)
{
 register int i, j;
 T tmp;
 for (i=1; i<count; i++)
   for (j=count-1; j>=i; j--)
       {
        if (item[j-1] > item[j])
         {
          //Перестановка значений
          tmp = item[j-1];
          item[j-1] = item[j];
          item[j] = tmp;
         }
    }
}
int main()
{
 //Сортировка массива символов
 char str[] = "сортировка";
 bubbleSort(str, (int)strlen(str));
 //Сортировка целых чисел
 int IntArr[] = {3,7,4,9,11,1};
 int acount = sizeof(IntArr)/sizeof(int);
 bubbleSort(IntArr, acount);
 cout << "Отсортированная строка:" << str << endl;
 for (int i=0; i<acount; i++)
 cout << IntArr[i] << ' ';
 cout << endl;
 return 0;
}
```

Мы не имеем здесь возможности рассмотреть другие методы сортировки и приведем лишь самый быстрый, принадлежащий Хоару (C.A.R. Hoare). Он назвал этот алгоритм *методом быстрой сортировки*. Алгоритм основан на идее разбиения исходного массива элементов на части. На каждом шаге выбирается пограничное значение, которое называется компарандом. Это значение разбивает очередную часть массива на две новые части. Все элементы, меньшие компаранда, переносятся в одну часть, остальные – в другую. Затем этот же процесс повторяется для каждой из частей, пока не будет отсортирован весь массив. Ниже приведен модифицированный предыдущий пример, в котором сортировка осуществляется методом Хоара.

```
#include <iostream>
#include <string>

using namespace std;

template <class T>
void qSort(T* item, int left, int right)
{
 register int i, j;
 T x, y;
 i = left; j = right;
 x = item[(left+right)/2];
 do
 {
  while(item[i]<x && i<right) i++;
  while(x<item[j] && j>left) j--;
  if (i<=j)
    {
        y = item[i];
        item[i] = item[j];
        item[j] = y;
        i++; j--;
    }
 } while(i<=j);
 if (left<j) qSort(item, left, j);
 if (i<right) qSort(item, i,right);
}

//Входная функция быстрой сортировки
template <class T>
void quickSort(T* item, int count)
{
 qSort(item, 0, count-1);
}

int main()
{
 //Сортировка массива символов
 char str[] = "сортировка";
 quickSort(str, (int)strlen(str));
 //Сортировка целых чисел
 int IntArr[] = {3,7,4,9,11,1};
 int acount = sizeof(IntArr)/sizeof(int);
 quickSort(IntArr, acount);
 cout << "Отсортированная строка:" << str << endl;
 for (int i=0; i<acount; i++)
 cout << IntArr[i] << ' ';
 cout << endl;

 return 0;
}
```

Вы не должны путать приведенный здесь метод быстрой сортировки с тем, который использует достаточно старая функция библиотеки времени выполнения с именем qsort().

Тема 20.5
Шаблоны классов

Кроме шаблонов функций, можно объявлять и шаблоны классов. *Шаблон класса* представляет собой обобщенное определение класса, включающее типы данных в качестве параметров, из которого компилятор автоматически создает класс для заданных пользователем типов данных. Когда компилятор создает по шаблону класса конкретный класс для определенных пользователем типов данных, то говорят, что он создал *порожденный класс*. В связи с этим обстоятельством шаблон класса называют иногда *родовым классом*. Синтаксис объявления шаблона класса имеет следующий вид:

```
template <class T1|T1 идент1,
class T2|T2 идент2,
...
class Tn|Tn идентn>
class имя_класса
{
 // Тело класса
}
```

За ключевым словом template следуют один или несколько параметров, заключенных в угловые скобки и разделенных между собой запятыми. Каждый параметр является:

- либо ключевым словом class, за которым следует имя типа;
- либо именем типа, за которым следует идентификатор.

Затем следует объявление класса. Объявление и определение класса используют список параметров шаблона. Для задания параметризованных типов данных вместо ключевого слова class в угловых скобках может также использоваться ключевое слово typename. Как и в случае шаблонных функций, ключевое слово class или typename показывает, что параметр представляет или встроенный, или определенный пользователем тип данных. Параметры шаблона, следующие за ключевым словом class или typename, называют *параметризованными типами*. Параметры шаблона, состоящие из имени типа и следующего за ним идентификатора, информируют

компилятор о том, что параметром шаблона является *константа* указанного типа. В связи с этим такие параметры называются *нетиповыми*. Имя формального параметра в списке параметров шаблона класса должно быть уникальным.

Функции-члены шаблона класса могут определяться в теле класса (то есть как встроенные) или вне тела класса. В последнем случае ее определение имеет вид:

```
template <class T1|T1 идент1,
class T2|T2 идент2,
...
class Tn|Tn идентn>
возвр_тип
имя_класса < T1|идент1,T2|идент2,
...
Tn|идентn>
::имя_функции (список параметров)
{
 // Тело функции
}
```

Таким образом, список параметризованных типов и нетиповых констант указывается дважды. Первый раз после ключевого слова template он указывается в точности так, как при объявлении шаблона. Второй раз – после имени класса – он указывается снова в угловых скобках, но на этот раз перечисляются только сами параметризованные типы и нетиповые константы.

В остальном определение функции-члена не отличается от обычного. Если необходимо объявить функцию-член статической или встроенной, то модификатор указывается вслед за списком параметризованных типов и нетиповых констант перед возвращаемым типом функции. Например:

```
template <class T>
inline void className<T>::funcName(int i)
{
 //Тело функции
}
```

Рассмотрим примеры определения шаблонов классов:

```
template <int size>
class TmplClass
{
 char* s;
public:
  TmplClass()  {s = new char[size];}
```

```
    ~TmplClass() {delete[] s;}
    operator char*() {return s;}
};
template <class T, int nElem>
class MyClass
{
 T* p;
 public:
   MyClass();
   ~MyClass();
   operator T*();
};
template <class T, int nElem>
MyClass<T, nElem>::MyClass()
{
  p = new T[nElem];
}
template <class T, int nElem>
MyClass<T, nElem>::~MyClass()
{
  delete[] p;
}
template <class T, int nElem>
MyClass<T, nElem>::operator T*()
{
  return p;
}
```

Нельзя определить внутренний шаблонный класс в теле другого шаблонного класса, однако ничто не мешает Вам использовать ранее объявленный шаблонный класс при объявлении и определении другого. Рассмотрим в качестве примера определение класса для обобщенного связного списка, предназначенного для работы с определяемым пользователем типом данных:

```
//Определение шаблонного класса узла списка
template <class T>
class Node
{
   Node<T>* Prev;
   Node<T>* Next;
   T*       Item;
 public:
   Node(Node<T>*, Node<T>*, T*);
   ~Node();
};
template <class T>
```

```
Node<T>::Node(Node<T>* prev,
  Node<T>* next, T* item)
{
 Prev = prev;
 Next = next;
 Item = item;
}

template <class T>
Node<T>::~Node()
{
 delete Item;
}

//Определение шаблонного класса связного списка,
//использующее предыдущий шаблонный класс
template <class T>
class List
{
   Node<T>* Head;
   Node<T>* Item;
   Node<T>* Tail;
 public:
   List();
   ~List();
   void Add(T*);
   void Remove();
};
template <class T>
List<T>::List()
{
 Head = Item = Tail = 0;
}
template <class T>
List<T>::~List()
{
 //Удаление всех элементов списка
 Item = Head;
 while(Item)
 {
  Remove();
  Item++;
 }
}
template <class T>
void List<T>::Add(T* item)
{
 if (!Head)
```

```
  {
   //Создание первого элемента
   Head = Tail =
   Item = new Node<T>(0, 0, item);
  }
 else
  {
   //Создание очередного узла и сцепление его со списком
    ...
  }
}
template <class T>
void List<T>::Remove()
{
 //Отсоединение узла от списка
   ...
 //удаление текущего узла
 delete Item;
}
```

Шаблонные классы можно объявлять дружественными другому классу. В результате любая функция, сгенерированная для такого шаблонного класса, тоже будет дружественной другому классу. Рассмотрим пример:

```
template <class T>
class List
{
  Node<T>* Head;
  Node<T>* Item;
  Node<T>* Tail;
 public:
  List();
  ~List();
  void Add(T*);
  void Remove();
 friend class FClass<S>;
};
template <class S>
class FClass
{
 //Объявление дружественного шаблонного класса
   ...
};
```

Теперь любой порожденный класс типа FClass<S> будет дружественным для класса типа List<T>.

Тема 20.6
Конкретизация шаблона класса

Процесс построения порожденного класса компилятором называют *конкретизацией* шаблонного класса. Шаблонный класс конкретизируется присоединением к его имени полного списка фактических параметров, заключенных в угловые скобки. При этом компилятор создает порожденный класс, в котором:

- каждый параметр типа class T_i заменяется именем действительного типа;
- каждый параметр типа $Type_i$ $идент_i$ заменяется константным выражением.

Фактические аргументы для нетиповых параметров должны вычисляться на стадии компиляции, так как именно на этой стадии производится конкретизация шаблонных классов.

Тема 20.7
Специализация шаблонов классов

Можно специализировать шаблонный класс для некоторых значений параметризованных типов, предусмотрев для этих случаев отдельную реализацию некоторых функций-членов. Рассмотрим пример:

```
#include <iostream>
#define SIZE 10
using namespace std;
template <typename T>
class Stack
{
  T stack[SIZE];
  int top;
 public:
  Stack(){top = 0;}
  void push(T item);
  T pop();
};
//Помещение элемента в стек
template <typename T>
void Stack<T>::push(T item)
{
 if (top >= SIZE)
   {
     cout << "\nСтек полон\n";
```

```
      return;
    }
  stack[top] = item;
  top++;
}
//Выталкивание элемента из стека
template <typename T>
T Stack<T>::pop()
{
  if (top <= 0)
    {
      cout << "\nСтек пуст\n";
      return 0;
    }
  top--;
  return stack[top];
}
//Специализация функции-члена
//для T = char
void Stack<char>::push(char item)
{
  if (top >= SIZE-1)
    {
      cout << "\nСтек полон\n";
      return;
    }
  stack[top] = item;
  top++;
}
int main()
{
  Stack<int> st1;
  Stack<char> st2;
  int i;
     //Помещение элементов в стеки
     for (i=0;i<8;i++)
     {
      st1.push(i+1);
      st2.push(char('A' + i));
     }
  cout << "Из стека st1 извлечены:\n";
  for (i=0;i<8;i++)
    cout << st1.pop() << ' ';
  cout << endl;
  cout << "Из стека st2 извлечены:\n";
  for (i=0;i<8;i++)
    cout << st2.pop() << ' ';
  return 0;
}
```

В рассматриваемом примере создаются два стека: стек целых чисел и стек символов. Для стека символов функция-член push() класса Stack специализируется. При выполнении программа выводит на экран:

```
Из стека st1 извлечены:
8 7 6 5 4 3 2 1
Из стека st2 извлечены:
H G F E D C B A
```

Тема 20.8
Статические члены шаблонного класса

В шаблонном классе можно объявлять статические данные-члены. В этом случае для каждой конкретизации шаблонного класса будет построена своя совокупность статических данных-членов. Рассмотрим пример:

```
template <class T>
class Queue
{
  //закрытые члены класса
  static int* p;
  public:
  Queue();
  static Queue* Free();
};
//Инициализация статического данного-члена
template <class T>
int Queue<T>::p = 10;
```

Можно определить и специализацию статического данного-члена:

```
int Queue<char*>::p = 1024;
```

Для статических данных-членов обычного класса память выделяется сразу после определения класса. Для статических данных-членов шаблонного класса память выделяется при каждой конкретизации класса. Доступ к статическому члену шаблонного класса осуществляется через конкретизацию этого класса:

```
int n = Queue<string>::p;
int m = Queue<char*>::p;
```

Тема 20.9
Ключевое слово typename

Об использовании ключевого слова typename вместо ключевого слова class в объявлении шаблона функции или класса мы уже упоминали.

Заметим здесь, что можно смешивать эти ключевые слова в объявлении параметризованных типов шаблона. Например:

```
template <typeneme T, class S>
bool IsEmpty();
```

С другой стороны, ключевое слово typename может использоваться для ссылки на тип данных, который еще не определен. Например:

```
template <class T>
void func()
{
 typedef typename T::A TA;
 TA ta1;
  typename T::A ta2;
 TA* pta2;
}
```

Здесь ключевое слово typename используется для объявления переменных ta1, ta2 и pta2 типа T::A, который еще не объявлен.

Тема 20.10
Недостатки шаблонов

Шаблоны предоставляют определенные выгоды при программировании, связанные с широкой применимостью кода и легким его сопровождением. В общем, этот механизм позволяет решать те же задачи, для которых используется полиморфизм. С другой стороны, в отличие от макросов, они позволяют обеспечить безопасное использование типов данных. Однако с их использованием связаны и некоторые недостатки:

- программа содержит полный код для всех порожденных представителей шаблонного класса или функции;
- не для всех типов данных предусмотренная реализация класса или функции оптимальна.

Преодоление второго недостатка возможно с помощью специализации шаблона.

Практикум «Шаблоны»

Упражнение 20.1 Шаблоны функций

В предлагаемом практическом задании Вам необходимо составить шаблонную функцию, суммирующую передаваемые ей аргументы и возвращающую целое значение полученной суммы.

Создайте специализированный вариант полученной функции, в котором первый аргумент будет целочисленным (int), а второй – вещественным (float).

Дополните полученный листинг перегруженными шаблонными функциями для трех, четырех и пяти аргументов.

Упражнение 20.2 Шаблоны классов

Проанализируйте пример создания шаблонных классов.

Пусть необходимо создать набор классов для манипуляции информацией, хранящейся в системном реестре операционной системы. Ниже представлено перечисление типов данных, хранящихся в реестре.

```
enum TYPE {
   REG_NONE,
   REG_SZ,
   REG_EXPANDREG_SZ,
   REG_BINARY,
   REG_DWORD,
   REG_DWORD_LITTLE_ENDIAN = 4,
   REG_DWORD_BIG_ENDIAN,
   REG_LINK,
   REG_MULTI_SZ,
   REG_RESOURCE_LIST,
   REG_FULL_RESOURCE_DESCRIPTOR,
   REG_RESOURCE_REQUIREMENT_LIST
};
```

Поскольку параметры, содержащиеся в реестре, разнотипные, целесообразно воспользоваться шаблоном класса:

```
template <class T>
class Param
{
```

```
    // Указатель на предыдущий параметр
    Param* prev;
    // Указатель на текущий параметр
    Param* item;
    // Указатель на следующий параметр
    Param* next;
protected:
    // Имя параметра
    string name;
    // Тип параметра
    TYPE type;
    // Значение параметра
    T* value;
public:
    Param(){prev=item=next=NULL;name="";}
    ~Param(){}
    // Интерфейсные функции
    void SetName(string nm){name=nm;}
    string GetName(){return name;}
    void SetType(TYPE t){type=t;}
    TYPE GetType(){return type;}
    Param* GetItem(){return item;}
    Param* GetNext(){return next;}
    Param* GetPrev(){return prev;}
    void SetNext(Param* par){
      if(par) next=new Param;
      next=par;
    }
    //Аналогично определяются функции void SetItem(Param* par)
    //и void SetPrev(Param* par)
     ...
};
```

Теперь разработаем класс для работы с веткой реестра. Как и в случае использования параметров, предоставим ветвям реестра возможность получать информацию о соседних элементах: предыдущей и следующей ветви. Кроме этого, снабдим создаваемый класс возможностью хранить информацию о ветке-родителе и указатель на ее первый параметр:

```
template <class P, class B>
class Branch
{
    // Название ветви реестра
    string name;
    // Указатель на первый параметр
    P* first_param;
    // Указатель на родительскую ветвь
    B* parent;
    // Указатель на предыдущую ветвь
```

