МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«УДМУРТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Филиал ФГБОУ ВО «УдГУ» в г. Воткинске

МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО ДИСЦИПЛИНЕ «ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ», «ОЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ»

Для студентов направлений подготовки:

Прикладная информатика (по отраслям)
09.03.03
Направленность (Профиль)
Прикладная информатика в экономике
09.03.03.02

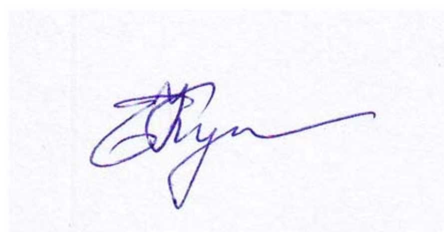
Воткинск 2016

В учебном пособии рассматриваются основы языка C++, методы процедурного и объектно-ориентированного программирования на этом языке. Учебное пособие предназначено в первую очередь для студентов направления подготовки 09.03.03 Прикладная информатика. Также оно будет полезным для студентов направления подготовки 38.03.05 Бизнес-информатика и студентов специальности 09.02.04 «Информационные системы (по отраслям)».

Коллектив авторов:

Екатерина Сергеевна Вяткина, старший преподаватель
Раскин Павел Николаевич, к.т.н, доцент

Методическое пособие одобрено на заседании кафедры «Информационных и Инженерных Технологий»
Протокол №10 от 21.04.16



Заведующий кафедрой / Кучерова Е.А./

Пособие утверждено на заседании научно-методического совета Филиала ФГБОУ ВО «УдГУ» в г. Воткинске
Протокол №3 от «29» апреля 2016 г.
Председатель научно-методического совета

………………………………/Мелекесова Р.М.

1. Введение

Язык программирования С++ был разработан на основе языка С Бьярном Страуструпом.

Авторство языка С принадлежит Денису Ритчи, сотруднику AT&T Bell Laboratories (1970 г.). Сначала язык программирования С был написан для создания и поддержки операционной системы UNIX. До того времени все программы операционной системы UNIX были написаны либо на языке ассемблера, либо на языке В, разработанном Кеном Томпсоном — создателем системы UNIX. Язык С – это язык общего назначения, и он может быть использован для написания различных программ, но его популярность была связана в основном с операционной системой UNIX. Для поддержки системы UNIX необходимо было писать программы на языке С. Язык С и UNIX настолько подходили друг к другу, что вскоре почти все коммерческие программы для среды UNIX оказались написанными на С. Язык С стал настолько популярным, что его специально приспособили и для других операционных систем, и его применение не ограничивалось только UNIX-системами. Несмотря на такую популярность, язык С не был лишен недостатков.

Особенность языка С заключается в том, что будучи языком высокого уровня, он сохранил многие черты языка низкого уровня. Язык С расположен где-то между языками очень высокого уровня и языками низкого уровня, и в этом его сильные и одновременно слабые стороны. Как и язык ассемблера (язык низкого уровня), язык программирования С может непосредственно управлять памятью компьютера. С другой стороны, С обладает чертами языков высокого уровня, поэтому программы на языке С читать и создавать легче, чем программы на языке ассемблера. Язык С является очень удобным для написания системных программ, но в программах на языке С для иных целей зачастую трудней разобраться, нежели в программах, написанных на других языках. Язык С также имеет меньше возможностей для автоматической проверки программ, чем некоторые друге языки высокого уровня.

Чтобы преодолеть эти и другие недостатки языка С, Бьярн Страуструп из AT&T Bell Laboratories в 1980 году разработал на его основе язык программирования С++. Большая часть С – это подмножество языка С++, и, таким образом, большинство С-программ являются также С++-программами. В программах на языке С++ можно использовать тексты на языке С и обращаться к библиотечным функциям языка С. Основное отличие языка С++ от С заключается в реализации объектно-ориентированного подхода программирования – чрезвычайно мощного современного способа программирования.

Через несколько лет практического использования языка С++ стандартом языка фактически стала спецификация языка AT&T C++ release 2.0, разработанная под руководством Б. Страуструпа в Bell Laboratories фирмы AT&T. В настоящее время в Американском Национальном Институте Стандартов (ANSI) существует комитет по языку С++ (X3J16). Изданное в 1990 году описание языка с комментариями принято комитетом ANSI в качестве исходного материала для стандартизации С++. Последняя версии стандарта С++ ISO/IEC JTC1 (полное название: «International Standard ISO/IEC 14882:2014(E) Programming Language C++») датирована 18 августа 2014 года.

В настоящее время широкое распространение получили разные реализации языка в системах Visual Studio C++, Borland C++ Builder, Rad Studio, Dev-C++, Qt-SDK и др.

2. Элементы языка

2.1. Первые программы

Напишем простейшую программу.

```
#include< iostream >
using namespace std;
void main(){
cout<<"Hello, World!\n";
}
```

Здесь в первой строке расположена так называемая **директива препроцессора include** (включающая директива). Выполнение этой директивы приведет к тому, что вместо первой строки в программу будет вставлено содержимое файла iostream (или iostream), (такие файлы называются **хедерами**, или заголовочными файлами; эти файлы содержат тексты на языке C++) и компилятор после этого будет обрабатывать новый полученный текст программы.

Вторая строка using namespace std; определяет так называемое-пространство имен **std**.

В третьей строке исходного текста программы находится заголовок функции с именем **main**. Пустые круглые скобки говорят о том, что эта функция не имеет аргументов, а **ключевое слово void** означает, что функция main не возвращает никакого значения. В фигурных скобках находится блок, который часто называют телом программы.

В третьей строке стоит оператор, действие которого – вывод в поток **cout** (на английском звучит как **see-out**), который здесь ассоциируется с экраном дисплея, строки

Hello, World!

Символ '\n' в конце текста в кавычках сообщает компилятору, что после вывода текста на экран нужно перейти на новую строку.

Теперь приведем пример простой программы и диалога, который получается на экране компьютера, если пользователь запускает эту программу и вводит соответствующие данные. В дальнейшем **пользователем** мы будем называть человека, который использует программу. Данные, введенные пользователем, выделены в тексте диалога жирным

шрифтом. **Программистом** мы будем в дальнейшем называть, естественно, автора программы.

```
#include <iostream>
using namespace std;
void main(){
int m, n, sum;                              // Описания.
cout<<”Для ввода чисел задайте два числа на клавиатуре\n”;
cout<<”(через пробелы) и нажмите Enter.\n”;
cin>>m>>n;                                  // Ввод чисел.
sum = m + n;
cout<<”При m=  ”<<m<<”  и  n= ”<<n
<<”    их  сумма равна ”<<sum<<”.\n”;       // Вывод результата.
}
```

При выполнении этой программы на экране монитора появится следующий диалог:

```
Для ввода чисел задайте два числа на клавиатуре
(через пробелы) и нажмите Enter.
20  45
При m=  20  и  n= 45    их  сумма равна 65.
```

В этой программе за символами // стоят **комментарии**. В третьей строке **описываются** целые переменные с **именами** m, n и sum.

Оператор, начинающийся со слова **cin** сообщает компилятору, что введенные пользователем значения, равные 20 и 45, нужно поместить соответственно в **переменные** m и n. Объект cin (читается как **see-in**) – поток ввода – здесь понимается как клавиатура, а стрелки << и >> указывают направление, в котором перемещаются данные.

Перейдем теперь к описанию языка C++ и его возможностей.

2.2. Алфавит языка

В алфавит языка входят:

- Прописные латинские буквы A .. Z
  Строчные латинские буквы a .. z
  Арабские цифры 0 .. 9
  Символ подчеркивания _ (рассматривается как буква).

Все эти символы используются для образования ключевых слов и имён языка.

Имя есть последовательность букв и цифр, начинающаяся с буквы и не являющаяся ключевым словом. (Символ _ в начале имени ставить не рекомендуется).

В С++ прописные и строчные буквы различаются, поэтому имена ARG1 и arg1 являются различными.

- Знаки пунктуации и специальные символы, приведенные в табл. 1.

Таблица 1

Знаки пунктуации и специальные символы

Символы	**Наименование**	**Символы**	**Наименование**
,	запятая	{	открывающая скобка
.	точка	}	закрывающая скобка
;	точка с запятой	<	меньше
:	двоеточие	>	больше
?	знак вопроса	[	открывающая скобка
‘	апостроф	]	закрывающая скобка
!	восклицательный знак	#	номер или решетка
|	прямая черта	%	процент
/	слэш	&	амперсанд
\	обратный слэш	^	НЕ-логическое
~	тильда	-	минус
*	звездочка	=	равенство
(	открывающая скобка	“	кавычки
)	закрывающая скобка	+	плюс

- Пробельные символы.

К этой группе относятся пробел, символы табуляции, перевода строки, возврат каретки, перевода страницы. Эти символы отделяют друг от друга **лексемы** языка. Любая последовательность пробельных символов рассматривается при компиляции как один пробел.

2.3. Комментарии

Комментарии ограничиваются так:

// символы до конца строки.

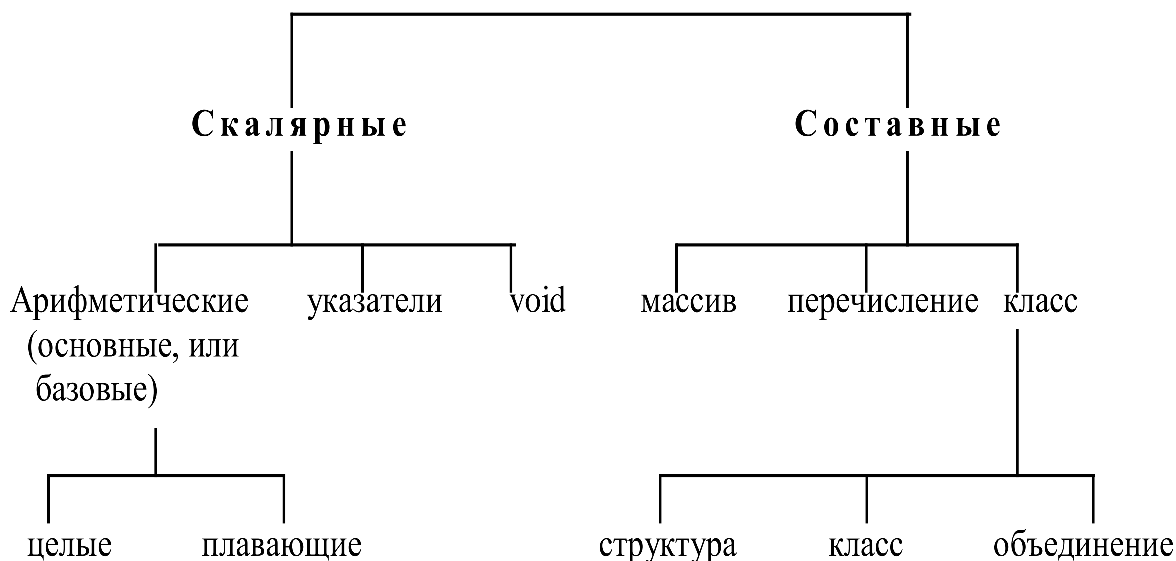
или так:

```
/* символы
 символы
 . . .
 символы */
```

В комментариях символы – это не только литеры из алфавита языка C++, но и любые возможные символы, включая русские буквы.

2.4. Типы данных

Все типы данных можно разделить на две категории: скалярные и составные.



Ключевыми словами, используемыми при объявлении основных типов данных, являются:

Для целых типов:

char, int, short, long, signed, unsigned;

Для плавающих типов:

float, double, long double;

Для классов: **structure union class**;

Для перечисления: **enum**;

Для типа void: **void** (пустой).

2.5. Целые типы данных

Тип char, или символьный

Данными типа char являются различные символы, причем значением этих символов является численное значение во внутренней кодировке ЭВМ.

Символьная константа – это символ, заключенный в апострофы, например: ‘&’, ‘4’, ‘@’, ‘a’. Символ ‘0’, например, имеет в кодировке ASCII значение 48.

Существуют две модификации этого типа:
signed char и unsigned char.

Данные char занимают один байт и меняются в диапазоне:
signed char (или просто char) -128 .. 127;
unsigned char 0 .. 255.

Отметим, что если необходимо иметь дело с переменными, принимающими значения русских букв, то их тип должен быть unsigned char, так как коды русских букв >127 (В кодировке ASCII).

Символы, в том числе и неграфические, могут быть представлены как символьные константы с помощью т.н. управляющих последовательностей.

Управляющая последовательность – это специальные символьные комбинации, которые начинаются с символа \, за которым следует буква или комбинация цифр (см. табл. 2).

Последовательности '\0ddd' и ‘\0xdd’ позволяют представлять любой символ из набора ЭВМ как последовательность восьмеричных или шестнадцатеричных цифр соответственно. Например, символ возврата каретки можно задать так: ‘\r’ или ‘\015’ или ‘0xD’.

Таблица 2

Специальные управляющие последовательности

Управляющая последовательность	Наименование
\a	Звонок
\b	Возврат на шаг
\t	Горизонтальная табуляция
\n	Перевод строки
\v	Вертикальная табуляция
\r	Возврат каретки
\f	Перевод страницы
\”	Кавычки
\’	Апостроф
\\	Обратный слэш

Тип **short** (эквивалент **short int**).

Данные типа short занимают 2 байта и принимают целые значения из диапазона: -32768 . . 32767.

Тип **int.**

Данные типа **int** в разных системах могут занимать либо

2 байта и принимать соответственно целые значения из диапазона 32768 . . 32767, либо

4 байта, и тогда их диапазон значений будет

-2147 483 648 .. 2 147 483 647.

Тип **unsigned short.**

Данные такого типа занимают 2 байта, их диапазон: 0 . . 65535.

Тип **long** (**long int**)

Такие данные занимают 4 байта и изменяются в диапазоне

-2 147 483 648 .. 2 147 483 647.

Тип **unsigned long** (**int**)

Такие данные занимают 4 байта и изменяются в диапазоне

0 . . 4 298 876 555.

Отметим, что если целая константа выходит из диапазона **int**, то она автоматически становится константой типа **long** или даже **unsigned long**.

Так, 32768 имеет (при двухбайтном int) тип **long**, 2676768999 имеет тип **unsigned long**.

Задать тип константы можно и явно с помощью суффиксов ‘U’ и ‘L’:

-6L 6U 33UL.

Заметим, что в стандарте языка определено лишь, что sizeof(char)=1 и

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

Здесь **sizeof** (**type**) – операция, определяющая размер типа **type** в байтах.

Целая константа, которая начинается с нуля, является восьмеричной константой, а начинающаяся символами 0x – шестнадцатеричной константой, например

031	0750	01	– восьмеричные константы;
0x17	0xA9	0xFF	– шестнадцатеричные константы.

Тип **bool** (логический тип)

Относится к целым типам. Данные типа bool занимает один байт. Диапазон допустимых значений – целые числа от 0 до 255. Данные типа bool используются для хранения результатов логических выражений. У логического выражения может быть один из двух результатов **true** или **false**; true – если логическое выражение истинно, false – если логическое выражение ложно.

Так как диапазон допустимых значений типа данных bool от 0 до 255, то нужно сопоставить этот диапазон с определёнными в языке логическими константами true и false. Константе true эквивалентны все числа от 1 до 255 включительно, тогда как константе false эквивалентно только одно целое число – 0.

2.6. Плавающие типы данных

Информацию о данных плавающих типов, которые представляют в ЭВМ вещественные числа, приведем в табл. 3[а,б].

В среде Borland C++, Rad Studio:

Таблица 3[а]

Тип	Длина	Диапазон	Десятичных цифр
float	4	3.4e-38 .. 3.4e38	7
double	8	1.7e-308 .. 1.7e308	15
long double	10	3.4e-4932 .. 1.1e4932	19

В Visual Studio C++:

Таблица 3[б]

Тип	Длина	Диапазон	Десятичных цифр
float	4	3.4e-38 .. 3.4e38	7
double	8	1.7e-308 .. 1.7e308	15
long double	8	1.7e-308 .. 1.7e308	15

По умолчанию, плавающие константы имеют тип **double**, если они не выходят из соответствующего диапазона:

1.0, .3 -6. 2.3e-6 (означает $2.3 \cdot 10^{-6}$), 3e-19, 1.2 – типа **double**.

Суффикс ℓ говорит о том, что плавающая константа будет иметь тип **long double**:

3ℓ, 3E8ℓ, 1.6e-19ℓ, 1.3e-200ℓ – типа **long double**.

2.7. Константы-строки, или литералы

Константа-строка – это последовательность символов, взятая в кавычки:

“строка“ “\“a+b=c\“ – это равенство“.

Строковый литерал представляется в памяти как массив элементов типа **char**.

При компиляции в конец каждой строки автоматически добавляется так называемый нуль-символ ‘\0’, являющийся признаком конца строки. Таким образом, в памяти литерал “три” занимает не три, а четыре байта.

Значением строки является адрес ее начала.

2.8. Директива препроцессора *define*

Директива препроцессора ***define*** имеет вид

#define имя текст_подстановки, например,

```
#define  nmax   1000
#define  km     (nmax*3+1)
```

Имя, которое указано в **#define,** в области его видимости заменяется в тексте программы текстом подстановки. Таким образом, вместо имени nmax (в области его видимости) в тексте везде появится 1000, а вместо km – (1000*3+1).

Отметим, что есть еще одна форма директивы #define (с параметрами).

2.9. Описания

Все переменные должны быть описаны до их использования. Описание состоит из спецификатора типа и следующего за ним списка переменных, которые будут иметь указанный тип:

```
int i, j, k, pmax;
float radix, a, b, s_m;
double k, kr;
char ch, ch1;
char symbol;
```

При описании переменная может быть инициализирована некоторым значением, например:

```
char t=‘t’, BACKSLASH=‘\\’;
int i=0, j, k, s=1;
float ro, eps=1e-6;
```

2.10. Модификатор *const*

Если в объявлении имени присутствует модификатор **const**, то объект, с которым сопоставлено данное имя, рассматривается в области существования этого имени как константа. Например:

```
const int i=50;
const double pi=3.14159265358979;
```

Такие именованные константы в программе изменять нельзя. Использовать эти константы можно так же, как и обычные.

3. Выражения

Выражение – это сочетание различных операндов и операций. Например:

a+b a/b c<<d и т.д.

3.1. Операция и выражение присваивания

Операция присваивания обозначается символом ‘=‘

Простейший вид операции присвоения:

v = **e**

Здесь **v –** любое выражение, которое может принимать значение, **e** – произвольное выражение.

Операция присвоения выполняется справа налево, т.е. сначала вычисляется значение выражения **e**, а затем это значение присваивается левому операнду **v**. Левый операнд в операции присваивания должен быть т.н. **адресным выражением**, которое иначе называют **ℓ-value**. Примером адресного, или именующего, выражения является имя переменной.

Не является ℓ- value, например, выражение a+b.

Адресным выражением никогда не являются константы.

В языке C++ операция присваивания образует выражение присваивания, то есть

a = b

означает не только засылку в **a** значения **b**, но и то, что a = b является выражением, значением которого является левый операнд после присвоения.

Отсюда следует, что возможна, например, такая запись:

a = b = c = d = e + 2;

Итак, результатом выражения присваивания является его левый операнд. Если тип правого операнда не совпадает с типом левого, то значение справа преобразуется к типу левого операнда (если это возможно). При этом может произойти потеря значения, например:

```
int i; char ch;
i=3.14; ch=777;
```

Здесь **i** получает значение 3, а значение 777 слишком велико, чтобы быть представленным как **char**, поэтому значение **ch** будет зависеть от способа, которым конкретная реализация производит преобразование из большего в меньший целый тип.

Существует так называемая комбинированная операция присваивания вида

a оп =b, здесь **оп** – знак одной из бинарных операций:

+ – * / % >> << & | ^ && ||.

Присваивание

a оп=b эквивалентно

a = a оп b , за исключением того, что адресное выражение вычисляется только один раз.

Примеры:

a+=2	*означает*	a=a+2
bottom_count [2*i+3*j+k]* =2	*означает*	bottom_count [2*i+3*j+k] = bottom_count [2*i+3*j+k]*2
s/=a	*означает*	s=s/a

Результатом операции присваивания является ее левый операнд; следовательно, ее результат – адресное выражение и поэтому возможна запись

(a=b)+=c;

Это эквивалентно следующим двум операторам:

a=b; a=a+c;

3.2. Арифметические операции

Бинарными арифметическими операциями являются:

\+ – * / %.

(Существуют также унарные + и –).

При делении **целых** дробная часть **отбрасывается**.

Так, 10/3 дает 3, в то время как 10/3.0 дает 3.33333...

Операция a % b применяется только к целым операндам и дает остаток от деления a на b, так

10%3 дает 1,
2%3 дает 2,
12%2 дает 0.

3.3. Операции отношения

Операциями отношения являются => > <= <.

Все они имеют одинаковое старшинство. Непосредственно за ними по уровню старшинства следуют операции **равенства** и **неравенства**:
= = (равно), != (не равно) с одинаковым старшинством.

Операции отношения младше арифметических операций, так что выражения типа i < lim+3 понимаются как i < (lim+3).

Операция сравнения определяет некоторое выражение. Значение этого выражения равно логической константе **true** (1), если условие, выраженное сравнением, выполняется и равно **false** (0), если нет.

3.4. Логические операции

К логическим операциям относятся:
унарная операция логическое НЕ, **!** (отрицание);
бинарная операция логическое И , **&&** (конъюнкция);
бинарная операция логическое ИЛИ, **||** (дизъюнкция).

Операнды логических операций могут быть целых, плавающих и некоторых других типов, при этом в каждой операции могут участвовать операнды различных типов.

Операнды логических выражений вычисляются слева направо.

Результатом логической операции является **false** или **true** типа **bool**.

Операция **!операнд** дает 0, если операнд ненулевой и 1 если операнд равен нулю.

Операция && (И-логическое, логическое умножение) дает значение 1, если оба операнда имеют ненулевое значение. Если один из операндов равен 0, то результат также равен 0. Если значение первого операнда равно 0, то второй операнд не вычисляется.

Операция || (ИЛИ-логическое, логическое сложение) вырабатывает значение 0, если оба операнда равны 0. Если какой-нибудь из операндов имеет ненулевое значение, то результат операции равен 1. Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

По приоритету эти операции распределены так: !, &&, ||.

3.5. Побитовые операции

К побитовым, или поразрядным операциям относятся:

операция поразрядного И &;
операция поразрядного ИЛИ |;
операция поразрядного исключающего ИЛИ ^;
унарная операция поразрядного отрицания (дополнение) ~.
Кроме того, рассматриваются операции сдвигов <<, >>.
Операнды поразрядных операций могут быть любого целого типа.

Операция & сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба соответствующих бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция | сравнивает каждый бит первого операнда с соответствующим битом второго операнда; если любой из них или оба равны 1, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция ^ . Если один из сравниваемых битов равен 0, а другой равен 1, то соответствующий бит результата устанавливается в 1, в противном случае, т.е. когда оба бита равны 1 или оба равны 0, бит результата устанавливается в 0.

Операция ~ меняет в битовом представлении операнда 0 на 1, а 1 – на 0.

Побитовая операция & часто используется для маскирования некоторого множества битов, например:

C=N&0177

передает в **C** семь младших битов **N**, полагая остальные равными 0. (С первого нуля в С++ начинаются восьмеричные константы; с **0X** – шестнадцатеричные константы).

Пусть N =642. Приведем побитовое представление N восьмеричной константы 0177 и результата C:

N	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0
0177	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Операция | используется для включения битов:

C= N|MASK

устанавливает в 1 те биты в **N**, которые равны 1 в **MASK**.

Еще примеры:

```
short a=0x45ff, b=0x00ff;
short c;
c= a ^ b;   // c: 0x4500
c= a | b;   // c: 0x45ff
c= a & b; // c: 0x00ff
c= ~ a;     // c: -0x3a00
c= ~ b;     // c: -0x7f00
```

Этот фрагмент программы можно проиллюстрировать так:

a	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1
b	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
c=a^b	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
c=a|b	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1
c=a&b	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
c=~b	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0

С помощью операции & можно определить остаток от деления операнда типа unsigned int на 2, 4, 8, 16 и т.д. Для этого достаточно применить операцию & к делимому с масками 0x01, 0x03, 0x07, 0x0f, 0x1f и т.д. Например:

7&0x03 дает 3.

Другими словами, выделяются младшие биты числа, а остальные устанавливаются в 0.

3.6. Сдвиги

Операции сдвига << >>осуществляют, соответственно, сдвиг влево и вправо своего левого операнда на число битовых позиций, заданных правым операндом. Таким образом, X << 2 сдвигает X влево на 2 позиции, заполняя освобождающиеся биты нулями, что эквивалентно умножению на 4. Сдвиг вправо величины без знака сопровождается дополнением старших битов нулями. Сдвиг вправо такой величины на **n** битов эквивалентен целочисленному делению левого операнда на 2 в степени **n**.

Так,
5 << 3 дает 40;
7 >> 2 дает 1.

Отметим, что правый операнд должен быть константным выражением, т.е. выражением, включающим в себя только константы. Если правый операнд отрицателен или он больше, или равен числу битов левого операнда, то результат сдвига не определен. Типом результата операции сдвига является тип левого операнда.

3.7. Операции автоувеличения и автоуменьшения ++ и – –

Эти операции являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значение операнда на 1. Операнд должен быть целого или плавающего типа (или типа указатель) и быть не константным адресным выражением (т.е. без слова **const** в описании). Тип результата соответствует типу операнда.

Префиксная форма операций:

++ операнд   – – операнд

Постфиксная форма:

операнд ++   операнд – –.

Если знак операции стоит перед **операндом**, результатом операции является **увеличенное или уменьшенное** значение **операнда**. При этом результат является адресным выражением (*l*-value).

Если знак операции стоит после **операнда**, значением выражения будет значение **операнда**. **После использования** этого результата значение операнда увеличивается или уменьшается. Результат постфиксной формы этих операций не является *l*-value.

Примеры:

```
int i=0, j=0, k, l;
k = ++i;              // Здесь  k=1 и  i  стало рано 1;
l= j ++;              // l=0 , а  j  стало равно 1;
– – k;                // k=0;
+ + j;                // j  стало равно 2.
```

Иначе говоря, результат выполнения

k = ++i; тот же, что и в последовательности операторов
i = i+1; k = i;

А результат

k = i++; такой же, как и в
k = i; i = i+1;

3.8. Тернарная или условная операция

Тернарная операция, т.е. операция с тремя операндами, имеет форму

операнд1 ? операнд2 : операнд3

Первый операнд может быть целого или плавающего типа (а также указателем, ссылкой или элементом перечисления). Для этой операции важно, является значение первого операнда нулем или нет. Если **операнд1** не равен 0, то вычисляется **операнд2** и его значение является результатом операции. Если **операнд1** равен 0, то вычисляется **операнд3**, и его значение является результатом операции. Заметим, что вычисляется либо **операнд2**, либо **операнд3**, но не оба.

Пример:

```
max= a<=b? b:a;
```

Здесь переменной max присваивается максимальное значение из переменных a и b.

Если в условной операции **операнд2** и **операнд3** являются адресными выражениями, то тернарная операция может стоять слева от знака присваивания:

```
a<b? a:b = c*x+d;
```

Здесь значение выражения c*x+d присваивается меньшей из двух переменных a и b.

3.9. Операция следования

Символом операции следования является **,** (запятая). Выражения, разделенные этим символом, выполняются слева направо строго в том порядке, в котором они перечислены.

Результатом этой операции является результат последнего выражения. Если оно является адресным выражением, то и результат операции также является адресным выражением.

Примеры:

```
int a=3, b=8, c;     // здесь запятая – разделитель, а не операция;
c=(a++, a+b);        // здесь  a станет равно 4, а  c  12;
(b--,c)*=3;          // здесь b станет равно 7, а  c  36.
```

Операция следования часто используется в операторе **for**. В различные части этого оператора можно включить несколько выражений,

например, для параллельного изменения двух индексов. Это иллюстрируется функцией REVERSE(S), которая располагает строку S в обратном порядке на том же месте.

```
void  REVERSE ( char S[ ] ){
int C, I, J;
for ( I = 0, J = strlen (S) – 1; I < J; I++, J – –){
C = S[I]; S[I] = S[J]; S[J] = C; }
}
```

В этом примере strlen(S) – функция, вычисляющая число символов в строке S (без символа '\0').

Запятые, которые разделяют аргументы функций, переменные в описаниях и т.д., не имеют отношения к операции **запятая** и не обеспечивают вычислений слева направо.

3.10. Приоритеты операций и порядок вычисления

Сведения обо всех операциях С++ сведены в следующую табл. 4:

Таблица 4

Приоритет операций и порядок их выполнения

Приоритет	Операция	Примечание	Порядок выполнения
1	**:: -> .**	Разрешение контекста, извлечение	слева-направо слева-направо
	[ ]	индексирование массива	слева-направо
	( )	вызов функции	слева-направо
	( )	преобразование типа	
2	++ – – ~ !		справа-налево
	– +	унарный –, унарный +	справа-налево
	&	получение адреса	справа-налево
	*	разрешение указателя	справа-налево
	new, delete	работа с динамической памятью	справа-налево
	sizeof	определение размера	справа-налево
3	*	умножение	слева-направо
	/	деление	слева-направо
	%	остаток от целочисленного деления	слева-направо
4	→* **.***	извлечение	слева-направо
5	+	бинарное сложение	слева-направо
	–	бинарное вычитание	слева-направо

<table>
<tr><td>6</td><td><< >></td><td>сдвиги</td><td>слева-направо</td></tr>
<tr><td>7</td><td>< <=> =></td><td>сравнение</td><td>слева-направо</td></tr>
<tr><td>8</td><td>= =   !=</td><td>равно   не равно</td><td>слева-направо</td></tr>
<tr><td>9</td><td>&</td><td>побитовое И</td><td>слева-направо</td></tr>
<tr><td>10</td><td>^</td><td>XOR (исключающее ИЛИ)</td><td>слева-направо</td></tr>
<tr><td>11</td><td>|</td><td>побитовое ИЛИ</td><td>слева-направо</td></tr>
<tr><td>12</td><td>&&</td><td>И-логическое</td><td>слева-направо</td></tr>
<tr><td>13</td><td>||</td><td>ИЛИ-логическое</td><td>слева-направо</td></tr>
<tr><td>14</td><td>?:</td><td>тернарная операция</td><td>справа-налево</td></tr>
<tr><td>15</td><td>= *= /= %= += и т.д.</td><td>операция присвоения</td><td>справа-налево</td></tr>
<tr><td>16</td><td>,</td><td>следование</td><td>слева-направо</td></tr>
</table>

Первый приоритет является наивысшим. Большинство операций выполняется слева направо. Например, выражение a + b + c интерпретируется как (a + b) + c. Исключение составляют унарные операции, операции присваивания и операция ?: , которые выполняются справа налево: a = b = c выполняется как a = (b = c).

Если нужно изменить порядок действий, то применяются круглые скобки.

Выражение

7.*a + b/-c   интерпретируется как (7. *a) + (b/(-c)).

Можно изменить этот порядок:
7. * (a+b)/(-c).

Отметим, что выражения, в которые входит одна из бинарных операций * + & ^ |, могут перегруппировываться компилятором, даже если они заключены в круглые скобки. Для обеспечения нужного порядка вычисления можно использовать явные промежуточные вычисления. В C++ не фиксируется порядок вычисления операндов в выражении. Например, в

```
c=sin(a*x+b)+fabs(x);
```

может сначала быть вычислен первый операнд, а затем второй, а может быть, и наоборот. В простых случаях это не имеет значения. Но если необходим определенный порядок, нужно вводить промежуточные переменные.

4. Операторы

В С++ точка с запятой является признаком конца оператора.

4.1. Пустой оператор

Пустой оператор состоит из **;**. Он используется там, где по правилам языка должен находиться какой-либо оператор, а по логике программы там ничего выполнять не надо.

4.2. Оператор-выражение

Любое выражение, за которым следует **;** , является оператором. Такой оператор называется **оператор-выражение**.

Примеры:

```
i ++;
a = b+c;
c + = (a<b)? a:b;
x+y;                    //Здесь результат не используется
                        //и будет выдано предупреждение.
```

4.3. Составной оператор

Составной оператор иначе называют блоком. Он представляет фрагмент текста программы, заключенный в фигурные скобки и, как правило, объединяющий несколько операторов. Составной оператор должен использоваться там, где синтаксис языка требует наличие лишь одного оператора, а логика программы – сразу нескольких:

```
{i=5; c=sin(i*x); c++;}    // Это блок.
```

4.4. Объявления

В С++ объявления являются операторами языка и могут стоять там, где возможен любой другой оператор С++:

```
s = 0.3;  d/=s;  int k = 5;
d = s+2*k;
double f=s+d;   f*=k;
```

4.5. Условный оператор

Имеется две формы условного оператора:

1) **if** (**выражение**) **оператор1**
2) **if** (**выражение**) **оператор1 else оператор2**

Оператор1 выполняется в случае, если **выражение** принимает ненулевое значение. Если **выражение** принимает значение 0 (или указатель NULL), то выполняется **оператор2**.

Примеры:

```
if (a > b) c = a – b; else c = b – a;
if (i < j) i++; else {j = i – 3; i ++;}
```

При использовании вложенных операторов **if** текущий **else** всегда относится к самому последнему **if**, с которым еще не сопоставлен ни один **else**.

```
void main ( ){
int a = 2, b = 7, c = 3;
if (a > b) { if (b <c ) c = b;}
else  c=a;
cout<<"c="<<c<<".\n";
}
```

Здесь результатом будет вывод строки **c=2**.

Если опустить фигурные скобки в операторе **if**, то программа примет вид

```
void main ( ){
int a=2, b=7, c=3;
if (a > b)
if (b < c) c=b;
else  c = a;
cout<<"c="<<c<<".\n";
}
```

Здесь **else** относится ко второму **if**.

В результате выведется строка **c=3**.

4.6. Оператор выбора *switch*

Этот оператор позволяет передать управление одному из нескольких помеченных метками операторов в зависимости от значения целочисленного выражения. Метки оператора **switch** имеют специальный вид:

case целая_константа**:**

Вид оператора **switch**:

```
switch (целое_выражение){
 [объявления]
 [case константное_целое_выражение1:]

...

 [case константное_целое_выражение2:]
 [операторы]
  . . .
 [case константное_целое_выражение  m:]

  . . .
 [case константное_целое_выражение  n:]
 [операторы]
 [default:] [операторы] }
```

Здесь [ ] означают необязательную часть оператора, а **...** говорит о том, что указанная конструкция может применяться сколько угодно раз. Блок после **switch**( ) называют телом оператора **switch**.

Схема выполнения оператора:

Сначала вычисляется выражение в круглых скобках (назовем его селектором).

Затем вычисленное значение селектора последовательно сравнивается с константным выражением, следующим за **case**.

Если селектор равен какому-либо константному выражению, стоящему за **case**, то управление передается оператору, помеченному соответствующим оператором **case**.

Если селектор не совпадает ни с одной меткой варианта, то управление передается на оператор, помеченный словом **default**.

Если **default** отсутствует, то управление передается следующему за **switch** оператору.

Отметим, что после передачи управления по какой-либо одной из меток дальнейшие операторы выполняются подряд. Поэтому, если необходимо выполнить только часть из них, нужно позаботиться о вы-

ходе из **switch**. Это обычно делается с помощью оператора **break**, который осуществляет немедленный выход из тела оператора **switch**.

Пример 1:

```
int i, d;
cout<<"Задайте целое значение  i\n";
cin>>i;
switch ( i ){
case 1: case2: case3: cout<<" i="<< i <<"\n";
case 4: cout<<" i="<< i <<"  i^2= "<<i*i<<"\n";
d=3*i – 4;  cout<<" d=" << d <<".\n";
break;
case 5: cout<<"i=5.\n";  break;
default: cout<<" Значение  i меньше  1  или  больше  5.\n";
}
```

Если ввести число 2, то будет напечатано

```
i=2
i=2  i^2=4
d=2.
```

Если **i** равно 4 , то будет выведено

```
i=4  i^2=16
d=8.
```

При **i**=5 будет выведено

i=5.

При всех остальных значениях **i** будет напечатано

```
Значение  i  меньше 1 или больше 5.
```

Пример 2:

```
char sign;
int  x, y, z;
cout<<"Задайте знак операции  +    –    *    / \n";
cin>>sign;
cout<<"Задайте  x  и  y \n";
```

```
cin>>x>>y;
switch (sign){
case '+':  z= x + y;          break;
case '–':  z= x – y;          break;
case '*':  z= x * y;          break;
case '/':  if ( y = = 0 ){cout<<"Делить на нуль нельзя!\n"; exit(1);}
           else  z = x / y;    break;
default:  cout<<"Неизвестная операция!\n"; exit(1);
}
```

Здесь exit(1) – вызов функции, который приводит к немедленному прекращению выполнения программы.

4.7. Оператор цикла *while*

Оператор цикла с **предусловием** имеет вид

while (**выражение**) **оператор**

Оператор называют телом цикла.

При выполнении такого оператора сначала вычисляется значение **выражения**. Если оно равно 0, то **оператор** не выполняется и управление передается оператору, следующему за ним. Если значение **выражения** отлично от 0, то выполняется **оператор**, затем снова вычисляется **выражение** и т.д.

Возможно, что тело цикла не выполнится ни разу, если **выражение** сразу будет равно 0.

Пример 1:

```
char c;
while ( cin.get(c) ) cout<<c;
```

Здесь происходит копирование символов, включая пробельные, из потока **cin** (в данном случае из буфера клавиатуры) в поток **cout** (в данном случае на экран дисплея). Здесь функция **get(c)** (член класса) извлекает один символ из входного потока, включая пробельные символы. Возвращает ненулевое значение до тех пор, пока не достигнет конца файла (признак конца файла – ctrl-z).

Пример 2:

```
while (1) {  операторы ... }
```

Это – бесконечный цикл.

Пример 3:

```
char c;
while (( c = cin.get (c)) = = ‘  ‘ || c = = ‘\n‘ || c = = ‘\t’);
```

Этот оператор цикла пропускает при считывании из потока **cin** так называемые пробельные символы. Здесь **get( )** – другая форма функции, считывающей из потока один символ. Она возвращает целое число – код символа, или число -1, если встретился признак конца файла.

4.8. Цикл с постусловием *do-while*

Этот оператор цикла проверяет условие окончания в конце, после каждого прохода через тело цикла; поэтому тело цикла всегда выполняется по крайней мере один раз.

Вид оператора:

do **оператор** while (**выражение**)

Сначала выполняется **оператор**, затем вычисляется выражение и, если оно отлично от нуля, то оператор выполняется снова и т.д.

Если выражение становится равно нулю, цикл завершается.

Такой цикл удобно, например, использовать при проверке вводимых пользователем данных:

```
int  input=0;
int  minvalue=10,  maxvalue=150;
  do{ cout <<“Введите значение input \n”;   cin >>input;
  cout <<“ input=“ << input  << “\n”; }
while( input < minvalue || input > maxvalue );
```

4.9. Оператор *for*

Этот оператор цикла имеет вид:

for(**оператор1 выражение1; выражение2**) **оператор2**

Оператор1 может быть объявлением, пустым оператором или оператором-выражением.

Наиболее распространенным является случай, когда **оператор1** и **выражение2** являются присваиваниями или обращениями к функциям, а **выражение1** – условным выражением. Этот цикл эквивалентен следующей конструкции:

оператор1
while (**выражение1**){ **оператор2 выражение2**; }

Иногда **оператор1** называют инициализатором цикла, а **выражение2** – реинициализатором.

Любая из трех частей может быть опущена, хотя точка с запятой обязательно должна оставаться. Если отсутствует проверка, то есть **выражение1**, то считается, как будто **выражение1** отлично от 0, так что

for(; ;){ . . . } – бесконечный цикл и его надо каким-либо образом прервать.

Пример1:

```
int n=20, s=0;
for( int i = 1; i <= n; i++ ) s+ = i*i;
```

Здесь вычисляется сумма квадратов целых чисел от 1 до 20.