```
    B* prev;
    // Указатель на следующую ветвь
    B* next;
    // Счетчик параметров в данной ветви
    unsigned int count;
  public:
    Branch(){
      parent=prev=next=NULL;
      count=0;
      first_param=NULL;
    }
    ~Branch(){}
    // Интерфейсные функции
    B* GetParent(){return parent;}
    B* GetPrev(){return prev;}
    B* GetNext(){return next;}
    P* GetParam(){return first_param;}
    string GetName(){return name;}
    void SetParent(B* par){parent=par;}
    void SetPrev(B* prv){prev=prv;}
    void SetNext(B* nxt){next=nxt;}
    void SetName(string nm){name=nm;}
    void AddParam(P* par){
      if(!first_param){
        first_param=par;
        count++;
        return;
      }
      unsigned int i=count;
      P* tpar = first_param->GetNext();
      while(i-- && tpar){tpar = tpar->GetNext();}
      if(tpar) tpar->SetNext(par);
    }
};
int main()
{
    Param<TYPE> par;
    par.SetName("First");
    par.SetType(REG_DWORD);
    Branch<Param<TYPE>,string> brRoot, brChild;
    brRoot.SetName("Root Branch");
    brChild.SetName("Child Branch");
    brChild.AddParam(&par);
    cout << "Branch: " << brRoot.GetName() << " Param: "
         << (brRoot.GetParam()) << endl;
    cout << "Branch: " << brChild.GetName() << " Param: "
         << (brChild.GetParam())->GetName() << endl;
    return 0;
}
```

Раздел 21
Стандартная библиотека шаблонов (STL)

Тема 21.1
Назначение и состав библиотеки

Стандартная библиотека шаблонов (STL – Standard Template Library) – это библиотека контейнерных классов, которая включает векторы, списки, очереди и стеки, а также ряд алгоритмов общего назначения. Цель включения библиотеки STL в стандарт языка состоит в том, чтобы избавить Вас от разработки рутинных общепринятых программ. Она была разработана сотрудниками Hewlett-Packard А.А.Степановым и М.Ли. После внесения незначительных поправок Комитет по стандартизации языка С++ принял решение о включении STL в стандарт языка С++. Библиотека стандартных шаблонов содержит более сотни различных шаблонов и алгоритмов. Прежде чем приступить к ее изучению, приведем краткий обзор этой библиотеки.

Ядро библиотеки составляют три группы шаблонных классов: контейнеры, алгоритмы и итераторы. Кроме этих групп стандартных компонентов, STL поддерживает несколько компонентов, среди которых распределители памяти, предикаты и функции сравнения.

Предикат – это просто функция, которая возвращает либо булевское, либо целое значение. Следуя обычному соглашению, целое значение рассматривается как эквивалент булевского значения true, если оно отлично от нуля. В противном случае – оно рассматривается как false. Предикаты используются в универсальных алгоритмах в качестве параметров. По числу передаваемых им аргументов предикаты делятся на *унарные* (один аргумент) и *бинарные* (два аргумента). Специальным типом бинарного предиката, предназначенным для сравнения значений двух

аргументов, являются *функции сравнения*. Функции сравнения возвращают true, если первый аргумент меньше второго.

Тема 21.2
Контейнеры

Контейнер – это объект, который предназначен для хранения других объектов. STL предоставляет два вида контейнеров: последовательные и ассоциативные. *Последовательные контейнеры* предназначены для обеспечения последовательного или произвольного доступа к своим членам (или элементам). *Ассоциативные контейнеры* получают доступ к своим элементам по ключу. Все контейнерные классы библиотеки STL определены в пространстве имен std.

Тема 21.3
Последовательные контейнеры

Последовательный контейнер (иногда говорят – последовательность) – это вид контейнеров, в котором введено отношение порядка для совокупности хранимых объектов. Это означает, что для элементов контейнера определены понятия первый, последний, предыдущий и следующий. К этому виду контейнеров принадлежат векторы, списки, деки и строки типа string.

Тема 21.4
Векторы

Вектор – это контейнерный класс, в котором доступ к его элементам осуществляется по индексу. В силу этого векторы во многом напоминают одномерные массивы. Библиотека STL предоставляет Вам контейнерный класс vector, определенный в заголовочном файле <vector> и доступный в пространстве имен std. Класс vector ведет себя подобно массиву, однако предоставляет больше возможностей по управлению ним. В частности, вектор можно наращивать по мере необходимости, и он обеспечивает контроль значений индексов. Определение класса vector выглядит следующим образом:

```
template <class T, class A = allocator<T>>
class vector
```

```
{
 // члены класса
}
```

Первый аргумент (class T) определяет тип элементов, хранимых в векторе. Второй аргумент (class A) определяет тип класса, отвечающего за распределение памяти для элементов вектора. По умолчанию память для элементов вектора распределяется и освобождается глобальными операторами new() и delete(). Таким образом, для создания нового элемента вектора вызывается конструктор класса T. Для встроенных типов данных векторы можно определить следующим образом:

```
vector<int> vInts;  //Вектор целых чисел
vector<double> vDbls;  //Вектор чисел типа double
```

Обычно пользователь знает, сколько элементов будет содержаться в векторе. В этом случае он может зарезервировать память в векторе для нужного числа элементов, указав его после имени объекта в круглых скобках, например:

```
vector<int> Ints(50);
```

Количество элементов в векторе можно узнать с помощью функции-члена size():

```
size_type size() const;
```

Функция resize изменяет величину вектора. Она имеет следующий прототип:

```
void resize(size_type sz);
```

Если новая величина вектора sz больше текущей, то в конец вектора добавляется нужное число элементов класса T. Если же новая величина вектора меньше текущей, вектор усекается с конца.

Функция

```
void pop_back();
```

удаляет последний элемент вектора.

Обращаться к элементам вектора можно по индексам. Значения индекса начинаются с нуля. Например, присвоить значение четвертому элементу объявленного выше вектора можно следующим образом:

```
Ints[3] = 15;
```

Если первоначально зарезервированная память для элементов вектора исчерпана, число элементов вектора будет автоматически увеличено. Добавить элемент в конец вектора можно с помощью

функции-члена push_back(), однако она требует, чтобы в классе хранимых в контейнере объектов был определен конструктор копирования (эта функция добавляет в контейнер копию элемента). Она имеет следующий прототип:

```
void push_back(const T& x);
```

Проверить, не является ли контейнер пустым, можно с помощью функции-члена empty():

```
bool  empty() const;
```

Она принимает значение true, если контейнер пуст.

Функция clear(), имеющая следующий прототип:

```
void clear();
```

удаляет все элементы из вектора.

Функция front() возвращает ссылку на первый элемент в списке, а функция back() – на последний. Функция at() работает подобно оператору индексирования ([]), но является более безопасной, поскольку проверяет, попадает ли переданный ей индекс в диапазон допустимых значений. Если индекс вне диапазона, она генерирует исключение out_of_range. Функция insert() вставляет один или несколько элементов, начиная с указанной позиции в векторе. Функция pop_back() удаляет из вектора последний элемент. Кроме того, для класса vector определены обычные операции сравнения.

Рассмотрим пример использования вектора в качестве контейнера:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
 //Создаем вектор нулевой длины
 vector<int> v;
 int i;
 //Помещение значений в вектор
 for (i=0;i<5;i++)
   v.push_back(i);
 //Вывод на экран текущего размера вектора v
 cout << "Размер вектора v =" << v.size() << endl;
 //Вывод на экран содержимого вектора
 cout << "Содержимое вектора:\n";
 for (i=0;i<v.size();i++)
 cout << v[i] << endl;
```

```
    return 0;
}
```

В примере создается вектор целых чисел путем помещения в него значений параметра цикла for. Затем на терминал выводится величина вектора и его содержимое.

Рассмотрим теперь функции-члены шаблонного класса vector. Функция

```
size_type capacity() const;
```

возвращает величину распределенной для вектора памяти в виде числа элементов, которые могут быть сохранены в векторе.

Функция

```
void reserve(size_type n);
```

добавляет емкость вектору в предположении, что вектор будет дополняться новыми элементами. После вызова этой функции последующие операции добавления элементов не будут вызывать перераспределения памяти, пока величина вектора не превзойдет n. Перераспределение памяти не осуществляется, если n не превосходит значения capacity(). Если происходит перераспределение памяти, то все итераторы и ссылки, указывающие на элементы вектора, становятся неверными.

Функция

```
reference back();
```

возвращает ссылку на последний элемент, а функция

```
reference front();
```

на первый элемент вектора. Наконец, функция

```
reference at(size_type n);
```

возвращает ссылку на элемент со значением индекса n.

Кроме того, в классе vector определены итераторы, которые играют роль указателей на элементы контейнера и позволяют осуществлять навигацию по этим элементам. Чтобы получить элемент, на который указывает итератор, его нужно разыменовать. К итератору можно применять операции инкремента и декремента, которые приводят к тому, что итератор позволяет осуществлять доступ, соответственно, к следующему и предыдущему элементам вектора. Типом итераторов является тип iterator. Следующая функция

```
iterator begin();
```

возвращает итератор произвольного доступа, который указывает на первый элемент, а функция

```
iterator end();
```

на область памяти за последним элементом. Функция

```
iterator insert(iterator position, const T& x);
```

вставляет элемент x перед элементом, позиция которого задана параметром position. Возвращаемое значение указывает на вставленный элемент.

Кроме обычных итераторов, в контейнере определены *обратные (или реверсивные) итераторы*, которые имеют тип reverse_iterator. Такие итераторы используются для прохода последовательного контейнера в обратном направлении, то есть от последнего элемента к первому. Инкремент обратного итератора приводит к тому, что он начинает указывать на *предыдущий* элемент, а декремент – к тому, что он указывает на *следующий* элемент.

Функция

```
reverse_iterator rbegin();
```

возвращает обратный итератор произвольного доступа, который указывает на область памяти за последним элементом. Функция

```
reverse_iterator rend();
```

возвращает обратный итератор произвольного доступа, который указывает на первый элемент.