Пример2:

```
double s, sum, den = 0.85, eps = 1e-10;
for( s=1, sum=0; s>eps;  s*=den ) sum+=s;
```

Здесь вычисляется сумма геометрической прогрессии 1, 1*0.85, 1*0.85*0.85 и т.д., пока ее очередной член не станет меньше 10^{-10}.

В следующем примере вычислим и выведем на экран дисплея таблицу функции $y = \sin(x^2)$ для $x \in [0, \pi/2]$ с шагом $\pi/20$.

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

void main(){
int n = 10;
double x0 = 0, xk = M_PI_2, y,
h = (xk – x0)/n,  xt = xk + h/2;
clrscr();
cout<<"    x            y\n"
for(double x = x0; x < xt; x+ = h){
y = sin(x*x);
cout.width(4);       cout.precision(2);
cout<<x;
cout.width(10);      cout.precision(4);
cout<<y<<'\n';
}
}
```

В этой программе используется константа M_PI_2, представляющая значение $\pi / 2$ и определенная в заголовочном файле cmath.h (для использования таких констант в visual c++ нужно написать

```
#define _USE_MATH_DEFINES
#include <cmath>).
```

Обращение к функции cout.width(**k**) устанавливает ширину поля следующего вывода в **k** позиций, что позволяет выровнять вид таблицы. Функция cout.precision(**k**) задает число цифр, выводимых после десятичной точки. Функция clrscr(), прототип которой находится в conio.h, очищает экран.

Использовать ли циклы while или for – это, в основном дело вкуса. Цикл for предпочтительнее там, где имеется простая инициализация и реинициализация, поскольку при этом управляющие циклом операторы наглядным образом оказываются вместе в начале цикла. Это наиболее очевидно в конструкции

```
for(i = 0; i < n; i ++),
```

которая применяется для обработки первых n элементов массива, аналогично оператору цикла for Паскаля. Аналогия, однако, не полная, так как границы цикла могут быть изменены внутри цикла, а понятие управляющей переменной в C++ отсутствует и переменная i сохраняет свое значение после выхода из цикла, какова бы ни была причина этого выхода.

4.10. Оператор безусловного перехода

Оператор безусловного перехода имеет вид **goto метка**;

Метка – это имя, за которым следует '**:**'. Этот оператор передает управление оператору, помеченному указанной меткой. С его помощью удобно выходить сразу из нескольких вложенных циклов:

```
for ( i = 0; i < n;  i++)
for ( j = 0; j < m; j++)
for ( k = 0;k < l;  k++) {
…
операторы;
…
if ( условие ) goto  lab;
операторы;}
lab:; . . .
```

С помощью оператора **goto** можно переходить извне в тело блока, если при этом управление не передается через объявления имен, которые присутствуют в этом блоке.

4.11. Оператор *break*

Этот оператор осуществляет выход из тела цикла **for**, **while**, **do-while** или оператора **switch**, в котором он появился. При этом управление передается на первый оператор после цикла.

Оператор не может обеспечить выход сразу из двух или более вложенных циклов.

4.12. Оператор *continue*

Этот оператор осуществляет переход на точку сразу за последним оператором тела цикла без выхода из цикла, так что дальнейшие итерации в цикле будут продолжаться.

Пример вывода четных чисел:

```
for ( int num = 0; num < 100; num++ ){
if ( num  % 2 ) continue;
cout << num << “\n“;
}
```

Когда **num** становится нечетным, выражение num % 2 получает значение 1 и выполняется оператор, который передает управление на следующую итерацию цикла **for**, не выполняя вывода.

4.13. Оператор *return*

Этот оператор завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию.

Управление передается в вызывающую функцию в точку, непосредственно следующую за вызовом.

Если **return** присутствует в функции main( ), то он вызывает прерывание выполнения программы.

5. Указатели

5.1. Определение указателей

Указатель – это переменная, содержащая адрес некоторого объекта, например, другой переменной. Точнее – адрес первого байта этого объекта. Это дает возможность косвенного доступа к этому объекту через указатель. Пусть **x** – переменная типа **int**. Обозначим через **px** указатель. Унарная операция & выдает адрес объекта, так что оператор

```
px = &x;
```

присваивает переменной **px** адрес переменной **x**. Говорят, что **px** “указывает” на **x**. Операция & применима только к адресным выражениям, так что конструкции вида &(**x**-1) и &3 незаконны.

Унарная операция * называется операцией разадресации или операцией разрешения адреса. Эта операция рассматривает свой операнд как адрес и обращается по этому адресу, чтобы извлечь объект, содержащийся по этому адресу.

Следовательно, если **y** тоже имеет тип **int**, то

```
y = *px;
```

присваивает **y** содержимое того, на что указывает **px**. Так, последовательность

```
px = &x;
y = *px;
```

присваивает **y** то же самое значение, что и оператор

```
y = x;
```

Все эти переменные должны быть описаны:

```
int x, y;
int *px;
```

Последнее – описание указателя. Его можно рассматривать как мнемоническое. Оно говорит, что комбинация ***px** имеет тип **int** или, иначе, **px** есть указатель на **int**. Это означает, что если **px** появляется в виде ***px**, то это эквивалентно переменной типа **int**.

Из описания указателя следует, что он может указывать только на определенный вид объекта (в данном случае **int**). Разадресованный ука-

затель может входить в любые выражения там, где может появиться объект того типа, на который этот указатель ссылается. Так, оператор

y = *px + 2;

присваивает **y** значение, на **2** больше, чем **x**.

Заметим, что приоритет унарных операций * и & таков, что эти операции связаны со своими операндами более крепко, чем арифметические операции, так что выражение

y = *px + 2

берет то значение, на которое указывает **px**, прибавляет 2 и присваивает результат переменной **y**.

Если **px** указывает на **x**, то

*px = 3;

полагает **x** равным 3, а

*px + = 1;

увеличивает **x** на 1 так же, как и выражение

(*px) ++

Круглые скобки здесь необходимы. Если их опустить, то есть написать ***px** ++, то, поскольку унарные операции, подобные * и ++, выполняются справа – налево, это выражение увеличит **px**, а не ту переменную, на которую он указывает.

Если **py** – другой указатель на **int**, то можно выполнить присвоение

py = px;

Здесь адрес из **px** копируется в **py**. В результате **py** указывает на то же, что и **px**.

5.2. Указатели и массивы

Массив – это совокупность элементов одного типа, которые расположены в памяти ЭВМ подряд, один за другим.

Признаком объявления массива являются квадратные скобки. Объявить массив из 10 элементов типа **double** можно так:

double a[10];

Чтобы обратиться к элементу этого массива, нужно применить операцию индексирования **a**[**ind**]. Внутри квадратных скобок помещается целое выражение, которое называется индексом. Нумерация эле-

ментов массива **начинается с 0** и поэтому вышеприведенное описание говорит о том, что в памяти ЭВМ зарезервировано место под 10 переменных типа **double** и эти переменные есть **a**[0], **a**[1], . . . , **a**[9].

Приведем пример с использованием массива.

Напишем программу подсчета числа появлений каждой цифры, пробельных символов и всех остальных символов.

Число пробельных символов будем хранить в **nwhite**, прочих символов – в **nother**, а число появлений каждой из цифр – в массиве **ndigit**:

```
void main( ) {
int c, i, nwhite = 0, nother = 0;
int ndigit [10];
for ( i=0; i<10; i++) ndigit[i]=0;
while ( ( c=cin.get( ) )!=EOF)
if(c>=‘0’ &&  c<=‘9’) ++ ndigit[c – ’0’];
else if (c = = ‘ ‘|| c = = ‘\n‘ || c = = ‘\t‘) ++ nwhite;
else ++ nother;
cout<<“ цифра \n”;
for( i=0; i<10; i ++)
cout<<i<<“  вошла”<<ndigit[i]<<“  раз \n”;
cout<<“ пробельных символов – “
<< nwhite <<“ прочих символов – ”<< nother <<“\n”;
}
```

При объявлении массива его можно инициализировать:

```
int c[ ] = { 1, 2, 7, 0, 3, 5, 5 };
char array[ ] = { ’h’, ’e’, ’l’, ’l’, ’o’, ’\n’, ’\0’};
```

Последнюю инициализацию разрешается выполнять проще:

```
char array[ ] = “hello\n”;
```

Такой синтаксис инициализации разрешен только для строк. Компилятор сам вычислит необходимый размер памяти с учетом автоматически добавляемого в конец строки символа ’\0’ с кодом 0, который является признаком завершения строки.

В языке C++ имя массива является **константным указателем** на первый элемент этого массива:

```
int mas[20];
int *pmas;
pmas = &mas[0];
```

Последний оператор можно записать и так: pmas = mas;

Операция индексирования массива [ ] имеет 2 операнда – имя массива, т.е. указатель, и индекс, т.е. целое: **a**[**i**]. В языке C++ любое выражение **указатель[индекс]** по определению трактуется как

∗(**указатель** + **индекс**)

и автоматически преобразуется к такому виду компилятором.

Таким образом, **a**[**3**] эквивалентно **∗**(**a** + **3**). Более того, это можно записать даже так **3**[**a**], так как это все равно будет проинтерпретировано как **∗**(**3**+**a**). Здесь складываются указатель **a** и целое **3**. В связи с этим рассмотрим так называемую адресную арифметику.

5.3. Адресная арифметика

Указатель можно складывать с целым.

Если к указателю **pa** прибавляется целое приращение **i**, то приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель **pa**.

Таким образом, **pa+i** – это адрес **i**-го элемента после **pa**, причем считается, что размер всех этих **i** элементов равен размеру объекта, на который указывает **pa**.

Итак, если **a** – массив, то

a+i – адрес i-го элемента этого массива, т.е.

&a[i] равен a+i и a[i] равняется ∗(a+i).

```
float b[10];
float *pb=b;
pb++;  // Это эквивалентно  pb=pb+1.
// Здесь указатель pb будет указывать на элемент массива b[1].
pb+=3; // Здесь  pb  указывает на элемент массива b[4].
```

Отметим, что нельзя написать **b**++ или **b = b+i**, так как имя массива **b** – это константный указатель и его изменять нельзя.

Указатели можно сравнивать.

Если **p** и **q** указывают на элементы одного и того же массива, то такие отношения, как < >= и т.д. работают надлежащим образом. Например,

$p < q$

истинно, т.е. равно 1, если **p** указывает на более ранний элемент массива, чем **q**. Любой указатель можно сравнить на равенство или неравенство с так называемым нулевым указателем **NULL**, который ни на что не указывает. Однако не рекомендуется сравнивать указатели, указывающие на различные массивы.

Указатели можно вычитать.

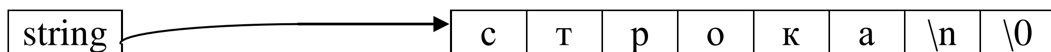
Если **p** и **q** указывают на элементы одного и того же массива, то **p – q** дает количество элементов массива между **p** и **q**.

5.4. Символьные массивы и строки

Строка является массивом символов. Значением строки является указатель на первый ее символ:

```
char ∗string = (char*)“строка\n”;
```

Здесь указатель на символы **string** будет содержать адрес первого символа ‘с’ строки “строка\n”, которая размещается в некоторой области памяти, начиная с этого адреса:



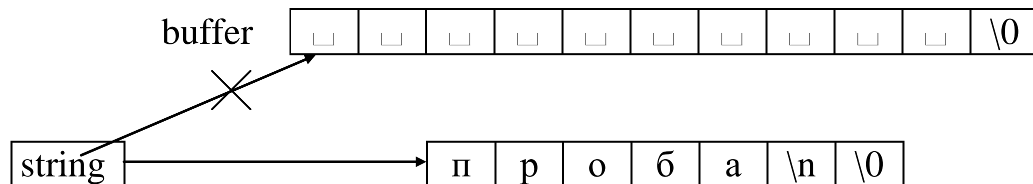
Здесь **string**[3] = = ’о’.

Рассмотрим фрагмент программы:

```
char buffer[ ]=“␣␣␣␣␣␣␣␣␣␣”;           // Инициализация
                                       // строки из 10 пробелов.
char *string = buffer;          // string  указывает на начало буфера.
string=(char*)“проба\n”;      // Присваивание!
```

При инициализации создается строка **buffer** и в нее помещаются символы (здесь 10 пробелов). Инициализация char *string=buffer устанавливает указатель **string** на начало этой строки.

Операция же присваивания в последней строке не копирует приведенную строку “проба\n” в массив **buffer**, а изменяет значение указателя **string** так, что он начинает указывать на строку “проба\n”:



Чтобы скопировать строку “**проба\n**” в **buffer**, можно поступить так:

```
char buffer[ ] = “␣␣␣␣␣␣␣␣␣␣”;
char ∗p =(char*)“проба\n”;
```

```
int i =0;
while ( ( buffer[i] = p[i] ) != '\0' ) i++;
```

Или так:

```
char buffer[ ] = "␣␣␣␣␣␣␣␣␣␣␣"
char * p = (char*)"проба\n";
char *buf = buffer;
while (*buf ++ = *p ++ );
```

Здесь сначала ***p** копируется в ***buf**, т.е. символ 'п' копируется по адресу **buf**, который совпадает с адресом **buffer**, т.е. **buffer**[0] становится равен 'п'. Затем происходит увеличение указателей **p** и **buf**, что приводит к продвижению по строкам "**проба\n**" и **buffer** соответственно. Последний скопированный символ будет '\0', его значение – 0 и оператор **while** прекратит цикл.

Еще проще воспользоваться библиотечной функцией, прототип которой находится в файле **string.h**:

```
strcpy( buffer, "проба\n" );
```

При копировании необходимо обеспечить, чтобы размер памяти, выделенной под **buffer**, был достаточен для хранения копируемой строки.

5.5. Многомерные массивы

Двумерный массив рассматриваются как массив элементов, каждый из которых является одномерным массивом. Трехмерный – как массив, элементами которого являются двумерные массивы и т.д.

После объявления

```
int a[5][6][7];
```

в программе могут появиться выражения:

a[i][j][j]	– объект типа int;
a[2][0]	– объект типа int* – одномерный массив из 7 целых;
a[1]	– двумерный массив из 6*7 = 42 целых;
a	– сам трехмерный массив.

Так как элементом массива **a** является двумерный подмассив размером 6*7, то при выполнении выражения **a + 1** происходит смещение на величину элемента массива **a**, т.е. переход от **a**[0] к **a**[1]. Значение адреса при этом увеличивается на 6*7*sizeof(int) =84.

Для двумерного массива **mas** выражение **mas**[**i**][**j**] интерпретируется как *(*(**mas**+**i**)+**j**). Здесь **mas**[**i**] – константный указатель на **i**-ю строку массива **mas**.

В памяти массивы хранятся по строкам, т.е. при обращении к элементам в порядке их размещения в памяти быстрее всего меняется самый правый индекс.

Так, для массива c[2][3] его шесть элементов расположены в памяти так:

c[0][0] c[0][1] c[0][2] c[1][0] c[1][1] c[1][2].

Многомерные массивы также можно инициализировать при описании:

```
int d[2][3]={ 1, 2, 0, 5 };
```

В этом случае первые 4 элемента массива получат указанные значения, а остальные два инициализируются нулями.

Если инициализируется многомерный массив, то самую первую размерность можно не задавать. В этом случае компилятор сам вычисляет размер массива:

```
int f [ ][2] = { 2, 4, 6, 1 };                  // массив  f [2][2];
int a [ ][2][2] = { 1, 2, 3, 4, 5, 6, 7, 8 };   // массив  a [2][2][2].
```

Инициализирующее выражение может иметь вид, отражающий факт, что массив является, например, двумерным:

```
int c[2][3]={ {1, 7}, {-5, 3} };
```

В этом случае в матрице **c** инициализированы нулевой и первой столбцы, а второй столбец, то есть элементы c[0][2] и c[1][2], – инициализируется нулями.

5.6. Указатели и многомерные массивы

Рассмотрим разницу между объектами **a** и **b**, описанными следующим образом:

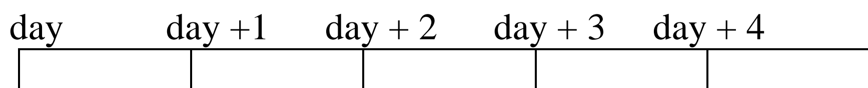
```
int a[10][10];
int * b[10];
```

И **a** и **b** можно использовать сходным образом в том смысле, что как **a**[5][5], так и **b**[5][5] являются обращениями к отдельному значению типа **int**. Но **a** – настоящий массив: под него отводится 100 ячеек памяти и для нахождения любого указанного элемента проводятся обычные вычисления с индексами, которые требуют умножения. Для **b** описание выделяет только 10 указателей. Каждый из них должен быть установлен так, чтобы он указывал на массив целых.

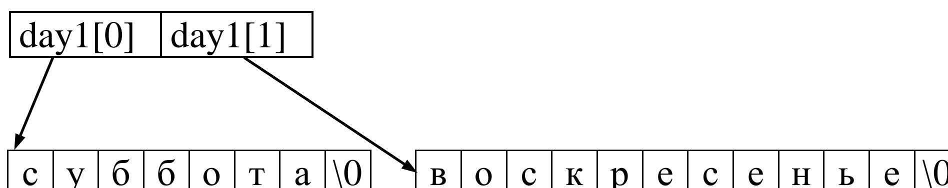
Если предположить, что каждый из них указывает на массив из 10 элементов, то тогда где-то будет отведено 100 ячеек памяти плюс еще 10 ячеек для указателей. Таким образом, массив указателей использует несколько больший объем памяти и может требовать наличие явного шага инициализации. Но при этом возникают 2 преимущества: доступ к элементу осуществляется косвенно через указатель, а не посредством умножения и сложения, и строки массива могут иметь различные длины. Это означает, что каждый элемент **b** не должен обязательно указывать на вектор из 10 элементов. Эту разницу можно увидеть в следующем примере:

```
char day [5][12] = {// В каждой строке 12 символов.
“понедельник”, “вторник”, “среда”, “четверг”, “пятница”};
```

Здесь константные указатели day[0], day[1], …, day[4] адресуют участки памяти одинаковой длины 12 байт каждый:




```
const char * day1[2] =
{"суббота",              // 7  символов + ‘/0’
  "воскресенье"          // 11 символов + ‘/0’
};
```



Здесь переменные-указатели day1[0] и day1[1] адресуют участки памяти соответственно в 8 и 12 байт.

6. Операция sizeof

Эта операция выполняется на стадии компиляции. Результатом этой операции является число байтов, необходимое для размещения объекта в памяти. Существует два варианта синтаксиса этой операции. В первом из них единственный операнд операции определяет некоторый тип языка, и он должен быть заключен в скобки:

```
sizeof ( float )
sizeof ( int )
```

Во втором операнд задает некоторое выражение и здесь использование скобок необязательно:

```
sizeof a;
sizeof *ip;
sizeof array[ i ];
```

Заметим, что при получении размеров массивов, несмотря на то, что имя массива является указателем, результатом операции

```
sizeof array,
```

где array – имя некоторого массива, является длина в байтах этого массива. Это свойство можно использовать для вычисления числа элементов в массиве:

```
const int n = 20;
int array [n];
. . .
int num=sizeof array / sizeof(int)        // num равно 20.
```

Результатом операции **sizeof** над ссылкой является длина типа, с которым сопоставлена ссылка, т.е. sizeof(double &) и sizeof(double) эквивалентны.

7. Операции для работы с динамической памятью

7.1. Операция выделения памяти *new*

Часто выражение, содержащее операцию new, имеет следующий вид:

указатель_на_тип_= new имя_типа (инициализатор)

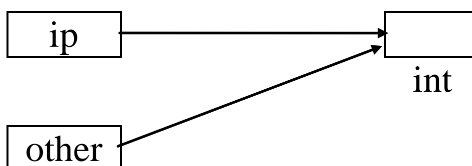
Инициализатор – это необязательное инициализирующее выражение, которое может использоваться для всех типов, кроме массивов.

При выполнении оператора

```
int *ip = new int;
```

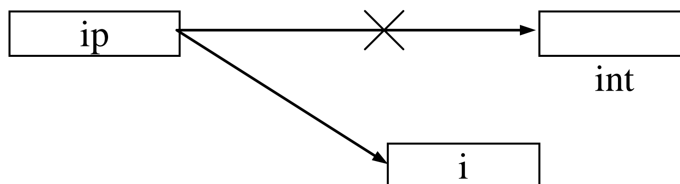
создаются 2 объекта: динамический безымянный объект и указатель на него с именем ip, значением которого является адрес динамического объекта. Можно создать и другой указатель на тот же динамический объект:

```
int *other=ip;
```



Если указателю ip присвоить другое значение, то можно потерять доступ к динамическому объекту:

```
int *ip=new (int);
int  i=0;
ip=&i;
```



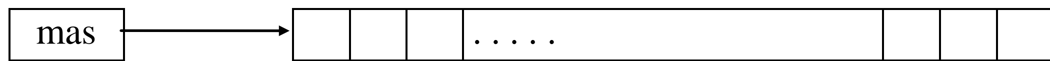
Теперь динамический объект по-прежнему будет существовать, но обратится к нему уже нельзя. Такие объекты называются мусором.

При выделении памяти объект можно инициализировать:

```
int *ip = new int(3);
```

Можно динамически распределить память и под массив:

```
double *mas = new double [50];
```



Теперь с этой динамически выделенной памятью можно работать как с обычным массивом:

```
*(mas+5) = 3.27;
mas[6] = mas[5] + sin(mas[5]);
```

В случае успешного завершения операция new возвращает указатель со значением, отличным от нуля.

Результат операции, равный 0, т.е. нулевому указателю NULL, говорит о том, что не найден непрерывный свободный фрагмент памяти нужного размера.

7.2. Операция освобождения памяти *delete*

Операция **delete** освобождает для дальнейшего использования в программе участок памяти, ранее выделенной операцией **new**:

```
delete ip;      // Удаляет динамический объект типа int,
                // если было ip = new int;
delete mas;  // удаляет динамический массив длиной  50, если было
                // double *mas = new double[50];
```

Совершенно безопасно применять операцию к указателю NULL. Результат же повторного применения операции delete к одному и тому же указателю не определен. Обычно происходит ошибка, приводящая к зацикливанию.

Чтобы избежать подобных ошибок, можно применять следующую конструкцию:

```
int *ip=new int[500];
. . .
if (ip){delete ip; ip=NULL;}
else { cout <<“ память уже освобождена \n”; }
```

Пример:

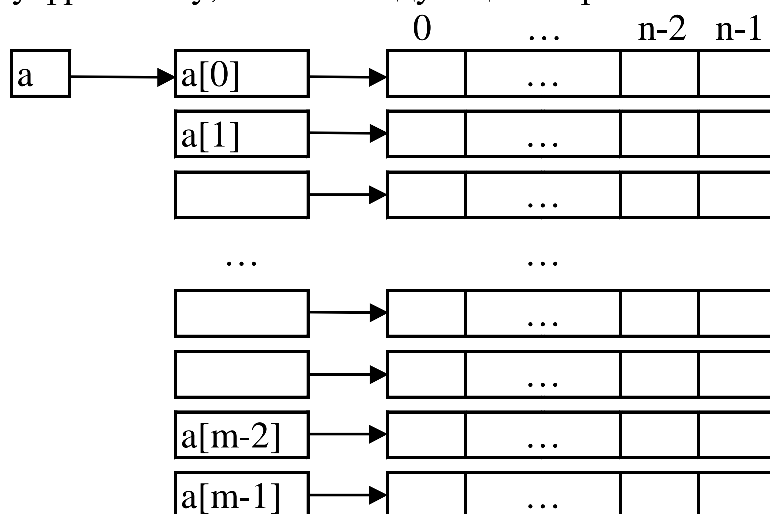
Распределить память для матрицы из m строк и n столбцов:

```
int m, n;
cout<<"Задайте число строк и столбцов матрицы: \n";
cin>>m>>n;
double **a = new double *[m];  // массив из m указателей на double
for (int i = 0; i<m; i++ )
if ((a[i] = new double[n])==NULL) // Распределяется строка матри-
цы
{ cout<<"Нет памяти!\n";       exit(1);}
```

Теперь к элементам этой матрицы можно обращаться обычным образом:

a[i][j] или *(a[i] + j)   или *(*(a + i) + j)

Изобразить распределение памяти, соответствующее вышеприведенному фрагменту, можно следующим образом:



Освободить память здесь можно так:

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

Или так:

```
for(i=0; i<m; i++) {delete a[ i ]; a[ i ]=NULL;}
delete a;
```

8. Объявления и определения

Любое имя, за исключением имён меток, должно быть объявлено в программе:

```
int i, j;
double d=7.3;
extern  int  m;
typedef  unsigned  int  size_t;
int  add (int a, int b){ return  a+b;}
void func (char*, int);
```

После этих объявлений компилятор знает, что i, j, m – имена переменных типа int, d – имя переменного типа double, size_t – имя типа, а add и func – имена функций (о функциях см. раздел 16).

В объявлении с именем может сопоставляться не только некоторый тип. С именем может быть сопоставлен некоторый элемент, идентификатором которого оно является. Например, при объявлении int i, j; для переменных i, j отводится память, с этими именами сопоставляются участки памяти по 2 байта, которые можно использовать для хранения значений переменных. С именем size_t сопоставлено имя конкретного типа (unsigned int), синонимом которого является теперь size_t; с именем add сопоставлен код функции. Все такие объявления называют определениями. Однако не все объявления являются определениями. Из всех вышеприведённых объявлений два не являются определениями:

```
extern  int  m;
void func (char *, int);
```

В первом из этих объявлений сообщается только, что переменная m имеет тип int, память же для неё должна выделяться где-то в другом месте, о чём говорит ключевое слово extern.

Второе объявление говорит, что func есть имя функции с двумя аргументами, первый из которых – указатель на char, а второй – int, а сама функция не возвращает никакого значения. Такое объявление называют **прототипом** функции. Сама же функция func должна быть определена вместе со своим телом где-то в другом месте.

В программе для каждого имени должно присутствовать одно и только одно определение, в то время как объявлений, не являющихся определениями, может быть как угодно много.

9. Область существования имени

После объявления имя можно использовать. Однако оно, как правило, может быть использовано только в некоторой части программы, которая называется областью существования имени.

Текст программы можно разместить в одном файле, а можно и в нескольких различных файлах, каждый из которых содержит целиком одну или несколько функций. Для объединения в одну программу эти файлы компилируются совместно. Информация обо всех этих файлах помещается в так называемый файл проекта (с расширением ".vcxproj"). Компилятор для каждого исходного файла создаёт объектный код (файл с расширением ".obj"). Затем все объектные файлы (вместе с библиотечными) объединяются компоновщиком в исполняемый, или загрузочный модуль, который имеет имя файла проекта и расширение ".exe". Область существования имени нужна компилятору для того, чтобы сгенерировать верный машинный код.

9.1. Компиляция, компоновка, библиотеки

Рассмотрим в связи с этим несколько подробнее саму технологию подготовки программ.

Изобразить ее схематически можно так (рис. 1):

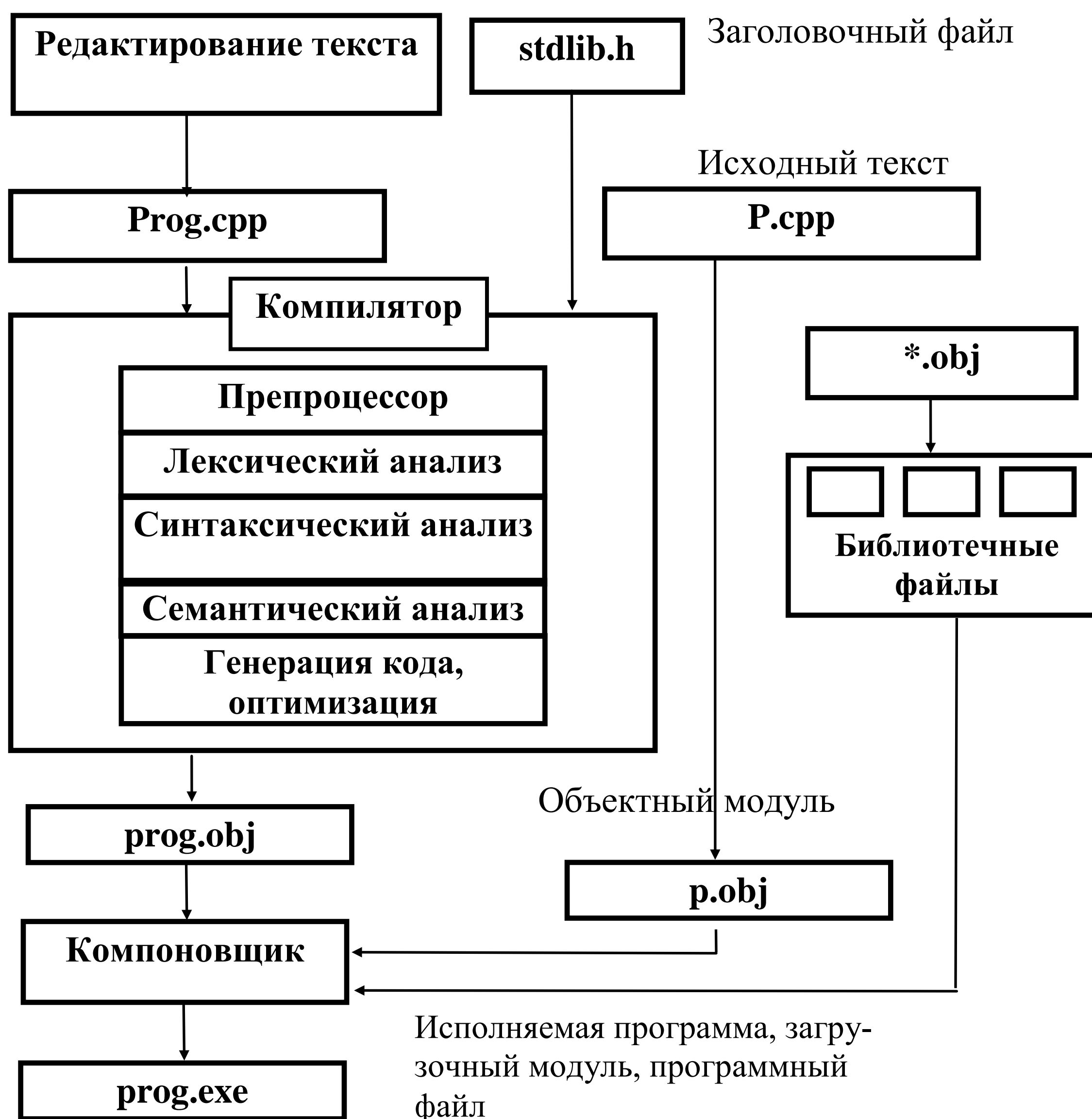


Рис. 1. Подготовка программы к выполнению

Подготовка программы начинается с редактирования файла, содержащего текст этой программы, который имеет стандартное расширение ".cpp". Затем выполняется его компиляция, которая включает в себя несколько фаз: препроцессор, лексический, синтаксический, семантический анализ, генерация кода и его оптимизация. В результате компиляции получается объектный модуль – некий "полуфабрикат" готовой программы, который потом участвует в ее сборке. Файл объектного модуля имеет стандартное расширение ".obj". Компоновка (сборка) программы заключается в объединении одного или нескольких объект-

ных модулей программы и объектных модулей, взятых из библиотечных файлов и содержащих стандартные функции и другие полезные вещи. В результате получается исполняемая программа в виде отдельного файла (загрузочный модуль, программный файл) со стандартным расширением -".exe", который затем загружается в память и выполняется.

9.1.1. Компиляция и ее фазы

Собственно компиляция начинается с лексического анализа программы. *ЛЕКСИКА* языка программирования – это правила "правописания слов" программы таких, как идентификатры, константы, служебные слова, комментарии. Лексический анализ разбивает текст программы на указанные элементы. Особенность любой лексики – ее элементы представляют собой регулярные линейные последовательности символов. Например, ***ИДЕНТИФИКАТОР*** –это произвольная последовательность букв, цифр и символа '_', начинающаяся с буквы или '_'.

СИНТАКСИС языка программирования – это правила составления предложений языка из отдельных слов. Такими предложениями являются операции, операторы, определения функций и переменных. Особенностью синтаксиса является принцип вложенности (рекурсивность) правил построения предложений. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частным случаем которого является все тот же оператор цикла.

СЕМАНТИКА языка программирования – это смысл, который закладывается в каждую конструкцию языка. Семантический анализ – это проверка смысловой правильности конструкции. Например, если мы в выражении используем переменную, то она должна быть определена ранее по тексту программы, а из этого определения может быть получен ее тип. Исходя из типа переменной, можно говорить о допустимости операции с данной переменной.

ГЕНЕРАЦИЯ КОДА – это преобразование элементарных действий, полученных в результате лексического, синтаксического и семантического анализа программы, в некоторое внутреннее представление. Это

могут быть коды команд, адреса и содержимое памяти данных, текст программы на языке Ассемблера либо стандартизованный промежуточный код (например, так называемый P-код). В процессе генерации кода производится и его оптимизация.

9.1.2. Модульное программирование, компоновка

Полученный в результате трансляции ***ОБЪЕКТНЫЙ МОДУЛЬ*** включает в себя готовые к выполнению коды команд, адреса и содержимое памяти данных. Но это касается только собственных внутренних объектов программы (функций и переменных). Обращение к внешним функциям и переменным, отсутствующим в данном фрагменте программы, не может быть полностью переведено во внутреннее представление и остается в объектном модуле в исходном (текстовом) виде. Но если эти функции и переменные отсутствуют, значит, они должны быть каким-то образом получены в других объектных модулях. Самый естественный способ – написать их на том же самом C++ и откомпилировать. Это и есть принцип ***МОДУЛЬНОГО ПРОГРАММИРОВАНИЯ*** – представление текста программы в виде нескольких файлов, каждый из которых транслируется отдельно. С модульным программированием мы сталкиваемся в двух случаях:

- когда сами пишем модульную программу;
- когда используем стандартные библиотечные функции.

БИБЛИОТЕКА ОБЪЕКТНЫХ МОДУЛЕЙ – это файл (библиотечный файл), содержащий набор объектных модулей и собственный внутренний каталог. Объектные модули библиотеки извлекаются из нее целиком при наличии в них требуемых внешних функций и переменных и используются в процессе компоновки программы.

КОМПОНОВКА – это процесс сборки программы из объектных модулей, в котором производится их объединение в исполняемую программу и связывание вызовов внешних функций и их внутреннего представления (кодов), расположенных в различных объектных модулях. Этот этап выполняет специальная программа – LINKER.

9.2. функция,

Вернемся к понятию "область существования имени". Можно выделить 5 видов областей существования имени.

Область существования ***БЛОК.*** Напомним, что блок – это фрагмент программы, заключённый в фигурные скобки { }, например

```
if (a != 5) {
int j=0;
double  k = 3.5;
a++;
…
}
```

Заметим, что тело любой функции является блоком.

Имя, объявленное в блоке, может быть использовано от точки, где находится его объявление, и до конца блока. Такую же область существования имеют и имена в определении функции:

```
int  f1 (int i){ return i; }
```

Имя i имеет область существования «блок». Область существования «блок» распространяется и на вложенные блоки.

Область существования ***ФУНКЦИЯ***. Эту область существования имеют только имена меток перехода, используемые оператором goto:

```
void f (){….
…
goto lab;
…
{… lab: ...}
…
}
```

Область существования ***ПРОТОТИП ФУНКЦИИ***. Прототип функции есть объявление функции, не являющееся её определением и имеющий, например, вид

```
int F(int a, double b, char* str);
```

Область существования «прототип» заключена между открывающей и закрывающей круглыми скобками. Иначе говоря, имена a, b, str в примере определены только внутри круглых скобок. Из этого следует, что в прототипах можно использовать для аргументов любые имена или не использовать их совсем:

```
int F(int, double, char*);
```

Область существования ***ФАЙЛ***. Область существования «файл» имеют имена, объявленные вне любого блока и класса. Такие имена называют глобальными. Глобальные имена определены от точки их объявления и до конца файла, где встретилось их объявление. Примером таких имён являются имена функций.

```
#include <iostream>
int a, b, c[40];          // Глобальные имена;
int f1()                  // локальное имя f1;
{int i;                   // локальное имя;
…
}
int count;                // глобальное имя;
void f2() { … }           // глобальное имя f2.
```

Область существования ***КЛАСС***.

Такую область существования имеют имена, объявленные в классах (см. раздел 18). Эти имена определены во всем классе, в котором они объявлены, независимо от точки их объявления.

10. Область видимости

Если, используя имя, можно получить доступ к элементу, с которым это имя сопоставлено, то говорят, что данное имя находится в области своей видимости.

Область видимости является подобластью области существования имени. Если элемент языка, чьё имя находится в области своего существования, тем не менее, недоступен по этому имени, то будем говорить, что это имя **скрыто** или **замаскировано**.

Глобальные имена видимы от точки их объявления до конца файла, если они не замаскированы локальными именами.

Переменные из объемлющих блоков, как и глобальные, видимы во внутренних блоках.

Если переменная, объявленная внутри блока, имеет то же имя, что и переменная объемлющего уровня, то имя объемлющего уровня маскируется, и определение переменной в блоке заменяет определение объемлющего уровня на протяжении всего блока. Видимость замаскированной переменной восстанавливается при выходе из блока. Метки в функции видимы во всём теле функции.

```
int i=3;
{ int c = i;            // c становится равно 3;
…
int i = 0;              // имя i маскирует внешнее имя i;
cout <<”c = ”<< c <<”,   i= ”<< i <<”.\n”;
}                       // конец области существования имен i, c из блока;
                        // опять видно имя i, объявленное перед блоком.
cout <<” i = ”<< i <<”.\n”;
…
```

Здесь будет выведено

```
c=3,  i=0.
i=3.
```

Если скрытым именем является глобальное имя функции или объекта, то обратиться к нему можно, используя операцию **разрешения доступа**, или **разрешения контекста ::**.

```
int i=5;                // Глобальная переменная;

void main (){
int i=1;                // локальная переменная.
i++;
::i++;
cout<<"i="<<i<<",   глобальное i="<<::i<<".\n";
}
```

Здесь будет напечатано

```
i=2,   глобальное i=6.
```

С помощью операции **::** нельзя обратиться к скрытому локальному объекту.

11. Классы памяти

В C++ существуют 3 класса памяти, или хранения.

1) Статическая память – статические данные, размещаемые в сегменте данных;

2) Автоматические данные, размещаемые в специальном стеке (сегмент стека) или как частный случай, в регистрах процессора;

3) Динамические данные, явно размещаемые в динамической памяти с помощью операций **new** и **delete**.

Статические объекты существуют в течение всего времени выполнения программы. К ним относятся глобальные и локальные переменные, объявленные со служебным словом static:

```
int i=3, j;     // глобальные переменные. Класс памяти – static;

void main(){
int a;                          // Автоматическая переменная;
static float b[1000], c=2.3;    // статические переменные;
…
}

int f(){int d;                  // автоматическая переменная;
static int m=2, k;              // статические переменные  m, k.
…
}
```

Статические и глобальные переменные, если они не инициализированы явно, инициализируются нулевыми значениями. В **любом случае**, инициализация статических переменных осуществляется только **один раз**.

Локальные переменные, не объявленные как static, являются автоматическими. Такие объекты начинают свое существование при объявлении его имени в блоке и заканчивают его при завершении этого блока. Если автоматический объект явно не инициализирован, то его значение до присвоения не определено.