Рассмотрим пример работы с векторным контейнером с помощью итераторов.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
 //Создаем вектор нулевой длины
 vector<int> v;
 int i;
 //Помещение значений в вектор
 for (i=0;i<5;i++)
   v.push_back(i);
 //Вывод на экран текущего размера вектора v
 cout << "Размер вектора v = " << v.size() << endl;
 //Вывод на экран содержимого вектора
 cout << "Содержимое вектора:\n";
 //Доступ к вектору с помощью итератора
```

```
vector<int>::iterator p = v.begin();
while(p!=v.end())
{
 cout << *p << endl;
 p++;
}
return 0;
}
```

Этот пример функционально полностью аналогичен предыдущему.

Тема 21.5
Списки

Список – это контейнер, предназначенный для обеспечения оптимального выполнения вставок и удалений объектов. Библиотека STL содержит контейнерный класс list, предоставляющий в ваше распоряжение двусвязный (двунаправленный) список. Он определен в заголовочном файле <list> в пространстве имен std. Этот класс имеет такие же функции, как вектор и ряд дополнительных функций. Если с векторами итераторы используются редко, то при работе со списком они являются основным средством. Для создания списков предназначены конструкторы, которые имеют несколько форм.

Конструктор

```
explicit list(const Allocator& alloc = Allocator())
```

создает список, не содержащий элементов, причем список использует для распределения памяти объект alloc. Следующий конструктор

```
explicit list(size_type n);
```

создает список длины n, содержащий копии значения по умолчанию объектов типа T. Тип T должен иметь конструктор по умолчанию. Список использует для распределения памяти объект alloc. Наконец, конструктор

```
list(size_type n, const T& value, const Allocator& alloc = Allocator());
```

создает список длины n, содержащий копии значения value объектов типа T. Тип T должен иметь конструктор по умолчанию. Список использует для распределения памяти объект alloc.

Кроме того, в классе list определены дополнительные функции.

Функции

```
void push_front(const T& x);
```

и

```
void pop_front();
```

аналогичны функциям push_back() и pop_back(), однако осуществляют добавление и удаление элементов в начале списка. Функция

```
void remove(const T& value);
```

удаляет элемент из списка. Вызвав функцию sort(), можно отсортировать список:

```
void sort();
```

Эта функция сортирует список в соответствии с отношением порядка, задаваемым операторной функцией operator<(). Например, в следующей программе создается список случайных символов, а затем эти символы сортируются:

```
#include <iostream>
#include <list>
#include <cstdlib>

using namespace std;
typedef list<char> CList;

int main()
{
 CList clist;
 //Заполнение списка
   for (int i=0;i<10;i++)
  clist.push_back('F' + (rand() % 26));
 cout << "Исходный список: ";
 CList::iterator iter = clist.begin();
 while(iter != clist.end())
 {
   cout << *iter;
  iter++;
 }
 cout << endl;
clist.sort(); //Сортируем список
 cout << "Отсортированный список: ";
 iter = clist.begin();
 while(iter != clist.end())
 {
  cout << *iter;
  iter++;
 }
 cout << endl;
 return 0;
}
```

При выполнении программа выводит на экран:

```
Исходный список: FP^NYYYVJN
Отсортированный список: FJNNPVYYY^
```

Тема 21.6
Деки

Название абстрактной структуры данных «дек» происходит от сокращения английских слов double-ended queue – двусторонняя очередь. Таким образом, дек – это абстрактная структура данных, которая позволяет осуществлять вставку и удаление из начала или конца совокупности элементов. Иначе говоря, дек – это тип последовательного контейнера, который поддерживает итераторы произвольного доступа к элементам. Контейнерный класс дек позволяет выполнять и многое другое. На самом деле, структуры данных «дек» почти объединяют в себе возможности, предоставляемые векторами и списками. Как и вектор, дек является индексированной совокупностью, то есть доступ к значениям возможен по индексу. Однако, как и список, он позволяет легко добавлять элементы в начало или конец совокупности. Эта возможность обеспечивается только частично в векторах. Как и списки, и векторы, дек поддерживает вставку элементов в середину хранимой совокупности. Такие операции в нем не столь эффективны, как в списке, однако все же более эффективны, чем в векторе. Класс deque определен в заголовочном файле <deque> в пространстве имен std.

Тема 21.7
Операции с деками

Функции-члены begin() и end() возвращают итераторы произвольного доступа, которые позволяют осуществлять доступ к *произвольным* элементам, а не двунаправленные итераторы, как в случае списка. Дек поддерживает операции вставки и удаления элементов в начало и конец контейнера, а также вставку в середину контейнера, которая занимает большее время. Вставка элементов выполняется функциями insert(), push_front() или push_back() и может потенциально приводить к неправильным значениям всех действующих итераторов и ссылок на элементы в деке. Поскольку деки обеспечивают итераторы, к ним могут применяться все описываемые ниже универсальные алгоритмы. Вектор хранит элементы в одном большом блоке памяти. С другой сто-

роны, дек использует большое число меньших блоков памяти. Это может оказаться важным при выборе типа контейнера на системах с ограниченными ресурсами.

Тема 21.8
Объявление и инициализация дека

Объявление дек осуществляется аналогично объявлению вектора.

По умолчанию STL предоставляет в качестве класса, отвечающего за распределения памяти, класс Allocator, который реализует этот интерфейс с помощью стандартных операторов new() и delete(). Любой тип данных, используемый в качестве параметра шаблона T, должен предоставлять конструктор копирования, деструктор, оператор адреса и оператор присваивания.

Рассмотрим конструкторы класса дек. Конструктор

```
explicit deque(const Allocator& alloc = Allocator());
```

является конструктором по умолчанию. Он создает дек из 0 элементов. Управление распределением памяти для элементов берет на себя объект, представленный параметром шаблона alloc. Дек будет использовать для управления распределением памяти объект Allocator(). Другой конструктор –

```
explicit deque(size_type n);
```

создает список длины n, содержащий копии значений по умолчанию для типа T. Тип T должен иметь конструктор по умолчанию. Для управления памятью используется Allocator(). Конструктор

```
deque(size_type n, const T& value,
const Allocator& alloc = Allocator());
```

создает список, содержащий n копий объектов класса T, и использует для управления памятью объект alloc. Следующий конструктор

```
template <class InputIterator>
deque(InputIterator first, InputIterator last,
      const Allocator& alloc = Allocator());
```

создает дек длины last - first, заполненный значениями, полученными разыменованием итераторов в диапазоне [first, last]. Он использует для управления памятью объект alloc.

В деке определены также итераторы:

```
iterator begin();
const_iterator begin() const;
```

возвращает (постоянный) итератор произвольного доступа, который указывает на первый элемент дека.

```
iterator end();
const_iterator end() const;
```

возвращает (постоянный) итератор произвольного доступа, который указывает на область памяти за последним элементом дека.

Кроме обычных итераторов, в деке определены обратные (или реверсивные) итераторы:

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

возвращает (постоянный) обратный итератор произвольного доступа, который указывает на область памяти за последним элементом дека.

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

возвращает (постоянный) обратный итератор произвольного доступа, который указывает на первый элемент дека.

В деке перегружены также ряд операторов, среди них – оператор присваивания:

```
deque<T, Allocator>&
operator=(const deque<T, Allocator>& x);
```

Он удаляет все элементы в деке, затем вставляет копию каждого элемента из дека x. Операторы ссылки:

```
reference operator[](size_type n);
const_reference
operator[](size_type n) const;
```

возвращают (постоянную) ссылку на элемент со значением индекса n. Класс deque содержит также значительное количество открытых функций-членов, которые во многом аналогичны членам класса vector. Здесь мы отметим лишь функции, предназначенные для операций помещения и извлечения в начало дека:

```
void push_front(const T& x);
void pop_front();
```

и в конец дека

```
void push_back(const T& x);
void pop_back();
```

Заметим, что функции извлечения приводят к удалению элемента из дека.

Рассмотрим пример использования дека:

```
#include <deque>
#include <iostream.h>
using namespace std;
int main()
{
 deque<int> dq;
 int x;
 cout <<"Введите положительные целые, "
<<"закончив ввод нулем:\n";
 while(cin>>x, x!=0)
 {
  if (x%2 == 0)
    dq.push_back(x);
  else
    dq.push_front(x);
 }
 deque<int>::iterator i;
 cout << "deque содержит:\n";
 for (i=dq.begin();i!=dq.end(); i++)
    cout <<*i << ' ';
 cout <<endl;
 return 0;
}
```

В этом примере пользователю предлагается ввести последовательность целых чисел, завершив ее нулем. Вводимые числа размещаются в деке. Если число четно, оно помещается в конец дека, а если нечетно, – в начало. Затем содержимое дека выводится на экран. Во время выполнения программа выводит на экран:

```
Введите положительные целые, закончив ввод нулем:
1
2
3
4
5
6
0
deque содержит:
5 3 1 2 4 6
```

Тема 21.9
Стеки

Стек является одной из самых распространенных в программировании структур данных. Данные в стеке организованы по принципу «последним вошел – первым вышел» (LIFO – Last In –

First Out). Это означает, что добавлять и удалять элементы стека можно только с одного конца. Доступный конец стека называется его *вершиной*, операция добавления элемента в стек называется *помещением* в стек (по англ. push()), а операция извлечения элемента из стека – *выталкиванием* (по англ. pop()).

Абстрактный тип данных стек традиционно определяется как объект, который реализует перечисленные ниже операции (см. табл. 21.1):

Таблица 21.1

Функции типа данных стек и их назначение

Функция	Операция
empty()	Возвращает true, если стек пуст
size()	Возвращает число элементов стека
top()	Возвращает (но не удаляет) элемент в вершине стека
push(newElement)	Помещает новый элемент в стек
pop()	Удаляет (но не возвращает) элемент из вершины стека

Заметим, что доступ к вершине стека и удаление элемента из вершины стека реализуются как отдельные операции. В библиотеке STL определен шаблонный класс stack, реализующий возможности стека. Для использования стандартного стека нужно подключить к программе заголовочный файл <stack>. В отличие от программного стека, абстрактный тип данных стек предназначен для хранения элементов одного типа. Это же ограничение относится и к стандартному классу stack.

Ввиду того, что класс stack не реализует итератор, универсальные алгоритмы, описываемые ниже, не могут применяться к стандартным стекам.

Тема 21.10
Объявление и инициализация стека

Стек не является независимым контейнерным классом. По терминологии STL он представляет собой *контейнерный адаптер*, который строится поверх других контейнеров, которые в действительности хранят элементы. Класс stack лишь позволяет

контейнеру вести себя как стек. В связи с этим объявление стека должно задавать тип элементов, которые будут помещаться и извлекаться из него, а также тип контейнера, который будет использоваться для хранения элементов. Контейнером по умолчанию для стека является дек, однако можно использовать также и список или вектор. Векторная версия, вообще говоря, меньше, тогда как версия, использующая дек в качестве контейнера, работает несколько быстрее. Для элементов, хранимых в очереди, должны быть определены операции меньше (<) и равно (==). Ниже приведены простые объявления стека:

```
stack<int> stackOne;
// стек с использованием дека
stack<double, deque<double> > stackTwo;
stack<Part *, list<Part * > > stackThree;
stack<Customer, list<Customer> > stackFour;
```

В последнем примере стек создается из объектов объявленного пользователем типа Customer.

Замечание. Для большинства компиляторов важно оставлять пробел между двумя правыми угловыми скобками в объявлении стека, чтобы они не интерпретировались как оператор сдвига вправо.

Тема 21.11
Очереди

Очередь также относится к одному из самых распространенных типов абстрактных структур данных. Данные очереди организованы по принципу «первым вошел – первым вышел» (FIFO – First In – First Out). Это означает, что элементы помещаются в очередь с одного конца (который называется ее *хвостом*), а извлекаются – из другого (который называется ее *началом*). STL предоставляет в ваше распоряжение двустороннюю очередь – класс queue. Подобно стеку, он представляет собой *контейнерный адаптер*, который позволяет контейнеру вести себя как очередь. Класс queue адаптирует любой контейнер, который поддерживает операции front(), back(), push_back() и pop_front(). В частности, могут использоваться дек и список. Контейнером по умолчанию для очереди является дек. Для элементов, хранимых в очереди, должны быть определены операции меньше (<) и равно (==).

Как и любая абстрактная структура данных типа очередь, класс queue поддерживает операции (см. табл. 21.2):

Таблица 21.2

Стандартные функции класса queue и их назначение

Функция	Операция
empty()	Возвращает true, если очередь пуста
size()	Возвращает число элементов очереди
front()	Возвращает (но не удаляет) элемент из начала очереди
back()	Возвращает (но не удаляет) элемент из конца очереди push(newElement). Помещает новый элемент в конец очереди
pop()	Удаляет (но не возвращает) элемент из начала очереди

Заметим, что операции доступа к элементам и удаления их из очереди реализованы отдельно. Для использования стандартной очереди нужно подключить к программе заголовочный файл <queue>.

Ввиду того, что класс queue не реализует итератор, универсальные алгоритмы, описываемые ниже, не могут применяться к стандартным очередям.

Тема 21.12
Объявление и инициализация очереди

Объявление очереди должно задавать тип хранимых элементов и вид используемого контейнера. Ниже приведены простые объявления очереди:

```
queue< int, list<int> > queueOne;
// Использует дек
queue< double> > queueTwo;
queue< Part *, list<Part * > > queueThree;
queue< Customer, list<Customer> > queueFour;
```

В последнем примере создается очередь элементов определенного пользователем типа Customer.

Приведем пример использования очереди:

```
#include <queue>
#include <string>
#include <deque>
#include <list>
#include <iostream>

using namespace std;

int main(void)
{
 //Создаем очередь, используя список в качестве контейнера
 queue<int> q;
 //Помещаем в него два значения
 q.push(1);
 q.push(2);
 //Извлекаем из очереди два значения
 cout << q.front() << endl;
 q.pop();
 cout << q.front() << endl;
 q.pop();
 //Создаем очередь строк, используя дек в качестве контейнера
 queue<string> qs;
 //Помещаем в очередь несколько строк
 int i;
 for (i = 0; i < 10; i++)
 {
  qs.push(string(i+1,'a'));
 }
 //Извлекаем из очереди строки
 for (i = 0; i < 10; i++)
 {
  cout << qs.front() << endl;
  qs.pop();
 }
 return 0;
}
```

При выполнении программа выводит на экран:

```
1
2
a
aa
aaa
aaaa
aaaaa
aaaaaa
aaaaaaa
aaaaaaaa
aaaaaaaaa
aaaaaaaaaa
```

Тема 21.13
Ассоциативные контейнеры

Если последовательные контейнеры предназначены для последовательного и произвольного доступа к элементам с помощью индексов или итераторов, то ассоциативные контейнеры разработаны для быстрого доступа к произвольным элементам с помощью ключей. STL предоставляет четыре ассоциативных контейнера: карту (map), мультикарту (multimap), множество (set) и мультимножество (multiset).

Карта – это вид ассоциативного контейнера, который предоставляет доступ к каждому элементу по уникальному ключу. Дублирование ключей не допускается. Иногда карты называют *ассоциативными списками* или *словарями*. При добавлении и удалении элементов из карты используются пары ключ/значение в виде:

```
map_Object[key_value] = object_value;
```

или в функциях push_back() и insert().

Мультикарта – это вид ассоциативного контейнера, в котором допускаются неуникальные ключи для доступа к элементам. Это означает, что несколько элементов могут иметь одинаковое значение ключа.

Множество – это вид ассоциативного контейнера, в котором хранятся только уникальные ключи.

Мультимножество – это вид ассоциативного контейнера, в котором хранятся неуникальные ключи.

Рассмотрим пример использования карты в качестве ассоциативного контейнера:

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
 char c;
 map<char, int> m;
 //Создаем ассоциативный список
 for (int i=0, i<10; i++)
  {
   m.insert(pair<char, int>('A' + i, i));
  }
 cout << "Введите ключ: ";
 cin >> c;
 map<char, int>::iterator iter;
 //Поиск значения в контейнере по ключу
```

```
iter = m.find(c);
if (iter != m.end())
cout << iter->second;
else
cout << "Соответствующего элемента нет\n";
return 0;
}
```

Эта программа вначале создает карту или ассоциативный список, состоящий из пар ключ/значение. В качестве ключей используются буквенные символы, начиная с буквы 'A', а в качестве значений – параметр цикла for. Для создания пар ключ/значение используется шаблонный класс pair. Поиск нужного значения в ассоциативном списке по ключу осуществляется с помощью функции find(). Эта функция возвращает итератор соответствующего ключу элемента или итератор конца списка, если итератор не найден.

Тема 21.14
Универсальные алгоритмы

STL предоставляет около 60 универсальных алгоритмов, которые выполняют рутинные операции над данными, характерными для контейнеров. Все они определены в заголовочном файле <algorithm> в пространстве имен std.

Чтобы понять, как работают универсальные алгоритмы, нужно познакомиться с понятием объектов-функций. *Объект-функция* – это объект класса, в котором определен перегруженный оператор вызова функции (т.е. operator()). В результате этот объект может вызываться как функция. Объекты-функции важны для эффективного использования универсальных алгоритмов библиотеки STL, поскольку в качестве параметров шаблонных алгоритмов можно передавать как указатели на функции, так и объекты-функции. Эта библиотека содержит много видов стандартных объектов-функций, которые Вы можете использовать в качестве базовых для создания собственных объектов-функций.

Для использования этих возможностей необходимо подключить к программе заголовочный файл <functional>. Объекты-функции, которые принимают один аргумент, называются *унарными*. Унарные объекты-функции должны содержать объявления типов данных argument_type и result_type. Аналогично, объекты-функции, которые принимают два аргумента, называются *бинарными*. Они должны содержать объявления типов данных

first_argument_type, second_argument_type и result_type. Классы unary_function и binary_function облегчают создание шаблонных объектов-функций. Объекты-функции, включенные в STL, и их назначение перечислены ниже в табл. 21.3.