```
void f();

void main(){
for (int i = 3; i > 0; i – –) f();
}
void f(){
static int i;
int j = 0;
cout<<”i =”<<i++<<”  j =”<< j++ <<”\n”;
}
```

Здесь будет напечатано:

```
i = 0 j = 0
i = 1 j = 0
i = 2 j = 0
```

Заметим, что если служебное слово static применено к глобальной переменной или к имени функции, то оно имеет другой смысл. В этом случае и глобальная переменная, и функция становятся видимы только в пределах файла, где они определены и невидимы из других файлов.

12. Объявления объектов и типов

При объявлениях можно использовать одновременно более одного модификатора (это * [ ] и ( )). Это даёт возможность создавать бесконечное множество сложных описателей типов. В то же время, некоторые комбинации недопустимы:

элементами массива не могут быть функции;
функции не могут возвращать массив или функцию.

При интерпретации сложных описателей квадратные и круглые скобки (справа от идентификатора) имеют приоритет перед * (слева от идентификатора). Квадратные или круглые скобки имеют один и тот же приоритет.

Спецификатор типа рассматривается на последнем шаге. Можно использовать круглые скобки, чтобы изменить порядок интерпретации на необходимый. Для правильной интерпретации сложных описателей можно придерживаться следующего правила («изнутри – наружу»).

Начать с идентификатора и посмотреть вправо, есть ли квадратные или круглые скобки.

Если они есть, то проинтерпретировать эту часть описателя и затем посмотреть налево в поиске *.

Если на любой стадии справа встретится закрывающая круглая скобка, то вначале необходимо применить все эти правила внутри круглых скобок, а затем продолжить интерпретацию.

Интерпретировать спецификатор типа.

Например, при использовании конструкции

```
int *(*array[10]) ( );
```

объявляется имя array как массив из 10 указателей на функцию без аргументов, возвращающую указатель на целое значение.

13. Имена типов

В некоторых конструкциях языка явно используются имена типов (операции sizeof, new, cast). Имя типа – это просто объявление объекта такого типа, в котором опущено имя самого объекта.

Примеры имён типов:

int – целое,

double – плавающее с двойной точностью,

int * – указатель на целое,

int * [10] – массив указателей на целое,

int (*)[10] – указатель на массив на 10 целых,

int * (void) – функция без аргументов, возвращающая указатель на целое,

int (*) (void) – указатель на функцию без аргументов, возвращающую результат целого типа,

int (*(void))[ ] – функция без аргументов, возвращающая указатель на массив целых.

14. Синоним имени типа

Синоним имени типа строится при помощи ключевого слова **typedef**. Выражение, в котором присутствует это ключевое слово, является описанием некоторого имени. Наличие слова typedef говорит о том, что объявляемый идентификатор становится не именем объекта некоторого типа, а синонимом имени этого типа.

```
int  INTEGER;              // INTEGER – имя переменной типа int
typedef int INT;           // INT – синоним типа int
typedef unsigned size_t;
typedef char string [255];
typedef void (*FPTR) (int);
```

Последние две строки определяют string как синоним типа «массив из 255 символов», а FPTR – синоним типа «указатель на функцию, имеющую один аргумент типа int и не возвращающую никакого результата».

После объявления с помощью typedef новое имя становится полноценным именем типа:

```
string array;              // array – массив из 255 символов
FPTR func_pointer;         // void (*func_pointer)(int);
typedef string STRING;
```

Использование typedef может упростить понимание сложных имен типов. Так, тип int (*(void))[ ] можно построить так:

```
typedef int intarray[];                // тип «массив целых»
typedef intarray * ptrtointarray;   //тип указателя на массив целых
typedef ptrtointarray TYPE (void);
```

Теперь имя TYPE можно использовать, например, в прототипах:

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

15. Правила преобразования стандартных типов

В любых случаях выполняются два преобразования:
имя массива преобразуется к указателю на его первый элемент;
имя функции преобразуется к указателю на эту функцию.

15.1. Явные преобразования

Разрешены любые преобразования стандартных типов одного к другому. При преобразовании более длинного типа к более короткому происходит потеря разрядов; при преобразовании более короткого целочисленного типа к более длинному свободные разряды заполняются 0 (если короткий тип – беззнаковый), или происходит размножение знакового разряда (для типа со знаком).

Разрешены любые преобразования друг на друга указателей, а также ссылок. Явное преобразование типов делается посредством операции приведения типов (cast), которая имеет две формы:

```
(имя_типа) операнд          // Традиционная форма;
или
имя_типа (операнд)          // функциональная форма.
```

Здесь **имя_типа** задаёт тип, а **операнд** является величиной, которая должна быть преобразована к заданному типу.

Отметим, что во второй форме **имя_типа** должно быть простым идентификатором, например, полученным с помощью typedef.

Примеры:

```
double d = (double)5;
int i = int(d);
int *ip = &i;
float *fp = (float*) ip;
typedef float*  FP;
fp = FP(ip);
```

15.2. Неявные преобразования стандартных базовых типов

Для стандартных базовых типов компилятор может выполнять любые преобразования одного типа к другому:

```
int i='A';          // i = 65;
char c=256;         // Теряются 8 старших битов; с станет равно '\0';
int j=-1;
long l=j;
long m=32768;       // Двоичное представление числа 32768
                    // содержит единственную единицу в 15 разряде.
short int k=m;      // k = -32768, так как 15-й разряд для short –
                    // знаковый.
unsigned u=m;       // u = 32768
double d=0.999999;
long n=d;           // n = 0
```

При выполнении арифметических операций также происходит неявное преобразование типов. Правила здесь такие:

а) типы char, short, enum преобразуются к типу int, а unsigned short – к unsigned int; тип float преобразуется к double;

б) затем, если один из операндов имеет тип long double, то и второй преобразуется к long double;

в) иначе, если один из операндов имеет тип double, то и второй преобразуется к double;

г) иначе, если один из операндов имеет тип unsigned long, то и второй преобразуется к unsigned long;

д) иначе, если один из операндов имеет тип unsigned, то и второй преобразуется к unsigned;

е) иначе, если один из операндов имеет тип long, то и второй преобразуется к long;

ж) иначе оба операнда имеют тип int.

Пример 1.

```
int g = 10, t = 5;
double t2=t*t/2;
double s = g*t2;               // s  станет равно  120;
double s0 = g*t*t/2.0;         // s0 станет равно 125.
```

Пример 2.

Функция **atoi** (упрощенная), которая ставит в соответствие строке цифр её числовой эквивалент:

```
int atoi ( char s[ ] ){
int i, n = 0;
for (i = 0; s[i] >= ’0’&& s[i] <= ’9’; ++i)
n = 10*n + s[i] – ’0’;         // Преобразование  char  в  int.
return  n;
}
```

15.3. Преобразование производных стандартных типов

Для указателей разрешено неявное преобразование указателя на любой тип к указателю на тип void. Все другие преобразования должны быть явными.

```
int *ip;
void *vp=ip;
ip=vp;                    // Ошибка!
ip=(int*)vp;              // Теперь верно.
float *fp=ip;             // Ошибка.
float *fp=(float*)ip;     // Верно.
```

Константа 0 может быть неявно преобразована к указателю на любой тип. При этом гарантируется, что такой указатель не будет ссылаться ни на один объект. Значение стандартной константы NULL равно 0 для всех видов указателей.

16. Функции

16.1. Определение и вызов функции

Программа С++ состоит из одной или нескольких функций. Функции разбивают большие задачи на маленькие подзадачи.

Имя одной из функций – **main** , которая обязательно должна присутствовать в любой программе, является зарезервированным. Функция main необязательно должна быть первой, хотя именно с нее начинается выполнение программы.

Функция не может быть определена в другой функции.

С использованием функции связаны 3 понятия – определение функции, объявление функции и вызов функции.

Определение функции имеет вид

тип имя (**список описаний аргументов**){ **операторы** }

Здесь **имя –** это имя функции;

тип – тип возвращаемого функцией значения;

операторы в фигурных скобках { } часто называют телом функции.

Аргументы в списке описаний называют **формальными параметрами**.

Например, функция, находящая и возвращающая максимальное значение из двух целых величин a и b определяется так:

```
int  max(int a, int b){ return(a>=b)? a:b; }
```

Это определение говорит о том, что функция с именем **max** имеет два целых аргумента и возвращает целое значение. Если функция действительно должна возвращать значение какого-либо типа, то в ее теле обязательно должен присутствовать оператор return **выражение**; при выполнении этого оператора выполнение функции прекращается, управление передается в функцию, вызывающую данную функцию, а значением функции будет значение **выражения**.

```
int max(int a, int b) { return (a >=b)? a:b; }
void main( ) {
int  i = 2,  j = 3;
int c = max( i, j );
cout<<“ max= “<<c<<“\n”;
c = max( i*i, j )*max( 5, i – j );
cout<<“ max= “<<c<<“\n”;}
```

В этой программе приведено определение функции max и 3 обращения к ней. При обращении указывается имя функции и в круглых скобках список **фактических параметров**.

Если у функции нет формальных параметров, то она определяется, например, так:

double f(void){тело функции};

или, эквивалентно,

double f( ) {тело функции};

Обращаются в программе к этой функции, например, так:

a = b*f( ) + c;

Функция может и не возвращать никакого значения. В этом случае ее определение таково:

void **имя** (**список описаний аргументов**){ **операторы** }

Вызов такой функции имеет вид

имя (**список фактических аргументов**);

Выполнение функции, не возвращающей никакого значения, прекращается оператором return без следующего за ним выражения. Выполнение такой функции и возврат из нее в вызывающую функцию происходит также и в случае, если при выполнении тела функции произошел переход на самую последнюю закрывающую фигурную скобку этой функции.

В качестве примера приведем функцию, копирующую одну строку в другую:

```
void copy (char* to, char* from){
while(* to ++ = *from ++ );}
void main(){
char str1[ ]="string1";
char str2[ ]="string2";
copy(str2, str1);
cout<<str2<<"\n";
}
```

16.2. Функции. Передача аргументов

В приведенных выше примерах происходит так называемая передача аргументов ***по значению***. Такая передача аргументов означает, что в вызываемой функции для каждого формального аргумента создаётся локальный объект, который инициализируется значением фактического аргумента. Следовательно, при такой передаче изменения значений формальных параметров функции не приводит к изменению значений соответствующих им фактических аргументов.

Рассмотрим, например, вариант функции, возводящей целое **x** в целую степень **n**, где используется это обстоятельство.

```
int power (int x, int n){
for (int p = 1; n > 0; n --) p* = x;
return p;
}
```

Аргумент **n** используется как временная переменная. Что бы ни происходило с **n** внутри функции power, это никак не влияет на фактический аргумент, с которым первоначально обратились к этой функции в вызываемой функции:

```
void main (){
…
int n=6,
x=3;
x=power(x, n);      // n – не меняется.
…
}
```

Рассмотрим процесс вызова функции более подробно. При вызове функции:

- в стеке резервируется место для формальных параметров, в которые записываются значения фактических параметров. Обычно это производится в порядке, обратном их следованию в списке;
- при вызове функции в стек записывается точка возврата – адрес той части программы, где находится вызов функции;
- в начале тела функции в стеке резервируется место для локальных (автоматических) переменных.

В случае, если функция должна менять свои аргументы, можно использовать указатели. Указатели также передаются по значению, внутри функции создается локальная переменная – указатель. Но так

как этот указатель инициализируется адресом переменной из вызываемой программы, то эту переменную можно менять, используя этот адрес.

В качестве примера рассмотрим функцию, меняющую местами свои аргументы:

```
void swap (int* x, int* y){
int t = *x;
*x = *y;
*y = t;
}
```

Обратиться к этой функции можно так:

```
int a = 3, b = 7;
swap (&a, &b);
```

Теперь a =7, и b=3.

Некоторую особенность имеет использование массивов в качестве аргументов. Эта особенность заключается в том, что имя массива преобразуется к указателю на его первый элемент, то есть при передаче массива происходит передача указателя. По этой причине вызываемая функция не может отличить, относится ли передаваемый ей указатель к началу массива или к одному единственному объекту.

```
int summa (int array[ ], int size){
int res=0;
for (int i = 0; i <size; i++) res+ = array[i];
return res;
}
```

В заголовке int array[ ] можно заменить на int* array, а выражение в теле функции array[i] заменить на *(array+i), или даже на *array ++, так как array не является именем массива, и следовательно, не является константным указателем. К функции summa можно обратиться так:

```
int mas[100];
for (int i = 0; i < 100; i++) mas[i] = 2*i + 1;
int j = summa (mas, 100);
```

Пример: вычисление многочлена по его коэффициентам.

Пусть требуется вычислить многочлены

$$
\begin{aligned}
&P_3(x) = 4x^3 + 2x^2 + 1, \\
&P_5(x) = x^5 + x^4 + x^3 + x^2 + x + 7, \\
&P_9(x) = x^9 + 2x^7 + 3x^6 + x^5 + x^2 + 2
\end{aligned} \tag{1}
$$

в точке x=0.6.

Ввиду важности вычисления многочленов составим функцию, осуществляющую вычисление многочлена степени n

$$P_n(x) = C_0 + C_1 x + C_2 x^2 + \ldots + C_n x^n$$

по его коэффициентам C_i. Для эффективного вычисления многочлена используем так называемую схему Горнера (которую на самом деле за 100 лет до Горнера описал Ньютон). Эта схема состоит в том, что многочлен переписывается в следующей форме:

$$P_n(x) = (\ldots((0 \cdot x + C_n) \cdot x + C_{n-1}) \cdot x + \ldots + C_1) \cdot x + C_0 .$$

При таком порядке вычислений для получения значения P_n(x) требуется лишь n умножений и n сложений.

Коэффициенты многочленов будем хранить в массиве **c**.

Для вычисления многочленов (1) напишем программу, в которой схема Горнера реализована в функции pol().

```
const  int n = 10;
double pol (int n, double c[ ], double x){
double p=0;
for (int i = n; i >= 0; i – –) p = p*x + c[i];
return  p;}

void main(){
double x=0.6, p, c[n] = {1, 0, 2, 4};
p=pol(3, c, x);
cout<<”x= ”<<x<<”  Polynom = ”<<p<<’\n’;
c[0]=7;
c[1]=c[2]=c[3]=c[4]=c[5]=1;
p=pol(5, c, x);
cout<<”x= ”<<x<<”  Polynom = ”<<p<<’\n’;
c[0]=2; c[2]=1; c[5]=1; c[6]=3; c[7]=2; c[9]=1;
c[1]=[3]=c[4]=c[8]=0;
cout<<”x= ”<<x<<”  Polynom = ”<<pol(9, c, x)<<’\n’;
}
```

16.3. Передача многомерных массивов

Если функции передаётся двумерный массив, то описание соответствующего аргумента функции должно содержать количество столбцов; количество строк – несущественно, поскольку фактически передаётся указатель.

Рассмотрим пример функции, перемножающей матрицы A и B; результат – C.

Размер матриц – не более 10.

```
const int nmax = 10;

void product (int a[ ][nmax], int b[ ][nmax], int c[ ][nmax],
                 int m, int n, int k){
/*  m –       число строк в матрице a;
     n –      число строк в матрице  b (должно быть равно числу
              столбцов  в  матрице  a);
     k –      число столбцов в матрице  b.
*/

for (int i=0; i<m; i++)
for (int j=0; j<k; j++){
c[i][j]=0;
for (int l=0; l<n; l++) c[i][j] + = a[i][l]*b[l][j];
}
}
```

Если заданы, например, квадратные матрицы A и B размером 5×5, то получить их произведение C можно так:

```
product (A, B, C, 5, 5, 5);
```

В приведённом примере есть недостаток – здесь заранее фиксируется максимальная размерность матриц. Обойти это можно различными способами, один из которых – использование вспомогательных массивов указателей на массивы.

Напишем функцию, транспонирующую квадратную матрицу произвольной размерности n.

```
void trans ( int n, double *p[ ] )
{double x;
for (int i=0; i<n-1; i++)
for (int j=i+1; j<n; j++)
{x=p[i][j]; p[i][j]=p[j][i]; p[j][i]=x;}}

void main(){
double A[4][4]={   10, 12, 14, 17
                   15, 13, 11,  0
                    -3, 5.1, 6,  6
                     2,   8,   3,  1};
double ptr[ ]={(double*)&A[0], (double*)&A[1],
               (double*)&A[2], (double*)&A[3]};
int n=4;
trans (n, ptr);
for (int i=0; i<n; i++){ count<<"\n строка"<<(i+1)<<":";
      for (int j; j<n; j++)
      cout<<"\t"<<A[i][j];
      cout<<\n";
}
}
```

В функции main матрица представлена двумерным массивом double A[4][4]. Такой массив нельзя непосредственно использовать в качестве фактического параметра, соответствующего формальному double *p[]. Поэтому вводится дополнительный вспомогательный массив указателей double *ptr[]. В качестве начальных значений элементам этого массива присваиваются адреса строк матрицы, преобразованные к типу double*.

Многомерный массив с переменными размерами может быть динамически сформирован внутри функции. В вызываемую функцию его можно передать как указатель на одномерный массив указателей на одномерные массивы с элементами известной размерности и заданного типа.

В качестве примера приведем функцию, формирующую единичную матрицу порядка n.

```
int** singl (int n){
int **p=new int*[n];

/* Тип  int* [n] – массив указателей на целые.
```

Операция new возвращает указатель на выделенную память под этот массив и тип переменной p есть int**. Таким образом, p есть массив указателей на строки целых будущей матрицы.

```
*/

if (p == NULL){
cout<<"Динамический массив не создан!\n";
exit(1);
}
// цикл создания одномерных массивов – строк матрицы:
for (int i=0; i<n; i++){
p[i]=new int[n];
if( !p[i] ){cout<<"Не создана динамическая строка!\n";  exit(1);}
for (int j=0; j<n; j++)
p[i][j]=(i = =j )? 1: 0;
}
return  p;
}

void main(){
int n;
cout<<"\n Задайте порядок матрицы:  ";
cin>>n;

int** matr;             //Указатель для формируемой матрицы
matr = singl(n);
for (int i=0; i<n; i++){cout<<"\n строка";
cout.width(2);
cout<<i+1<<": ";
for (int j=0; j<n; j++){
cout.width(4);
cout<<matr[i][j];
}
}
for(i=0; i<n; i++) delete matr[i];
delete matr;
}
```

В этой программе обращение к функции cout.width(**k**) устанавливает ширину поля следующего вывода в **k** позиций, что позволяет выровнять вид полученной матрицы.

16.4. Указатели на функции

Указатель на функцию определим следующим образом:

Тип_функции (*имя_указателя) (список параметров);

Например,

int (*fptr) (double);

Здесь определяется fptr как указатель на функцию с одним аргументом типа double, которая возвращает значение int. Имя функции без следующих за ним ( ) – это указатель на функцию, который содержит адрес начала кода этой функции.

Пример:

```
void f1(void){
cout<<"\n Выполняется f1().";
}
void f2(void){
cout<<"\n Выполняется f2().";}
void main(){
void (*ptr)(void);
ptr = f2;
(*ptr)();       //вызов функции f2();
ptr = f1;
(*ptr)();       //вызов f1();
ptr();          //альтернативная запись!
}
```

Результат:

```
Выполняется f2().
Выполняется f1().
Выполняется f1().
```

Иногда удобно использовать формальные параметры функции, являющиеся указателями на функции.

Проиллюстрируем это при решении следующей задачи.

Вычислить интеграл, используя метод трапеций, от двух разных функций.

```
// Файл TRAP.CPP
double trap(double (*func)(double), double a, double b, int n){
double x, h = (b-a)/n, i = (func(a) + func(b))/2;
for (x = a+h; x<b-h/2; x+=h) i + = func(x);
return  h*i;
}
//Файл INTEGRAL.CPP
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include “trap.cpp“

double f1(double x){
return x*x+sin(3+x);}

double f2(double x){
return x/(x*x+1)+exp(-2*x*x);}

void main( ){
double a=1, b=3;
double i = trap(f1, a, b, 50);
cout <<”Интеграл от первой функции = ”<<i<<’\n’;
i = trap(f2, a, b, 50);
cout <<”Интеграл от второй функции = ”<<i<<’\n’;
}
```

16.5. Ссылки

Тип "ссылка на **тип**" определяется так: **тип**&, например:

int& **или** double&

Ссылочные типы устанавливают псевдонимы объектов. Ссылка обязательно должна быть инициализирована. После инициализации использование ссылки дает тот же результат, что и прямое использование переименованного объекта.

Рассмотрим инициализацию ссылки:

```
int i=0;
int&  iref = i;
```

Здесь создана новая переменная типа ссылка на int с именем iref. Физически iref – это **постоянный** указатель на int и, следовательно, значение ссылки после инициализации не может быть изменено. Инициализирующим значением в нашем случае является адрес переменной i, т.е. при инициализации ссылка ведёт себя как указатель.

При использовании ссылка ведёт себя не как указатель, а как переменная, адресом которой она инициализирована:

```
iref ++;            // то же, что и i++;
int *ip=&iref;      // то же, что и ip=&i.
```

Итак, iref стало другим именем, псевдонимом переменной i.

Ссылку можно определить так:

ссылка есть такой постоянный указатель на объект, к которому всегда при его использовании неявно применяется операция разрешения указателя *.

Если тип инициализированной ссылки не совпадает с типом объекта, создаётся новый анонимный объект, для которого ссылка является псевдонимом. Инициализатор преобразуется и его значение используется для установки значения анонимного объекта.