Таблица 21.3

Стандартные объекты-функции и их назначение

Название	Операция
Арифметические функции	
plus	Сложение x + y
minus	Вычитание x – y
multiplies	Умножение x * y
divides	Деление x / y
modulus	Остаток x % y
negate	Унарный минус – x
Функции сравнения	
equal_to	Проверка на равенство x == y
not_equal_to	Проверка на неравенство x != y
greater	Сравнение больше x > y
less	Сравнение меньше, чем x < y
greater_equal	Сравнение больше или равно x >= y
less_equal	Сравнение меньше или равно x<= y
Логические функции	
logical_and	Логическое И x && y
logical_or	Логическое ИЛИ x || y
logical_not	Логическое НЕ ! x

Рассмотрим пример использования стандартных объектов-функций.

```
#include<functional>
#include<deque>
#include<vector>
#include<algorithm>
#include <iostream>
```

```
using namespace std;
//Создаем новый объект-функцию из унарной функции
template<class Arg>
class factorial: public unary_function
<Arg, Arg>
{
 public:
 Arg operator()(const Arg& arg)
 {
  Arg a = 1;
   for(Arg i = 2; i <= arg; i++)
    a *= i;
   return a;
  }
 };
 int main()
 {
  //Инициализируем дек массивом целых чисел
  int init[7] = {1,2,3,4,5,6,7};
  deque<int> d(init, init+7);
  //Создаем пустой вектор для хранения факториалов
  vector<int> v((size_t)7);
  //Преобразуем числа в деке в их факториалы
  //и сохраняем в векторе
  transform(d.begin(), d.end(), v.begin(), factorial<int>());
  //Выводим на экран результат
  cout << "Следующие числа: " << endl << "   ";
  copy(d.begin(),d.end(),
     ostream_iterator<int,char>(cout," "));
  cout << endl << endl;
  cout << "Имеют такие факториалы: " << endl << "   ";
  copy(v.begin(),v.end(),
     ostream_iterator<int,char>(cout," "));
  return 0;
 }
```

Программа выводит на экран:

```
Следующие числа:
  1 2 3 4 5 6 7
Имеют такие факториалы:
  1 2 6 24 120 720 5040
```

STL позволяет применять универсальные алгоритмы к контейнерам. Она предоставляет совокупность таких алгоритмов для поиска, сортировки, слияния, преобразования и т.д. Каждый из алгоритмов может применяться не только к стандартным контейнерам, включенным в библиотеку, но и к контейнерам, опреде-

ленным пользователем. В самом широком смысле все алгоритмы классифицируются на две категории: алгоритмы, изменяющие содержимое контейнера (например, fill(), copy() и sort()), и алгоритмы, не изменяющие содержимое контейнера (например, find(), search(), count() и for_each()). Названия и назначение стандартных алгоритмов перечислены в табл. 21.4. В таблице алгоритмы разбиты на категории по принципу типового использования.

Таблица 21.4
Универсальные алгоритмы STL

Алгоритм	Назначение
Алгоритмы инициализации последовательности	
fill()	Заполняет последовательность начальным значением
fill_n()	Заполняет n позиций последовательности начальным значением
copy()	Копирует последовательность в другую
copy_backward()	Копирует последовательность другую
generate()	Инициализирует последовательность с помощью генератора
generate_n()	Инициализирует n позиций последовательности с помощью генератора
swap_ranges()	Меняет местами значения двух последовательностей
Алгоритмы поиска	
find()	Находит элемент, соответствующий аргументу
find_if()	Находит элемент, соответствующий условию
adjacent_find()	Находит последовательные дубликаты элементов
find_first_of()	Находит первое появление одного члена последовательности в другой
find_end()	Находит последнее появление подпоследовательности в последовательности

Алгоритм	Назначение
search()	Ищет подпоследовательность в последовательности
max_element()	Находит максимальное значение в последовательности
min_element()	Находит минимальное значение в последовательности
mismatch()	Находит первое несовпадение в двух последовательностях
Алгоритмы для преобразований «по месту»	
reverse()	Устанавливает обратный порядок элементов в последовательности
replace()	Заменяет определенные значения новым
replace_if()	Заменяет элементы, соответствующие предикату
rotate()	Переставляет элементы последовательности относительно заданной точки
partition()	Разбивает элементы на две группы
stable_partition()	Разбивает элементы на группы, сохраняя исходный порядок
next_permutation()	Генерирует перестановки в последовательности
prev_permutation()	Генерирует перестановки в последовательности, имеющей обратный порядок элементов
inplace_merge()	Объединяет две смежные последовательности в одну
random_shuffle()	Переупорядочивает элементы последовательности случайным образом
Алгоритмы, удаляющие элементы	
remove()	Удаляет элементы, которые соответствуют заданному условию

Алгоритм	Назначение
unique()	Удаляет все элементы последовательности до первого дубликата
Алгоритмы, приводящие к скалярам	
count()	Подсчитывает число элементов с заданным значением
count_if()	Подсчитывает число элементов, соответствующих предикату
accumulate()	Сводит последовательность к скалярному значению
inner_product()	Дает скалярное произведение двух последовательностей
equal()	Проверяет равенство двух последовательностей
lexicographical_compare()	Лексикографически сравнивает две последовательности
Алгоритмы, генерирующие последовательности	
transform()	Преобразует каждый элемент
partial_sum()	Генерирует последовательность частичных сумм
adjacent_difference()	Генерирует последовательность из несовпадающих элементов двух последовательностей
Различные операции	
for_each()	Применяет функцию к каждому элементу совокупности

Для использования универсальных алгоритмов нужно подключить в программу соответствующий заголовочный файл. Большинство алгоритмов определены в заголовочном файле <algorithm>. Однако алгоритмы accumulate(), inner_product(), partial_sum(), и adjacent_difference() определены в заголовочном файле <numeric>.

В качестве примера алгоритма, не изменяющего содержимое контейнера, рассмотрим алгоритм for_each(). Этот алгоритм имеет следующий интерфейс вызова:

```
template
<class InputIterator, class Function>
void for_each(InputIterator first,
    InputIterator last,
    Function f);
```

Алгоритм for_each применяет функцию f ко всем членам последовательности в диапазоне [first, last], где first и last – итераторы, которые определяют последовательность. Так как этот алгоритм является неизменяющим контейнер, функция f не должна производить какие-либо изменения в указанной последовательности. Если функция f возвращает результат, он игнорируется. В приведенном ниже примере алгоритм for_each() используется для перебора элементов вектора:

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
using namespace std;
 //Класс для создания объектов-функций,
 //которые умножают свой аргумент на x
 template <class Arg>
 class out_times_x: private
unary_function<Arg,void>
 {
  private:
    Arg multiplier;
  public:
 out_times_x(const Arg& x):
multiplier(x){}
  void operator()(const Arg& x)
    {cout << x * multiplier << " " << endl;}
 };
int main()
{
 int seq[5] = {1,2,3,4,5};
 //Создаем вектор
 vector<int> v(seq,seq + 5);
//Создаем объект-функцию
 out_times_x<int> fmult(2);
 //Применяем объект-функцию
 for_each(v.begin(),v.end(),fmult);
 return 0;
}
```

Упомянутый здесь стандартный класс unary_function предназначен для создания унарного объекта-функции (для создания бинарных объектов-функций библиотека шаблонов содержит класс binary_function).

При выполнении программа выводит на экран:

```
2
4
6
8
10
```

В качестве примера алгоритма, изменяющего контейнер, рассмотрим алгоритм fill() заполнения последовательности заданным значением. Он имеет следующий интерфейс вызова:

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

Приводимый ниже пример демонстрирует использование этого алгоритма:

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;
int main()
{
 int d1[4] = {1,2,3,4};
 //Создаем два вектора
 vector<int> v1(d1,d1 + 4), v2(d1,d1 + 4);
 //Создаем пустой вектор
 vector<int> v3;
 //Заполняем вектор v1 цифрами 9
 fill(v1.begin(),v1.end(),9);
 //Заполняем первые три элемента вектора v2 цифрами 7
 fill_n(v2.begin(),3,7);
 //Используем итератор insert,
 //чтобы заполнить 11 элементов вектора v3 цифрами 5
 fill_n(back_inserter(v3),5,11);
 //Выводим все три вектора на экран
 ostream_iterator<int,char> out(cout," ");
 copy(v1.begin(),v1.end(),out);
 cout << endl;
 copy(v2.begin(),v2.end(),out);
 cout << endl;
 copy(v3.begin(),v3.end(),out);
 cout << endl;
 return 0;
}
```

При выполнении программа выводит на экран:

```
9 9 9 9
7 7 7 4
11 11 11 11 11
```

В заключение заметим, что мы привели лишь беглое описание библиотеки STL. Вместе с тем, если Вы освоили изложенный здесь материал, ее использование не должно вызвать у Вас затруднений.

Практикум «Стандартная библиотека шаблонов (STL)»

Упражнение 21.1
Векторы и списки

Проанализируйте приведенный ниже пример использования векторов и списков.