```
double d=0.0;
int& ir = d;        // Создан анонимный объект типа int;
ir = 3.0;           // d – не меняется!
```

Здесь создаётся анонимная переменная типа int, которая инициализируется значением, полученным в результате преобразования значения типа double к типу int. Затем ссылка инициализируется значением адреса этой переменной.

Анонимный объект создаётся также, когда инициализатор не является объектом, например, является константой:

```
int& ir = 3;            // Анонимный объект получил значение 3.
```

Здесь сначала создается анонимный объект типа int и он инициализируется значением 3. После этого создаётся ссылка ir и инициализируется адресом анонимного объекта. Теперь ir – его псевдоним и оператор

```
ir = 8;
```

устанавливает новое значение этого анонимного объекта.

16.6. Ссылки в качестве параметров функций

Ссылки часто используются в качестве формальных параметров функций. Механизм передачи параметров в функции с помощью ссылок называют в программировании передачей аргументов ***по ссылке.*** С помощью ссылок можно добиться изменения значений фактических параметров из вызывающей программы (без применения указателей).

```
void swap (int & x, int & y){
int t = x;
x = y;
y = t;
}
```

Теперь обращение в вызывающей функции имеет вид:

```
int a = 3, b = 7;
swap (a, b);
```

Здесь создаются локальные относительно функции swap() переменные x и y ссылочного типа, которые являются псевдонимами переменных a и b и инициализирующиеся переменными a, b. После этого все действия с x и y эквивалентны действиям с a и b, что приводит к изменению значений a и b.

Заметим, что в последнем примере можно обратиться к функции swap() и с аргументами, отличными по типу от int, и даже с аргументами, не являющимися объектами:

```
float a = 5, b = 2.7;
swap (a, b);
swap (3, a+b);
```

Однако в этих случаях функция swap() фактически никакого действия со своими аргументами не выполняет. Создаются временные объекты типа int, которые инициализируются значениями, полученными в результате преобразования a, b, a+b к типу int; затем ссылки x и y инициализируются значениями адресов этих анонимных объектов; анонимные объекты и будут изменены. Фактические же параметры останутся неизменными.

Компилятор выдаст предупреждение, что он вынужден завести временные переменные, и будет работать с ними.

16.7. Рекурсивные функции

Ситуацию, когда функция тем или иным образом вызывает саму себя, называют рекурсией. Рекурсия, когда функция обращается сама к себе непосредственно, называется прямой; в противном случае она называется косвенной.

Все функции языка C++ (кроме функции main) могут быть использованы для построения рекурсии.

В рекурсивной функции обязательно должно присутствовать хотя бы одно условие, при выполнении которого последовательность рекурсивных вызовов должна быть прекращена.

Обработка вызова рекурсивной функции в принципе ничем не отличается от вызова функции обычной: перед вызовом функции в стек помещаются её аргументы, затем адрес точки возврата, затем, уже при выполнении функции – автоматические переменные, локальные относительно этой функции. Но если при вызове обычных функций число обращений к ним невелико, то для рекурсивных функций число вызовов и, следовательно, количество данных, размещаемых в стеке, определяется глубиной рекурсии. Поэтому при рекурсии может возникнуть ситуация переполнения стека.

Если попытаться отследить по тексту программы процесс выполнения рекурсивной функции, то мы придем к такой ситуации: войдя в рекурсивную функцию, мы “движемся” по ее тексту до тех пор, пока не встретим ее вызова, после чего мы опять начнем выполнять ту же самую функцию сначала. При этом следует отметить самое важное свойство рекурсивной функции – ее первый вызов еще не закончился. Чисто внешне создается впечатление, что текст функции воспроизводится (ко-

пируется) всякий раз, когда функция сама себя вызывает. На самом деле этот эффект воспроизводится в компьютере. Однако копируется при этом не весь текст функции (не вся функция), а только ее части, связанные с данными (формальные, фактические параметры, локальные переменные и точка возврата). Алгоритм (операторы, выражения) рекурсивной функции не меняется, поэтому он присутствует в памяти компьютера в единственном экземпляре.

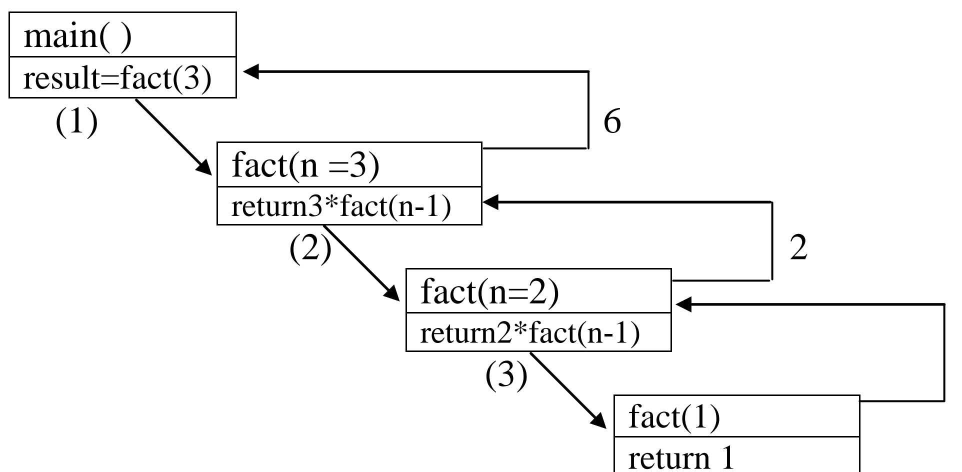
Пример1. Вычислить n!

Определение факториала рекурсивно:
0!=1; n!=(n-1)!*n при n=1,2,3, …

В соответствии с этим определение фкнкции, вычисляющей факториал, можно записать следующим образом;

```
long fact (int n){
if ( n<1 ) return 1;
else return n*fact(n-1);
}
```

Если, например, в main написать long result=fact(3), то последовательность вызовов можно показать так:



Пример 2. По заданному целому числу распечатать символьную строку цифр, изображающую это число:

```
void cnum(int n){int a=10;
if(n = = 0) return;
else {cnum(n/a); cout <<n%a;}
}
```

При косвенной рекурсии осуществляется перекрёстный вызов функциями друг друга. Хотя бы в одной из них должно быть условие, вызывающее прекращение рекурсии.

Косвенная рекурсия является одним из тех случаев, когда нельзя определить функцию до использования её имени в программе.

Пусть функция f1() вызывает f2(), которая, в свою очередь, обращается к f1(). Пусть первая из них определена ранее второй. Для того чтобы иметь возможность обратиться к функции f2() из f1(), мы должны поместить объявление имени f2 раньше определения обеих этих функций:

```
void f2();
void f1(){
…
if (…);
f2();
…}
void f2(){
…
f1();
…}
```

16.8. Аргументы по умолчанию

Удобным свойством С++ является наличие предопределённых инициализаторов аргументов. Значения аргументов по умолчанию можно задать в объявлении функции, при этом они подставляются автоматически в вызов функции, содержащий меньшее число аргументов, чем объявлено. Например, следующая функция объявлена с тремя аргументами, два из которых инициализированы:

```
void error (const char* msg, int level = 0, int kill = 0);
void error (char* msg, int level = 0, int kill = 0);
// Перегрузка функции.
```

Эта функция может быть вызвана с одним, двумя или тремя аргументами:

```
error (“Ошибка!”);        // Вызывается error (“ошибка”, 0, 0);
error (“Ошибка!”, 1);     // вызывается  error (“ошибка”, 1, 0);
error (“Ошибка!”, 3, 1);  // значения аргументов по умолчанию
                          // не используются.
```

Все аргументы по умолчанию должны быть последними аргументами в списке – ни один явный аргумент не может находиться правее их.

Если аргумент по умолчанию уже определён в одном объявлении, он не может быть переопределён в другом. Аргументы по умолчанию должны быть объявлены при первом объявлении имени функции и не обязаны быть константами:

```
int i=8;
void func (int = i);
```

Заметим, что если инициализация аргументов произведена в прототипе функции, в определении функции задавать инициализацию аргументов не надо.

16.9. Перегрузка функций

В C++ можно перегружать имена функций и использовать одно и то же имя для нескольких функций с различным типом или числом аргументов.

Пусть объявлены следующие функции:

```
int func(int, int);
int func(char, double);
int func(long, double);
int func(float, …);          // Функция с неопределенным числом
                             // аргументов.
int func(char*, int);
```

Рассмотрим, что будет происходить при вызове функции с именем func с некоторым списком аргументов.

Первое, что будет делать компилятор – это пытаться найти функцию, формальные аргументы которой соответствуют фактическим без всяких преобразований типов или с использованием только неизбежных преобразований – например, имени массива к указателю.

```
char string[ ]=”Строка – это массив символов”;
int i=func (string, 13);          // func (char*, int);
int j=func(1995L, 36.6);          // func(long, double);
```

Если на первом этапе подходящая функция не найдена, то на втором этапе совершается попытка подобрать такую функцию, чтобы для соответствия формальных и фактических аргументов достаточно было использовать только такие стандартные преобразования, которые не влекут за собой преобразований целых типов к плавающим и наоборот.

При этом подбирается функция, для которой число таких преобразований было бы минимально.

Пусть обращение к функции выглядит так:

```
float a=36.6;
j=func('a', a);
```

Применяя указанные стандартные преобразования, найдём, что будет вызвана функция с прототипом func(char, double) и аргумент a будет преобразован к типу double.

Третьим этапом является подбор такой функции, для вызова которой достаточно осуществить любые стандартные преобразования аргументов (и опять так, чтобы этих преобразований было как можно меньше).

Так, в следующем фрагменте

```
char y[] = "ГОД:";
int l = func(y, 2018.3);
```

будет вызвана функция func (char*, int), фактический аргумент типа double которой будет преобразован к int с отбрасыванием дробной части числа.

На четвёртом этапе подбираются функции, для которых аргументы можно получить с помощью всех преобразований, рассмотренных до этого, а также преобразований типов, определённых самим программистом.

Если и в этом случае единственная нужная функция не найдена, то на последнем, пятом этапе, компилятор пробует найти соответствие с учётом списка неопределённых аргументов.

Так, при вызове

```
func (1, 2, 3);
```

подходит лишь одна функция func(float, …).

При обращении

```
int i, j, n;
…
n=func(&i, &j);
```

компилятор не найдёт ни одной подходящей функции и выдаст сообщение об ошибке.

16.10. Шаблоны функций

Цель введения шаблонов функций – автоматизация создания функций, которые могут обрабатывать разнотипные данные.

В определении шаблонов семейства функций используется служебное слово **template**, за которым в угловых скобках следует список параметров шаблона. Каждый формальный параметр шаблона обозначается служебным словом **class**, за которым следует имя параметра.

Пример: определение шаблона функций, вычисляющих модули величин различных типов.

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

Шаблон функций состоит из двух частей – заголовка шаблона и из обычного определения функции, в котором тип возвращаемого значения и типы любых параметров и локальных переменных могут обозначаться именами параметров шаблона, введённых в его заголовке.

Пример (снова функция swap):

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

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в тексте программы. Например, при обращении

```
abs(-10, 3)
```

компилятор сформирует такое определение функции:

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

Далее будет организовано выполнение именно этой функции и в точку вызова в качестве результата вернется значение 10.3.

Пример: шаблон функций для поиска в массиве максимального элемента.

```
#include <iostream>
// Функция устанавливает ссылку
// на элемент с максимальным значением
template <class type>
type & r_max ( int n, type d[ ] ){
int im=0;
for (int i = 1; i < n; i++) im = d[im] > d[i] ? im : i;
return d[im];
}

void main(){
int n = 4, x[ ]={10, 20, 30, 5};
cout<<"\n  r_max (n, x)="<< r_max (n, x);   // Печать максимального
                                            // элемента.
r_max (n, x) = 0;                   // Замена в целом массиве
                                    // максимального элемента нулем.
for (int i=0; i<n; i++)
cout<<"\t x["<<i<<"]="<< x[i];
float f[]={10.3, 50.7, 12.6};
cout<<"\n  r_max (3, f)="<< r_max (3, f);
r_max (3, f)=0;
for (i=0; i<3; i++)
cout<<"\t f["<<i<<"]="<<f[i];
}
```

Результат выполнения программы:

```
r_max (n, x)=30 x[0]=10  x[1]=20  x[2]=0  x[3]=5
r_max (3, f)=50.7        f[0]=10.3  f[1]=0  f[2]=12.6
```

При использовании шаблонов уже нет необходимости готовить заранее все варианты функций с перегруженным именем. Компилятор

автоматически, анализируя вызовы функций в тексте программы, формирует необходимые определения именно для таких типов параметров, которые использованы в обращении.

Перечислим основные свойства параметров шаблона.

- Имена параметров шаблона должны быть уникальными во всём определении шаблона.

- Список параметров шаблона функций не может быть пустым.

- В списке параметров шаблона функций может быть несколько параметров. Каждый из них должен начинаться со служебного слова class.

- Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами.

- Имя параметра шаблона имеет в определяемой шаблоном функции все права имени типа. Имя параметра шаблона видно во всём определении и скрывает другие использования того же идентификатора в области, внешней по отношению к данному шаблону.
- Все параметры шаблона должны быть обязательно использованы в спецификациях формальных параметров определения функции.

Заметим, что при необходимости можно использовать прототипы шаблона функций. Например, прототип функции swap():

```
template <class type>
void swap (type&, type&);
```

При конкретизации шаблонного определения функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковыми.

Так, недопустимо:

```
int n=5;
double d=4.3;
swap (n, d);
```

17. Перечисления

Перечисление – это тип данных, который удобно использовать в случае применения в программе переменных и констант, принимающих значения из сравнительно небольшого множества целых чисел, причём таких, что обычно обращение к ним разумнее осуществлять по имени. Может быть и так, что их значение само по себе не важно.

Примером такого множества констант могут служить названия цветов, названия дней недели или месяцев года, названия шахматных фигур или символические имена арифметических операций языка C++.

Задание типа **перечисление** начинается ключевым словом **enum**, после которого идёт **имя** типа (иногда называемое **тегом**), за которым в фигурных скобках следует **список членов перечисления** – перечислителей:

```
enum chess {king, queen, rook, bishop, knight, p};

enum month {Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec};

enum operator_CPP{plus='+', minus='-', mult='*', div='/', rem='%'};
```

Члены перечислений являются константами типов unsigned char или int, в зависимости от их значений и режима компиляции. При использовании перечислителя в выражении его тип всегда преобразуется в int.

Если перечислителям явно не присвоено никакого значения, как в chess и mouth, то первый из них получает значение 0, второй – 1, и т.д. Вообще, любой перечислитель по умолчанию имеет значение, на 1 превышающее значение предыдущего, если умолчание не отменяется явной инициализацией.

Все члены перечисления operator_CPP получают явно указанные значения, равные ASCII – кодам символов '+', '-', '*', '/', '%'.

Значения перечислителей вычисляются на стадии компиляции, поэтому при задании их значений можно использовать значения всех ранее определенных констант. При этом допускается, чтобы несколько перечислителей имели одно и то же значение:

```
const int TWO = 2;
enum {first, second=first, next=first+TWO,
last =next*next+1}dummy;
```

Отметим, что в последнем случае не вводится тег перечисления, а сразу описывается переменная dummy, которая в соответствии с шаблоном перечисления может принимать одно из четырех значений:

first, second, next, last.

Следовательно, возможны присваивания

```
dummy = first;
dummy = last;
```

В общем случае переменной-перечислению можно присваивать значения, задаваемые перечислителями.

Так, если сделать описания

```
month  m1, m2;
operator_CPP  op1, op2, op3;
enum colour { red, green} c[10];
```

то переменным op1, op2, op3 можно присваивать значения plus, minus, ...; m1, m2 – Jan, Feb и т. д., а элементам массива c[0] ... c[9] – значения red и green.

Перечислитель может появиться везде, где разрешено появление значения типа int. Обратное неверно без явного преобразования типа.

```
int i = dummy;                    // i равно 5
month current_month = Jan;
current_month =0;                 // Ошибка!
current_month = ( month )0;       // Теперь верно!
Feb = 1;                          // Ошибка: Feb – константа!
```

Пример:

```
# include < iostream >
const int NUMDAYS = 7;
enum DAYS { mon, tue, wen, thur, fri, sat, sun } day1, day2, day3;
DAYS day_before(DAYS), day_after(DAYS);
void print_day(DAYS);
void  main() {
        setlocale(0, "rus");
        day1 = sun;
        day2 = day_after(day1);
        day3 = day_before(day1);
        cout << "Если сегодня ";
```

```
        print_day(day1);
        cout << ", то завтра будет ";
        print_day(day2);
        cout << ", \nа вчершний день - ";
        print_day(day3);
        cout << ".\n";
        system("pause");
}

DAYS  day_after(DAYS day) {
        return ((DAYS)((day + 1) % NUMDAYS));
}

DAYS day_before(DAYS day) {
        int prev = (day - 1) % NUMDAYS;
        return (DAYS)((prev <0) ? (NUMDAYS - 1) : prev);
}

void print_day(DAYS day) {
        int day_i = day;
        const char * days[] = {
                "Понедельник",
                "Вторник" ,
                "Среда",
                "Четверг",
                "Пятница",
                "Суббота",
                "Воскресенье"
        };

        if (day_i <0 || day_i > NUMDAYS)
                cout << "  Ошибка!\ n"; else
                cout << days[day_i];
        }
```

Результат выполнения:

```
Если сегодня воскресенье, то завтра будет
Понедельник,
а вчерашний день - Суббота.
```

18. Классы

18.1. Объявление классов

Тип данных **класс** можно определить с помощью конструкции

ключ_класса, **имя_класса**{**список _членов**};

Здесь **ключ_класса** – одно из служебных слов struct, union, class;
имя_класса – произвольный идентификатор;
список_членов – определения и описания членов класса – данных и функций.

Класс – это набор из одной или более переменных и функций, возможно, различных типов, сгруппированных под одним именем.

Пример структуры – учётная карточка сотрудника, в которой содержится фамилия, имя, отчество, адрес, должность, год поступления на работу и т.д. Некоторые из этих атрибутов сами могут оказаться структурами. Так, Ф.И.О. имеет три компоненты, адрес – также несколько компонент.

Класс может иметь имя, иногда называемое тегом. Тег становится именем нового типа в программе. Каждый член класса распознаётся по своему имени, которое должно быть уникальным в данном классе. Члены класса иногда называют его элементами или полями.

Хотя каждому имени члена класса сопоставлен определённый тип, самостоятельным объектом такой член не является. Память выделяется только для конкретного объекта вновь определенного типа в целом.

Введем новые типы FIO и sotrudnik:

```
struct FIO{  char familia [39],
             imya [30],
             otchestvo [30]
};
struct sotrudnik{ FIO name;
             char dolgnost [30];
             int year;
             double oklad};
```

Здесь заданы два новых типа структурных переменных и имена этих типов – FIO, sotrudnik. Заметим, что наличие «;» после фигурных скобок здесь обязательно.

Теперь можно объявить структурные переменные типа FIO или sotrudnik обычным образом:

```
FIO name1, name2, name3;
sotrudnik s1, s2, s[50];
```

Теперь компилятором будет выделена память под переменные name1, name2, name3, s1, s2 и под массив s из пятидесяти структур. Отметим, что число байтов, выделяемое под структурную переменную, не всегда равно сумме длин отдельных членов структуры из-за эффекта выравнивания, производимого компилятором.

Чтобы определить выделенное число байтов, надо воспользоваться операцией sizeof, например, так:

```
int nf=sizeof (FIO), ns=sizeof (sotrudnik);
```

Заметим также, что объявить структурные переменные можно одновременно с определением имени структуры:

```
struct DATE {
int day;
int month;
int year;
char mon_name[4] } d1, d2, d3;
```

Здесь объявлены три переменных d1, d2, d3, которые имеют тип структуры DATE.

Можно объявить структурную переменную и без введения имени (тега) структуры:

```
struct {int price;
double length [10]} a, b, c, d;
```

После того, как определены структурные переменные, доступ к их членам осуществляется с помощью операции извлечения '**.**':

```
a.price  c.length d1.day  d3.mon_name  s[25].oklad  s[0].name.familia.
```

Имена наподобие c.length, d1.day, d3.mon_name, с помощью которых происходит доступ к членам класса, иногда называют уточненными именами. Если определить указатель на структуру, DATE* datep=&d1,

то обратиться к члену структуры d1 можно так: **(*datep).year**, или с помощью операции извлечения из указателя на структуру **-> datep->year**, что эквивалентно.

Введем теперь простейший класс “комплексное число”:

```
struct compl{ double real, imag;
void define ( double re = 0.0, double im = 0.0 ){
real = re; imag = im;               // задание комплексного числа.
}
void display( ){cout <<   “real = ”<< real <<
                          “, imag = ” << imag << “\n“;
               }
};
```

Здесь real, imag – данные-члены, а define(), display() – **функции-члены**, которые часто называют **методами** класса.

Теперь можно описать объекты типа compl:

```
compl a, b, c, *pc = &c;
```

После этих определений данные-члены структурных переменных доступны в области их видимости:

```
a.define(3, 7);       // Определяется  комплексное число 3+7i,
                      // т.е. a. real = 3; a.imag = 7;
b.define(2);          // определяется комплексное число 2+0*i = 2;
c.define( );          // комплексное число = 0;
                      // оба параметра выбираются по умолчанию.
```

Данные-члены можно задавать и использовать непосредственно, не через функции define( ), display( ):

```
a.real = 3; a.imag = 7; (*pc).real = 1; pc->imag = -1;
a.real+ = b.real*3+7;
cout <<“pc→real: ” <<pc->real<<“\n”;
a.display( );
b.display( );
c.display( );
```

Здесь данные-члены структуры доступны для использования в программе минуя функции-члены. Можно запретить произвольный доступ к данным. Для этого обычно вместо слова struct в определении класса используют слово class:

```
class complex{double real, imag;
public:
void display( ){      cout <<"  real =" <<real;
                      cout <<", imag =" << imag <<'\n';
}

void define(double re = 0.0, double im = 0.0){
real = re; imag = im;
}
};
```

Метка **public**, которая может присутствовать в объявлении класса, в нашем примере делит его тело на две части – "личную", или "собственную" (private) и общую – (public).

Доступ к данным-членам класса, находящимся в собственной части, возможен лишь через функции-члены класса:

```
complex s1, s2, *ps = &s1;
s1.define( );          // s1.real=0; s1.imag=0;
s1.display( );         // Выводится  real=0, imag=0;
ps->display( );        // то же самое.
s1.real=3;             // Ошибка! private-член  s2.real недоступен!
```

В определении класса может также явно присутствовать метка private.

Метки private и public делят тело класса в общем случае на части, различающиеся по уровню доступа. К членам класса, находящимся в собственной (private) части, доступ возможен только с помощью функций-членов и так называемых **дружественных** или **привилегированных** функций.

К общим же членам класса можно обратиться из любой функции программы.

Основное отличие struct и class состоит в уровне доступа по умолчанию. Если нет явного указания уровня доступа, то все члены структуры считаются общими, а класса – собственными. Явное указание уровней доступа делает слова struct и class равнозначными. Обычно использование слова struct вместо class говорит о том, что в ограничении уровня доступа к данным нет необходимости (здесь предполагается, что все члены структуры общие).

Заметим, что типы, созданные программистом с помощью механизма классов, часто называют **абстрактными типами данных**.

18.2. Конструкторы

В предыдущем примере инициализация объектов типа complex производилась с помощью функции-члена define ( ). При этом переменная s2 осталась неинициализированной. В C++ предусмотрены специальные функции-члены класса, которые в большинстве случаев вызываются не программистом, а компилятором и которые предназначены для инициализации объектов абстрактных типов. Такие функции называются **конструкторами**. Рассмотрим пример:

```
class cl {
int num;
public:
void set (int i) { num = i; }
void show ( ) { cout <<"Число: " << num <<'\n'; }
};
```

Чтобы использовать объект такого типа, его надо объявить, инициализировать, а затем уже использовать:

```
void f( ){
cl obj;                    // Объект создан.
obj.set (10);              // Объект инициализирован.
obj.show ( );              //Объект можно использовать.
}
```

Теперь используем для инициализации конструктор. Это просто специальная функция – член класса cl, имя которой обязательно совпадает с именем класса:

```
class cl{int num;
public:
cl ( int i ) { num = i ;}     // Конструктор.
void show ( ) { cout << "Число:" << num << '\n';}
};
```

Заметим, что для конструктора никогда не указывается тип результата!

Функция, использующая этот класс, примет вид:

```
void f( ){
cl obj(10);                // Объект создан и инициализирован!
obj.show( );               // Здесь объект obj используется!
}
```

Возможна другая, полная форма записи объявления объекта абстрактного типа, имеющего конструктор:

```
cl  obj = cl (10);
```

В этом примере конструктор – так называемая инлайн-функция (inline), так как его определение находится в теле класса. Однако его можно представить и как обычную функцию, для чего в классе конструктор только объявляется, а определяется он вне тела класса с использованием квалифицированного имени:

```
class cl{ int num ;
public:
cl ( int i );
void show( ){cout <<"Число:" << num <<'\n';}
};
cl::cl( int i ) {// Полное, или квалифицированное имя.
num = i;}
```

Часто бывает удобно предусмотреть несколько путей инициализации, используя механизм перегрузки функций.

Приведем пример программы, в которой происходит вывод строки символов на экран.

```
#include<windows.h> // Заголовочный файл для WinApi –
// совокупности различных библиотек ОС Window.
const unsigned short
Black = 0,
Blue = 1,
Green = 2,
Cyan = 3,
Red = 4,
Magenta = 5,
Brown = 6,
LightGray = 7,
DarkGray = 8,
LightBlue = 9,
LightGreen = 10,
LightCyan = 11,
LightRed = 12,
LightMagenta = 13,
Yellow = 14,
White = 15;
```

```
void strcopy(char *to, const char*from){
// Функция копирования строки
while (*to++ = *from++);
}

class mystring{
char *str;
unsigned short text, background;
int row, col;
void SetColor(unsigned short text, unsigned short background){
void* out = GetStdHandle(STD_OUTPUT_HANDLE);
```

/* Функция GetStdHandle извлекает дескриптор для стандартного ввода-вывода данных. Консольный процесс использует так называемые дескрипторы для того, чтобы обратиться к буферу ввода и вывода данных и экранным буферам консоли.*/

```
SetConsoleTextAttribute(out, (background << 4 | text));
/* Функция SetConsoleTextAttribute
устанавливает атрибуты символов,
записанных в экранный буфер консоли */
}

void gotoxy(int x, int y){
COORD coord;
coord.X = x;
coord.Y = y;

SetConsoleCursorPosition
(GetStdHandle(STD_OUTPUT_HANDLE), coord);
/* Функция SetConsoleCursorPosition устанавливает позицию курсора в
   заданном экранном буфере консоли.
   Структура COORD содержит новую позицию курсора.
*/
}

public:
mystring();
mystring(char *, unsigned short, unsigned short, int = 0, int = 0);
void write();
};
```

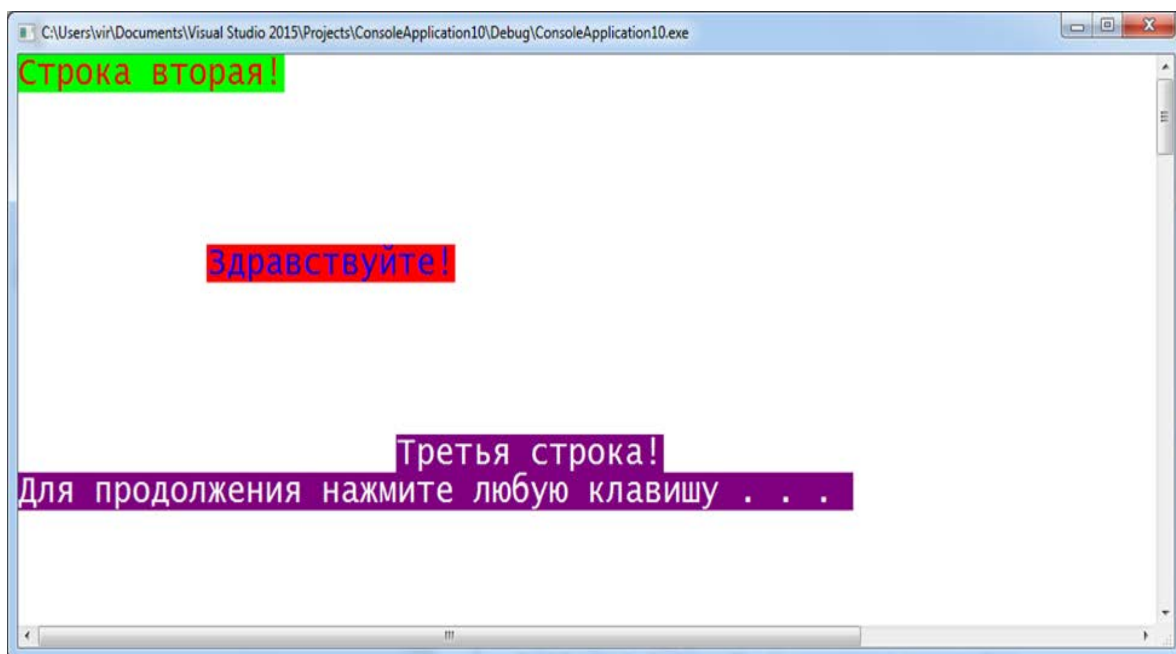
/* Конструктор без аргументов. Определяются все данные объекта – строка, видеоатрибут ее символов и позиция для вывода на экран:
*/

```
mystring::mystring() {
str = new  char[sizeof "Здравствуйте!"];
strcopy(str, "Здравствуйте!");
text =LightBlue; background=LightRed;
row = 5;
col = 10;
}
mystring::mystring(char *line, unsigned short tex, unsigned short back,
int y, int x) {
str = new char[strlen(line) + 1];
strcopy(str, line);
text = tex;
background = back;
row = y;
col = x;}
void mystring::write(){
SetColor(text, background); // установление видеоатрибута.
gotoxy(col, row);
cout<<str<<"\n";
}

void main( ){
setlocale(0, "rus");
system("color F0");  // Установка белого фона и черного текста
mystring string1;   // Эквивалентно string string1=string();
// Написать string  string(); нельзя, так как это – прототип функции!

mystring string2("Строка вторая!", LightRed,  LightGreen);
mystring string3("Третья строка!", White , Magenta, 10, 20);

// Печать строк:

string1.write ( );
string2.write ( );
string3.write ( );
}
```

В случае вызова первого конструктора – без аргументов – инициализация любого объекта будет происходить всегда совершенно одинаково, теми значениями, которые жестко определенны в этом конструкторе. В данном случае объект string1 инициализирован конструктором без аргументов и при вызове функции string.write( ) произойдет печать строки **Здравствуйте!** голубого цвета на красном фоне в 5-й строке начиная с 10-й позиции.

Объекты string2 и string3 инициализируются другим конструктором.

Результат работы программы:



Как и для других перегружаемых функций, выбор требуемого конструктора осуществляется по числу и по типу аргументов.

Отметим, что в классе может быть только один конструктор с умалчиваемыми значениями параметров.

18.3. Деструкторы

Важную роль наряду с инициализацией объектов абстрактных типов имеет и обратная к ней операция – удаление таких объектов. В частности, конструкторы многих классов выделяют память под объекты динамически, и после того, как необходимость в таких объектах отпадает, их рекомендуется удалить.

Это удобно выполнить в **деструкторе** – функции, которая вызывается для объекта абстрактного типа, когда он выходит из области существования. В рассмотренном выше примере место для хранения строк в памяти выделяется динамически, поэтому полезно определить деструктор. Имя деструктора, как и конструктора, не может быть произвольным, оно образуется символом ~ и именем класса (дополнение к конструктору):

```
class string { . . .
public:
~ string ( ){ delete str; }
. . .};
```

Здесь деструктор очень прост. Он может быть сложнее, и оформлен в виде аутлайн-функции. Деструктор никогда не может иметь аргументов. Отметим, что нельзя получить адрес ни конструктора, ни деструктора. Вызов конструктора происходит автоматически во время определения объекта абстрактного типа. Вызов деструктора происходит автоматически при выходе объекта из области своего существования. Деструктор может быть вызван и явно с обязательным указанием его полного имени. Отметим также, что для класса, в котором явно не определен ни один конструктор, компилятор самостоятельно генерирует так называемый конструктор по умолчанию, не имеющий аргументов, с уровнем доступа public. То же относится и к деструктору.

Замечание: данные класса не обязательно должны быть определены или описаны до их первого использования в принадлежащих классу функциях. То же самое справедливо и для принадлежащих классу функций, то есть обратиться из одной функции класса к другой можно до ее определения внутри тела класса. Все компоненты класса видны во всем классе.

18.4. Статические члены класса

Данное-член класса можно объявить со служебным словом static. Память под такие данные резервируется при запуске программы, то есть еще до того, как программист явно создаст первый объект данного абстрактного типа. При этом все объекты, сколько бы их ни было, используют эту заранее созданную одну-единственную копию своего статического члена. Статический член класса должен быть инициализирован после определения класса и до первого описания объекта этого класса с помощью так называемого полного или квалифицированного имени статического члена, которое имеет вид

имя_класса::имя_статического_члена.

Если статический член имеет доступ public, то его также можно использовать в программе с помощью квалифицированного имени, как и обычным образом, с помощью уточненного имени.

Пример:

Напишем класс object, в статическом члене которого хранится число существующих в каждый момент времени объектов типа object.

```
class object {
char *str;
public:
static int num_obj;
object ( char *s){ // Конструктор.
str = new char [strlen (s) + 1];
strcpy ( str, s );
cout <<"Создается " << str <<'\n'; num_obj ++ ;
}
~ object ( ){ cout <<"Уничтожается " <<str << '\n';
delete str;
num_obj – –;}
};
int object::num_obj = 0;          // Инициализация. Об этом говорит
                                  // ключевое слово int!
object s1 ("первый глобальный объект.",
       s2 ("второй глобальный объект.");
void f ( char *str ) {
object s ( str );
cout <<"Всего объектов – " << object::num_object<<".\n";
cout <<"Проработала функция f()" <<".\n";}
```

```
void main( ){
cout <<"Пока объектов – " <<object::num_obj << ".\n";
object  m ("объект в main ( ).");
cout <<"А сейчас объектов – " << m.num_obj <<".\n";
f("локальный объект.");
f("другой локальный объект.");
cout <<"Перед окончанием main() объектов – "
<<s1.num_obj<<".\n";
}
```

Результаты работы программы:

```
Создается  первый глобальный объект.
Создается  второй глобальный объект.
Пока объектов – 2.
Создается объект в  main ( ).
А сейчас объектов – 3.
Создается  локальный объект.
Всего объектов – 4.
Проработала функция  f().
Уничтожается локальный объект.
Создается  другой локальный объект.
Всего объектов – 4.
Проработала функция  f().
Уничтожается другой локальный объект.
Перед окончанием  main ( ) объектов – 3.
Уничтожается объект в main ( ).
Уничтожается второй глобальный объект.
Уничтожается первый глобальный объект.
```

Обратим внимание, что конструкторы для глобальных объектов вызываются **до** функции main(), а деструкторы **после** main( ).

Отметим, что классы, определенные внутри функции не могут иметь статических членов.

18.5. Указатель *this*

Рассмотрим пример:

```
class str{
char * string;
public:
void set( char *text){string = text;}
void write () {
cout<<”Строка:  ”<<string<<’\n’;}
};
void main(){
str  str1, str2;
str1.set((char*)"Привет!");
str2.set((char*)"Hello!");
str1.write ();
str2.write ();
}
```

В результате выполнения этой программы на экране появится следующее:

```
Строка:  Привет!
Строка:  Hello!
```

Зададимся вопросом: как функция-член write узнает, для какого именно объекта она вызвана? Функция-член определяет, для какого объекта она вызвана потому, что ей в качестве неявного первого аргумента передается адрес этого объекта. В данном случае – указатель типа str*.

Внутри функции-члена класса можно явно использовать этот указатель. Он всегда имеет имя **this** (ключевое слово).

Перед началом выполнения кода функции указатель this инициализируется значением адреса объекта, для которого вызвана данная функция-член. Таким образом, приведенное выше определение функции str**::**write() представляет собой сокращенную форму следующей записи:

```
void write(){
cout <<”Строка  :”<<this ->string<<’\n’;}
```

Отметим, что явное присваивание указателю this некоторого значения запрещено.

18.6. Статические функции-члены

Перед объявлением функции-члена класса можно поставить служебное слово static. Особенностью таких статических функций-членов является следующее: как и к статическому данному-члену класса, к ней можно обратиться еще до того, как в программе создан первый объект такого класса. Статические функции-члены (компонентные функции) позволяют получить доступ к частным статическим данным-членам класса, не имея еще ни одного объекта данного типа в программе.

Для статической компонентной функции не определен указатель this. Когда это необходимо, адрес объекта, для которого вызывается статическая функция-член, должен быть передан ей явно в виде аргумента.

Пример:

```
class prim{
int numb;
static stat;
public:
prim (int i) {
numb=i;
}
/* Далее – статическая функция. Указатель this не определен и выбор объекта осуществляется по явно переданному указателю.
Член stat не требует указателя на объект, так как он общий для всех объектов класса prim.
*/
static void func (int i, prim *p = 0) {
if (p) p-> numb = i;
else stat = i;
}
static void show ( ){
/* Статическая функция обращается только к статическому члену класса, никаких указателей не требуется: */

cout<<"stat="<<stat<<'\n';
}
};                    // Конец класса prim.

int prim::stat = 8;   // Инициализация статического члена класса.
```

```
void main(){
/* До создания объектов типа  prim  возможен единственный
способ обращения к статической функции-члену: */
prim::show ();
// Можно изменить значение статического члена класса:
prim::func(10);
/* После создания объекта типа  prim  можно обратиться к статической функции обычным для абстрактных типов способом: */
prim obj(23);              // obj.numb становится равным 23.
obj.show();
// Можно изменить значение созданного объекта:
prim::func(20, &obj);      // obj.numb равно 20.
obj.func(27, &obj);        // obj.numb равно 27.
}
```

18.7. Указатели на члены класса

Для членов класса (кроме битовых полей) определена операция получения адреса. Указатели на данные-члены класса никаких особенностей не имеют. Особенностью указателя на функцию-член класса является явное присутствие в его объявлении имени класса, за которым следует **::**.