Пусть необходимо составить программу, обеспечивающую вывод некоторого набора данных в виде таблицы на форме. Как известно, в таблице данные представляются в виде строк и столбцов, в пересечении которых располагаются ячейки с информацией. Число столбцов строго определено, в то время как число строк может меняться в зависимости от результата выборки данных.

Для упрощения решения поставленной задачи допустим, что информация в ячейках может храниться только в виде динамических символьных массивов (char*), а каждая колонка содержит информацию о своем названии и ширине. Предположим, что строка состоит только из набора входящих в нее ячеек, которые в свою очередь могут быть либо отображаемыми, либо невидимыми.

Поскольку предполагается, что число столбцов относительно невелико (по сравнению с количеством строк), не будет меняться и каждая строка содержит заведомо известное число ячеек с данными, имеет смысл определить в качестве контейнера для хранения ячеек в строке класс vector. Так как в конейнерном классе list содержатся удобные функции вставки и удаления элементов, воспользуемся им для хранения набора строк таблицы.

В результате получим следующий набор классов:

```
#include <vector>
#include <list>
// Определим короткие имена для типов:
typedef unsigned int   uint;
typedef unsigned char  byte;
// Класс формы
class CForm
{
public:
   CForm();
   ~CForm();
   // Описание класса
protected:
   // Описание переменных
private:
   // Описание переменных
};
// Класс отображаемых ячеек
class CLabel
{
public:
   CLabel();
   ~CLabel();
   // Описание класса
protected:
   // Описание переменных
private:
   // Описание переменных
};
// Ячейка
class CCell
{
public://CCell
   CCell();
   CCell(char* text);
   ~CCell();
   void SetText(char* text);
   char* GetText(){return Text;}
protected://CCell
   char *Text;
private://CCell
};
// Колонка
class CCol
{
public://CCols
```

```
    CCol();
    CCol(uint width);
    ~CCol();
    void SetCaption(char* txt);
    char* GetCaption(){return Caption;}
    uint GetWidth(){return Width;}
protected://CCols
    char *Caption;
    uint Width;
private://CCols
};
// Строка
class CRow
{
public://CRow
    CRow();
    CRow(CRow* row);
    CRow(uint height);
    ~CRow();
    void AddCell(CCell* cell);
    void AddCell(char* text);
    char* GetText(char* buf);
    char* GetCellText(byte idx);
    CCell* GetCell(byte idx);
    byte GetCellCount(){return Cells.size();}
    void Clear();
protected://CRow
    vector<CCell*> Cells;
private://CRow
};
// Таблица
class CGrid
{
public://CGrid
    CGrid(CForm* parent, uint x, uint y, uint w, uint h);
    virtual ~CGrid();
    void AddCol(uint width, char* caption);
    void AddRow(uint height=20);
    void AddRow(CRow* row);
    void SetCaption(CRow* row);
    void DeleteRow(uint index);
    void DeleteCol(uint index);
    void Show();
    void Hide();
    short GetColCount(){return Cols.size();}
    uint GetRowCount(){return Rows.size();}
    void SetBackground(const byte color);
    void SetForeground(const byte color);
```

```
    void SetCaptionFont(const char*, int size);
    void SetFont(const char*, int size);
    char* GetText(uint row, byte col);
    void SetText(uint row, byte col, char* text);
protected://CGrid
    CForm *hParent;              // Указатель на родительскую
                                 // форму
    vector<CCol*> Cols;          // Набор ширин колонок
    list<CRow*> Rows;            // Набор строк
    vector<CLabel*> Labels;      // Набор видимых ячеек
    vector<CLabel*> Captions;    // Заголовки колонок
    uint Height;                 // Высота фрейма
    uint Width;                  // Ширина фрейма
    uint X;                      // Начальные координаты
    uint Y;                      // Начальные координаты
    uint FirstVisibleRow;        // Индекс первой видимой строки
private://CGrid
    byte Cursor;                 // Номер строки с курсором
    byte VisibleRows;            // Число отображаемых строк
    void DrawCaption();          // Прорисовка заголовка
    void DrawLabels();           // Прорисовка ячеек
    CLabel* GetLabel(uint row, byte col);
    void UpdateLabel(uint row, byte col);
    CRow* GetRow(uint idx);
    void SetRow(uint idx, CRow* row);
    void SetLabelText(uint row, byte col, char* text);
};
int main()
{
    CForm *frm = new CForm(0, 0, 600, 400);
    CRow *row = new CRow;
    CGrid *grid = new CGrid(app, 50, 50, 500, 300);
    grid->AddCol(100, "Колонка-1");
    grid->AddCol(100, "Колонка-2");
    grid->AddCol(100, "Колонка-3");
    grid->SetCaption(row);
    grid->SetCaptionFont(ARIAL_BOLD, 10);
    row->AddCell("1234567890");
    row->AddCell("9876543210");
    row->AddCell("Пример");
    for(int i=0; i<57; i++) {grid->AddRow(row);}
    char num[5] = "";
    for(int i=0; i<58; i++) {
        itoa(num,i);
        grid->SetText(i,0,num);
    }
    grid->Show();
}
```

Реализация функций классов зависит от того, в какой операционной системе предполагается использовать данный пример.

Упражнение 21.2
Списки

Модифицируйте приведенный выше пример с тем, чтобы в нем в качестве контейнеров использовались только двусвязные списки.

Упражнение 21.3
Очереди

Проанализируйте пример моделирования обработки очереди задач многопроцессорной системой.

Имеется генератор заданий, который ставит их в очередь конвейера. Далее конвейер определяет, свободен ли один из имеющихся в наличие процессоров (ЦПУ), и, если ресурс доступен, отправляет задачу на обработку. Пусть каждое задание (см. класс Task) обладает некоторым свойством (flops), определяющим количество процессорных операций, которые необходимо затратить для его выполнения, а ЦПУ (класс CPU), в свою очередь, обладает определенной пропускной способностью (capacity).

```
#include <queue>
#include <stdlib.h>
// Класс задач
class Task
{
   unsigned int flops;
public:
   Task(unsigned int f):flops(f){}
   ~Task(){}
   unsigned int GetFlops(){return flops;}
};
// Класс ЦПУ
class CPU
{
   int capacity;
   int free;
public:
   CPU() {
            capacity=rand()%100;
```

```
            free=0;
          }
    CPU(int c):capacity(c){}
    ~CPU(){}
    int GetCapacity() {return capacity;}
    int GetFree(){return free;}
    int Proceed(int flops){
      if(flops<capacity)
          free=0;
      else
          free=capacity-flops;
      return free;
    }
};
// Функция-генератор задач
Task* Generate(int i)
{
   Task* t = NULL;
   if(i) t = new Task(rand()%100);
   return t;
}
// Класс-конвейер
class Pipeline
{
   vector<CPU*> cpu;
   queue<Task*> que;
public:
   Pipeline(){}
   ~Pipeline(){cpu.clear();while(!que.empty())que.pop();}
   void AddCPU(CPU* c){cpu.push_back(c);}
   void AddTask(int i){
      Task* t=NULL;
      for(int j=0; j<i; j++) {
         t=Generate(rand()%5);
         if(t)que.push(t);}}
   bool IsEmpty(){return que.empty();}
   void Proceed() {
      vector<CPU*>::iterator c=cpu.begin();
      Task* t;
      int f=0;
      while(!IsEmpty())
      for(unsigned int i=0; i<cpu.size(); i++,c++) {
x:
        if(!IsEmpty()){t = que.front();que.pop();}else break;
        f = (*c)->GetFree();
        if(f<0) f=-f;
        f += t->GetFlops();
        f=(*c)->Proceed(f);
```

```
            // Проверка, закончил ли ЦПУ выполнение задачи;
            // если да – взять из очереди следующую задачу,
            // если нет – перейти к следующему ЦПУ
            if((*c)->GetFree()>=0)
                goto x;
        }
    }
};
int main()
{
    Pipeline* pipeline = new Pipeline;
    CPU* cpu1 = new CPU;
    CPU* cpu2 = new CPU;
    pipeline->AddCPU(cpu1);
    pipeline->AddCPU(cpu2);
    randomize();
    for(;;) {
        pipeline->AddTask((rand()%5));
        if(pipeline->IsEmpty()) break;
        pipeline->Proceed();
    }
    return 0;
}
```

Сложность задачи и пропускная способность процессора для упрощения генерируются псевдослучайно, с помощью функции rand().

Вам предлагается обнаружить и устранить самостоятельно имеющиеся в приведенном выше листинге ошибки.

Приложение

UNICODE-Строки

Вначале системы программирования для языка C++ поддерживали только 7-битовые символы кода ASCII. Затем потребность в использовании этого языка программирования для других национальных языков привела к расширению этого кода до 8-разрядного. Дополнительные 128 символов были использованы для языков стран Восточной и Центральной Европы. В результате были созданы так называемые *кодовые страницы*. В частности, комитет ANSI установил для русского языка кодовую страницу 1251. Для поддержки языков, которые имеют значительно больше символов, например арабского, китайского, японского, потребовалась разработка *многобайтовых кодов (MBCS* – Multibyte Character Set*)*. С целью обеспечения обратной совместимости с ASCII многобайтовый набор символов был создан как надмножество набора символов ASCII. В MBCS символы кодируются одним или двумя байтами. В двухбайтовых символах первый или ведущий байт сигнализирует, что он и следующий за ним байт должны интерпретироваться как один символ. При этом первый байт берется из диапазона кодов, зарезервированных для ведущих байтов. Какой диапазон байтов может быть ведущим, зависит от используемой кодовой страницы. Недостатки MBCS-кодов привели к созданию нового стандарта.

Unicode-стандарт на двухбайтовые символы разработан компаниями Apple и Xerox в 1988 году. Затем в 1991 году был создан консорциум по внедрению Unicode, куда вошла и компания Microsoft. В настоящее время поддержка Unicode-символов стала составной частью стандарта языка C++. Unicode позволяет закодировать 65536 символов. В результате были созданы группы символов для различных языков. В частности, были созданы группы:

```
0000 – 007F – код ASCII
0100 – 017F – Европейские латинские
0400 – 04FF – Кириллица
```

В отличие от обычных 8-разрядных символов, символы стандарта Unicode называют *«широкими»*, а символы 8-разрядных кодов – *«узкими»*.

Широкие символы в C++ стали фундаментальным типом данных, описываемым ключевым словом wchar_t. Например, буфер для хранения Unicode-строки выделяется так:

```
wchar_t buf[100];
```

Для оперирования с Unicode-строками разработан новый набор функций, полностью соответствующий ANSI-функциям (strcpy, strcat и др.). Имена всех новых функций начинаются с префикса wsc (от англ. wide character set – набор широких символов). Чтобы получить имя Unicode-функции, просто замените префикс str у ANSI-функции на wcs. Например,

```
char* strcat(char*, const char*);
```

преобразуется в

```
wchar_t* wcscat(wchar_t*, const wchar_t*);
```

Строка, которой непосредственно предшествует символ L, рассматривается как строка широких символов:

```
const wchar_t str = L"Это - Unicode-строка.";
```

Unicode-символ определяется аналогично:

```
wchar_t ch = L'A';
```

Если вы хотите совершить условную компиляцию (для ANSI- или Unicode-символов), замените заголовочный файл <string.h> на <tchar.h>. Этот файл состоит из макросов, заменяющих реальные вызовы str- или wcs-функций. Если при компиляции определен символ _UNICODE, макросы ссылаются на wcs-функции, в противном случае – на str-функции. Для задания ANSI/ Unicode-символов применяйте тип TCHAR. Он определен так:

```
#ifdef _UNICODE
  typedeg wchar_t TCHAR;
#else
  typedef char TCHAR;
```

Таким образом, объявления строк можно писать так:

```
TCHAR buf[80];
```

Для определения литеральной константы соответствующего типа используйте макрос _T (или _TEXT), который определен следующим образом:

```
#ifdef _UNICODE
  #define _T(x)  L##x
#else
  #define _T(x)  x
```

Например, следующая символьная константа инициализируется ANSI- или Unicode-литеральной константой в зависимости от того, объявлен ли в программе символ _UNICODE:

```
TCHAR* buf = _T("Строка");
```

Кроме макросов для функций работы со строками, заголовочный файл <tchar.h> содержит много других макросов. В частности, он содержит макросы для функций ввода-вывода, часть из которых приведена в табл. 1.

Таблица 1

Некоторые макросы для функций ввода-вывода

Имя макроса в <tchar.h>	_UNICODE определен	_UNICODE не определен
_tprintf	Wprinf	printf
_stprintf	swprintf	sprintf
_vtprintf	vwprinf	vprintf
_tscanf	wscanf	scanf
_stscanf	swscanf	sscanf
_fgetts	fgetws	fgets
_fputts	fputs	fputws
_tfopen	fopen	_wfopen

Используя идентификаторы из левой колонки табл. 1, Вы можете писать код, который будет компилироваться и для Unicode, и для ANSI.

Следующие две функции являются частью соответствующих операционных систем и будут работать только в Windows 95/98/Millenium и Windows NT/2000/XP. Их объявления содержатся в заголовочном файле <winnls.h>.

Для преобразования MBCS- и ANSI-сиволов в Unicode-символы используется функция:

```
Int MultiByteToWideChar(UINT uCodePage,
DWORD dwFlags,
LPCSTR lpMultiByteStr,
int cchMultiByte,
LPWSTR lpWideCharStr,
int cchWideChar);
```

Здесь:

uCodePage – номер кодовой страницы для многобайтовой строки;

dwFlags – флажки, используемые для преобразования диакритических символов; обычно – 0;

lpMultiByte – указатель на преобразуемую строку;

cchMultiByte – длина преобразуемой строки в байтах; если указано 1, функция самостоятельно определяет длину строки;

lpWideCharStr – указатель на Unicode-строку; если задано NULL, следующий параметр должен быть равен нулю;

cchWideChar – размер буфера для Unicode-строки в символах; если параметр равен нулю, функция просто возвращает размер буфера, необходимый для сохранения результата преобразования.

Если функция успешно завершилась и параметр cchWideChar!=0, она возвращает число широких символов, записанных в буфер, на который указывает параметр lpWideCharStr.

Пример:

```
MultiByteToWideChar(CP_ACP, 0, lpMultiByteStr, -1,
   lpWideCharStr, nLengOfWideCharStr);
```

Обратное преобразование осуществляется функцией:

```
int WideCharToMultiByte(UINT uCodePage,
DWORD dwFlags,
LPCWSTR lpWideCharStr,
int cchWideCharStr,
LPSTR lpMultiByteStr,
int cchMultiByteStr,
LPCSTR lpDefaultChar,
LPBOOL lpfUsedDefaultChar);
```

Большая часть параметров этой функции имеет то же назначение, что и у предыдущей. Опишем здесь вновь встретившиеся параметры:

lpDefaultChar – указатель на символ, который используется для непреобразуемых в ANSI-символы Unicode-символов; обычно он равен NULL и в этом случае используется символ по умолчанию: знак вопроса ("?");

lpfUsedDefaultChar – указатель на булевскую переменную; при возврате из функции эта переменная получит значение TRUE, если хотя бы один Unicode-символ не имеет ANSI-соответствия.

Пример:

```
WideCharToMultiByte(CP_ACP, 0, lpWideCharStr,-1, lpMultiByteStr,
strlen(lpMultiByteStr), NULL, NULL);
```

Содержание

Учебное издание

ГЛУШАКОВ Сергей Владимирович
КОВАЛЬ Александр Викторович
СМИРНОВ Сергей Викторович

ПРАКТИКУМ ПО C++

Главный редактор *Н. Е. Фомина*
Художественный редактор *С. И. Правдюк*
Технический редактор *А. С. Таран*
Компьютерная верстка: *С. А. Теребилов*
Корректор *О. А. Кравец*

Подписано в печать 20.02.06. Формат 84x108 $^{1}/_{32}$.
Бумага газетная. Гарнитура Таймс. Печать офсетная.
Усл. печ. л. 27,72. Усл. кр.-отт. 28,14. Уч.-изд. л. 29,25.
Тираж 4000 экз. Заказ № 6-25

ТОВ «Видавництво Фоліо»
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготівників
і розповсюджувачів видавничої продукції
ДК № 502 від 21.06.2001 р.

ТОВ «Фоліо»
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготівників
і розповсюджувачів видавничої продукції
ДК № 683 від 21.11.2001 р.

61057, Харків, вул. Донець-Захаржевського, 6/8

Електронна адреса:
www.folio.com.ua
E-mail: realization@folio.com.ua
Інтернет магазин
www.bookpost.com.ua

Надруковано з готових позитивів
у ТОВ «ФОЛІО-АРТ»,
61143, м. Харків, вул. Велика кільцева, 99
Свідоцтво про реєстрацію
ДК №2271 від 26.08.2005 р.

Глушаков С. В., Коваль А. В., Смирнов С. В.

Г 55 Практикум по С++ / Худож.-оформитель С. И. Правдюк. — Харьков: Фолио, 2006. — 525 с. — (Учебный курс).

ISBN 966-03-2661-0.

Книга содержит практическое руководство по языку С++. Кроме этого, в ней описываются некоторые важнейшие функции языка С, определенные в стандарте ANSI C, Unicode-строки и ставшая составной частью языка стандартная библиотека шаблонов (STL).

Большое внимание уделяется практическому применению С++. Каждый раздел содержит множество прмеров и практикум, включающий задания для анализа и самостоятельной работы.

Издание будет полезно студентам, изучающим язык С++, начинающим программистам, желающим повысить свою квалификацию, и профессионалам, которые зхотят получить справочные сведения по конкретным вопросам.

ББК 32.97