```
class cl {. . .
public:
int f (char*, int);
void g();
. . .
};
```

Как и для указателя на обычную функцию, при объявлении указателя на компонентную функцию необходимо объявить типы результата и аргументов функции, на которую заводится указатель. Как обычно, указатель можно инициализировать при объявлении:

```
int (cl ::*fptr) (char *, int) = cl::f;
```

Пример:

```
struct s {int  mem;
s (int a){mem=a;}
void func(int a){cout<<a+mem<<'\n';}
};

void main(){
void (s::*fp)(int) = s::func;
s  obj(5);
s *p = &obj;
                // Два варианта вызова функции-члена по указателю –
(obj.*fp)(6);          // используя объект   obj  типа s
 (p->*fp)(9);          // и указатель   p   на него.
}
```

Здесь **.*** (как и **->***) являются символами одной – единственной операци, а не находящимися рядом символами двух ранее знакомых нам операций '.' ('->') и *. Правым операндом операций **.*** и **->*** обязательно должен быть **указатель на член класса**, а не любой указатель.

18.8. Инициализация данных-членов класса

Инициализация членов абстрактных типов

Пусть класс содержит в себе члены абстрактных типов. Особенностью их инициализации является то, что она выполняется с помощью соответствующего конструктора. Рассмотрим класс

```
class coord {double x, y, z;
public:
coord (){x = y = z =0;}
coord (double xv, double yv, double zv=0) { x = xv; y = yv; z = zv;}
coord (coord & c){x = c.x; y = c.y; z = c.z;}
};
class triang{
coord  vert1, vert2, vert3;  // Координаты вершин треугольника.
public:
triang ();
triang (coord &v1, coord &v2, coord &v3);
};
```

При инициализации некоторого объекта класса triang потребуется три раза вызвать конструкторы для его вершин – объектов типа coord. Для этого в определении конструктора класса triang после двоеточия нужно поместить список обращений к конструкторам класса coord:

```
Traing::triang (coord &v1, coord &v2, coord &v3):
vert1 (v1), vert2 (v2), vert3 (v3){. . .}
```

Вызов конструкторов класса coord происходит до выполнения тела самого конструктора класса triang. Порядок их вызова определяется порядком появления объявлений членов типа coord при создании класса triang.

Класс coord содержит конструктор без аргументов. Вместо записи при обращении к такому конструктору

```
triang::triang (): vert1(), vert2(), vert3() {. . .}
```

допускается написать просто так:

```
triang::triang (){. . .}
```

Инициализация констант

Если среди данных-членов класса имеются члены, описанные с модификатором const, то при инициализации используется та же форма записи конструктора, что и в случае с данными абстрактных типов:

```
class cl{ int v;
const int c;
public:
cl (int a, int b):c (b) {v=a;}
};
```

Константу можно инициализировать только в конструкторе, попытка сделать это любым другим способом (например, с помощью другой компонентной функции) приведет к сообщению об ошибке. Инициализация констант в теле конструктора тоже недопустима.

Заметим, что способ записи конструктора, обязательный для констант и данных абстрактных типов, можно использовать и для обычных членов класса:

```
class  ro{ int var; const int c;
public:
ro (int v, int u): c(u), var (v) {}
};
```

18.9. Конструктор копирования и операция присваивания

При работе с объектами абстрактных типов может возникнуть ситуация, когда один объект должен являться копией другого. При этом возможны два варианта:

1) *Вновь создаваемый* объект должен стать копией уже имеющегося.

2) Нужно скопировать один объект в другой, когда *оба были созданы ранее*.

В первом случае используется **конструктор копирования**, во втором – **операция присваивания**.

Конструктор копирования – это конструктор, первым аргументом которого является ссылка на объект того типа, в котором этот конструктор объявлен.

```
class cl{. . .
cl (cl&);      // Конструктор копирования.
. . .
};
cl  ca;        // Здесь используется конструктор без аргументов.
cl  cb = ca;   // Используется конструктор копирования.
```

Инициализация копированием происходит и при передаче функциям их аргументов и при возврате результата. Если аргумент или возвращаемое значение имеет абстрактный тип, то неявно вызывается конструктор копирования, как это было в примере с классами coord и triang.

Конструктор копирования генерируется компилятором самостоятельно, если он не был написан программистом. В этом случае создается точная копия инициализирующего объекта, что требуется далеко не всегда.

Пример1:

```
class cl {int num; float val;
public:
cl (int i, float x) {num=i; val=x;}
};
void main () {cl obj1 (10, 20.3);
```

```
// Для создания объектов  obj2  и  obj3
// используется конструктор копирования по умолчанию:
cl obj2 (obj1);
cl obj3 = obj2;
}
```

Пример2:

```
class prim{int n; float v;
public:
prim (int i, float x){n=i; v=x;}
prim (const prim &obj, int i = 0){
if (i) n=i;
else  n=obj.n;
v=obj.v; }
};
void main () {
prim obj1(10, 23.5);
/* Для создания объектов obj2 и obj3 используется явно описанный конструктор копирования: */
prim obj2 = obj1;
prim obj3 (obj1, 12);}
```

Теперь внесем минимальные изменения, и компилятор будет вынужден в дополнение к имеющемуся конструктору копирования добавить свой:

```
class prim {int n; float v;
public:
prim (int i, float x){n=i; v=x;}
prim (const prim &obj, int i){ n = i; v = obj.v; }
};
void main() {
prim obj1(10, 23.5);
/* Сейчас будет использован конструктор копирования
по умолчанию */
prim obj2=obj1;
// а сейчас будет использован явно определенный конструктор,
// причем копируется лишь часть объекта:
prim obj3(obj,12);}
```

Отметим, что модификатор const используется для предотвращения изменения копируемого объекта.

Объект одного класса можно инициализировать значением объекта другого класса. При этом конструктор не является конструктором копирования, так как в качестве аргумента в нем фигурирует ссылка на объект другого класса:

```
struct s1{int i; float x;
s1(int j, float y){i = j; x = y;}
};
struct s2{int i; float x;
s2 (const s1& a){            // Это не конструктор копирования!
i =a.i;  x=a.x;}
};

void main(){
s1 obj1 (1, 3.7);
s2 obj2 (obj1);
}
```

В отличие от конструктора копирования, операция присваивания используется тогда, когда объекты, являющиеся операндами этой операции, уже существуют. Операция присваивания, наряду с операцией получения адреса, предопределена для объектов абстрактных типов по умолчанию, и ее можно использовать без каких либо дополнительных действий со стороны программиста.

```
class cl{. . . };
void f(){
cl obj1; cl obj2 = obj1;     // Используется конструктор копирования.
cl obj3;
obj3 = obj1;                 // Присваивание!
}
```

Не всегда требуется при выполнении присваивания просто создавать копию. Если требуется нечто иное, то нужно переопределить операцию присваивания для класса.

18.10. Дружественные функции

Могут встретиться ситуации, когда желательно иметь доступ к личным данным класса, минуя функции-члены. Наиболее распространена ситуация, когда функция-член одного класса должна иметь доступ к личным членам другого.

Рассмотрим снова пример с классами coord и triang.

```
class coord {double  x, y, z;
public:
coord ();
coord (double, double, double = 0);
coord (coord & c);
};

class triang {coord  vert1, vert2, vert3;
public:
triang ();
triang (coord &v1, coord &v2, coord &v3);
};
```

Пусть нам необходимо добавить в класс triang функцию-член, вычисляющую координаты центра треугольника.
Язык предоставляет для некоторых функций, как обычных, так и членов некоторого класса X, возможность получения доступа к личным членам класса Y. Такая функция называется **привилегированной в классе** Y. Говорят также, что класс **X** является **дружественным** классу **Y**.

Для объявления привилегированной функции используется служебное слово **friend**. Чтобы функция стала привилегированной в классе Y, она должна быть объявлена в этом классе как дружественная функция.

Напишем три функции-члена класса triang, вычисляющие координаты центра треугольника по каждой из осей:

```
double triang::midx (){ return (vert1.x+vert2.x+vert3.x)/3;}
```

и аналогично triang::midy (), triang::midz ().

Для того чтобы компилятор не выдал сообщение об ошибке, необходимо добавить в объявление класса coord, в любой его части, следующие объявления:

```
class coord {
…
friend triang::midx ();
friend triang::midy ();
friend triang::midz ();
}
```

Достаточно распространенным является случай, когда все функции-члены одного класса являются привилегированными в другом; предусмотрена даже упрощённая форма записи:

```
class coord {…
friend triang;
…
};
```

или

```
class coord {…
friend class triang;
…
};
```

В этом случае говорят, что класс triang является дружественным классу coord.

Заметим, что для дружественных функций не определён указатель this, они не имеют неявных аргументов, для них не определены уровни доступа. Одна и та же функция может быть объявлена привилегированной сразу в нескольких классах.

Разницу в способе использования функций-членов и дружественных функций покажем в следующем примере:

```
class cl {int numb;
                              // f_func() – не личный член класса,
                              // хотя объявляется в private-части.
```

```
friend void f_func (cl*, int);
public:
void m_func (int);};
void f_func (cl* cpt, int i){
cptr->numb = i;             // Нужен явный указатель на объект,
                            // так как указатель  this  не определён!
}
void cl::m_func(int i){
numb = i;                   // То же, что this->numb = i;
}
void main (){
cl obj;
f_func (&obj, 10);
obj.m_func (10);
// Сравните способы вызова и аргументы функций!
…}
```

Следующий пример демонстрирует возможность доступа к статическим личным членам класса ещё до создания хотя бы одного объекта этого класса.

```
class cl {static int num;
public:
void set (int i){num = i;}
void m_show () {cout<<num<<'\n';}
friend void f_show (){cout << cl::num<<'\n';}
};

int cl::num = 8;

void main(){
cout <<"Объектов типа  cl  нет.\n";
cout <<"Статический член класса = ";
// Пока можно использовать только дружественную функцию:
f_show ();
cl obj;
obj.set (200);
cout <<"Создан объект типа cl.\n";
cout <<"Статический член класса = ";
// Теперь можно использовать и функцию-член.
obj.m_show ();
}
```

18.11. Конструктор и операция *new*

Если абстрактный тип имеет конструктор без аргументов, то обращение к операции new полностью совпадает с тем, что используется для выделения памяти под обычные типы данных без инициализирующего выражения.

```
class integer {int i;};
void main () {integer *ptr = new integer; . . .}
```

Если же конструктор класса integer имеет аргументы, то список аргументов помещается там же, где при работе со стандартными типами данных находится инициализирующее выражение.

```
class integer {int i;
public:
integer ();
integer (int j): i(j) { }
};

void main(){
int *ip = new int (10);
integer *iptr = new integer (30);
}
```

Если в операции new происходит обращение к конструктору без аргументов, то допустимы обе следующие формы записи:

```
integer *ip1 = new integer ();
integer *ip2 = new integer;
```

Если конструктор без аргументов для класса X не определён, то при попытке выполнить оператор

```
X *xp = new X;
```

компилятор выдаст сообщение об ошибке. В этом случае требуется явно определить конструктор без аргументов.

18.12. Вызов деструктора

Вызов деструктора для объекта абстрактного типа производится автоматически при его выходе из области существования. Для локальных переменных деструктор вызывается при выходе из блока, в котором эта переменная была объявлена. Для глобальных переменных вызов деструктора является частью процедуры завершения программы, выполняемой после функции main(). Выход указателя на объект абстрактного типа из области существования этого указателя не приводит к вызову деструктора для объекта, на который он указывает. Надо различать указатели на объект, созданные при помощи операции new, и другие объекты.

Рассмотрим пример с указателем на автоматический объект абстрактного типа:

```
class cl{int num;
public:
cl (int i){
num = i;}
~cl (){}
};

void main (){
// Создание объекта obj типа cl:
cl obj (1);
// Создание указателя ptr на объект класса cl и его инициализация
// адресом, создаваемой здесь же безымянной переменной типа cl:

cl *ptr = &cl (2);
{// Указатель в блоке относится к тому же объекту, что и ptr.
cl *tmp = ptr;
}
}
```

В этом случае как конструктор, так и деструктор будут вызываться дважды. Сначала вызывается конструктор для объекта obj, затем конструктор для безымянного объекта, на который указывает ptr. При выходе из внутреннего блока указатель tmp теряется, однако сам объект сохраняется. При завершении main () в первую очередь вызывается деструктор для безымянного объекта, а затем – деструктор для obj.

Если указатель относится к объекту абстрактного типа, созданному динамически, то деструктор для него вызывается в операции delete:

```
class cl{int num;
public:
cl (int i){num = i;}
~cl (){cout<<”Деструктор класса cl. \n ”;
};
void main (){
cl *ptr = new cl (1);
…
delete ptr;}
```

Перед удалением из памяти объекта, на который указывает ptr, для него будет вызван деструктор. В результате на экране появится строка

```
деструктор класса cl.
```

Вызов деструктора можно осуществить явно по его полному имени:

```
class cl{int num;
public:
cl (int i){
num = i;}
~cl(){}
};
void main(){cl obj (1);
cl *ptr = &cl (2);
obj.cl::~cl ();
ptr -> cl::~cl ();}
```

Также можно вызвать деструктор и для динамического объекта.

```
cl *ptr = new cl (1);
ptr -> cl::~cl ();
```

Отмети, что явный вызов деструктора не отменяет его автоматический вызов.

```
class X{ int *ip;
public:
X(int y){
ip = new int(y);}
~X(){cout <<"Деструктор класса X;\n";
delete ip;
}
};

void main(){
X *xp = new X(5);
xp->X::~X();        // Явный вызов деструктора.
delete xp;          // Вызов деструктора из  delete.
}
```

В результате получим два сообщения, если не произойдёт зацикливания при повторной операции delete, применённой к одному и тому же указателю ip:

```
Деструктор класса X;
Деструктор класса X;
```

Пользоваться явным вызовом деструктора надо очень осторожно.

19. Производные классы

19.1. Построение производного класса

Рассмотрим класс с конструктором и деструктором:

```
class Base {
int *bmember;
public:
Base (int arg = 0){bmrmber = new int(arg);}
~Base (){delete bmember;}
};
```

Предположим, что нам нужно изменить этот класс так, чтобы объект такого типа содержал не один, а два указателя. Вместо того, чтобы изменять класс Base, можно поступить иначе. Ничего не меняя в Base, построить на его основе новый класс Derived:

```
class Derived: public Base{
int *dmember;
public:
Derived (int arg){
dmember = new int(arg); }
~Derived (){ delete dmember; }
};
```

Запись вида **class Derived: public Base** говорит о том, что класс Derived является таким заново создаваемым классом, который построен на основе класса Base. При этом класс Derived **наследует** все свойства класса Base. Говорят, что Derived является классом, **производным** от класса Base, а класс Base является **базовым** классом для Derived.

Если в программе будет создан объект типа Derived, то он будет содержать два указателя на две области динамической памяти – bmember, как подобъект типа Base и dmember. Процесс создания объекта типа Derived будет проходить в два этапа: сначала будет создан "подобъект" типа Base, причём это сделает конструктор класса Base. Затем будет выполнен конструктор класса Derived. Вызов деструкторов осуществляется в обратном порядке. Поскольку конструктор класса Base может требовать наличия одного аргумента при обращении к нему, то этот аргумент необходимо передать. Чтобы передать список аргументов конструктору базового класса, этот список должен быть помещён в определении конструктора производного класса, подобно тому,

как это делалось при инициализации данных абстрактного типа, являющихся членами некоторого класса:

```
Derived::Derived (int arg): Base (arg) {
dmember = new int (arg);
}
```

Если конструктор базового класса не имеет аргументов или использует аргументы по умолчанию, помещать пустой список в конструктор производного типа не надо.

19.2. Защищенные члены класса

Для регулирования уровня доступа к членам классов используются служебные слова public и private. Для этих же целей введено ключевое слово **protected** (защищенный).

Если класс A не служит базовым ни для какого другого класса, то его защищенные члены ничем не отличаются от личных – доступ к ним имеют только функции-члены данного класса и дружественные этому классу функции. Если же класс B является производным от A, то пользователи как класса A, так и B по-прежнему не имеют доступа к защищенным членам, но такой доступ могут иметь функции-члены класса B и функции, привилегированные в B:

```
class Base{
private:
int privatem;
protected:
int protectedm;
};
class Derived: public Base{
memberF (){
cout<<privatem;            // Ошибка!
cout<<protectedm;          // Верно.
}
};

void main(){
Base  b;
cout<<b.protectedm;              //Ошибка!
Derived  d;
cout<<d.protectedm;              //Ошибка.
}
```

19.3. Управление уровнем доступа к членам класса

В предыдущих примерах базовый класс являлся общим базовым классом для производного класса:

```
class Derived: public Base{…};
```

Это означает, что уровень доступа к членам класса Base для функций-членов класса Derived и просто пользователей класса Derived остался неизменным: личные члены класса Base не доступны в классе Derived, общие и защищенные члены класса Base остались соответственно общими и защищенными в Derived. Если не указать, что базовый класс является общим, то по умолчанию он будет личным:

```
class  Derived: Base{…};         // Base – личный базовый класс.
```

Если базовый класс является личным базовым классом, то его личные члены по-прежнему недоступны ни в производном классе, ни для пользователя производного класса, а защищенные и общие члены базового класса становятся личными членами производного класса.

Базовый класс не может быть защищенным базовым классом.

Если базовый класс является личным, то для некоторых его членов в производном классе можно восстановить уровень доступа базового класса. Для этого их полное имя приводится в соответствующей части определения класса:

```
class Base {
private: int privm;
protected: int protm;
public: int pubm;
};

class Derived: Base{// Личный базовый класс.
public:
Base::pubm;  // Теперь pubm – общий член класса Derived;
Base::protm; // ошибка – изменение уровня доступа.
protected:
Base::protm; //Теперь protm – защищенный член класса Derived;
Base::pubm;  // ошибка – изменение уровня доступа.
```

Структуры могут использоваться подобно классам, но с одной особенностью. Если производным классом является структура, то ее базовый класс всегда является общим базовым классом, т.е. объявление вида:

```
struct B: A{…};
```

Эквивалентно

```
class B: public A{…};
```

Если же производный класс строится на основе структуры, все происходит точно так же, как и при использовании в качестве базового обычного класса. Таким образом, если и базовым, и производным классами являются структуры, то запись вида:

```
struct B: A{…};
```

эквивалентна

```
class B: public A {public: …};
```

19.4. Последовательность вызова конструктора и деструктора при построении производного класса на основе одного базового

Объект производного класса в качестве данных-членов класса, может содержать объекты абстрактных типов.

```
class string{. . .
public:
string (char*);
~string ();
…
};

class Base{…
public:
Base (int);
~Base ();
. . .
};
class Derived: public Base {
Base b;
string s;
```

```
public:
Derived (char*, int);
~Derived ();
. . .
};
```

Перед обращением к собственно конструктору класса Derived необходимо, во-первых, создать подобъект типа Base, во-вторых, члены b и s. Поскольку для их создания нужно обратиться к конструкторам соответствующих классов, мы должны им всем передать необходимые списки аргументов:

```
Derived::Derived (char *st, int len): Base (len), b (len+1), s (str) {…}
```

В этом случае при создании объекта типа Derived сначала будет создан подобъект типа Base. При этом будет вызван конструктор Base::Base() с аргументом len. Затем будут созданы объекты b и s в том порядке, в котором они указаны в определении класса Derived. После этого будет выполнен конструктор Derived::Derived(). Деструкторы будут вызваны в обратном порядке.

19.5. Преобразования типов

Объект производного типа может рассматриваться как объект его базового тип. Обратное неверно. (Кошка есть млекопитающее, но не любое млекопитающее есть кошка.) Компилятор может неявно выполнить преобразование объекта производного типа к объекту типа базового:

```
class Base {…};
class Der: public Base {…};
Der derived;
Base b = derived;
```

Обратное преобразование – Base к Der – должно быть определено программистом:

```
Der tmp = b;         // Ошибка, если для  Der
                     // не определён конструктор Der (Base).
```

Значительно чаще, чем преобразование типов, используется преобразование указателей на эти типы. Существует два типа преобразования указателей – явное и неявное. Явное преобразование будет выполнено всегда, неявное – только в определённых случаях.

Если базовый класс является общим (public) базовым классом, т.е. мы имеем дело с отношением вида

```
class Base {…};
class Der: public Base {…};
```

то принципы преобразования очень просты:

неявно может быть выполнено преобразование указателя типа Der* к указателю типа Base*. Обратное преобразование обязательно должно быть явным.

Другими словам, при обращении через указатели объект производного типа может рассматриваться как объект базового тип. Обратное утверждение неверно:

```
Der derived;
Base *bp = &derived;      // Неявное преобразование.
Der *dp1 = bp;            // Ошибка.
Der *dp2 = (Der*) bp;     // Явное преобразование; теперь верно.
```

То, что производный класс в некотором смысле может рассматриваться как его базовый, оказывает влияние на выбор нужной версии перегруженной функции. Сложность возникает, если для выбора одного из вариантов нужно выполнить неявное преобразование типов.

Правила здесь таковы.

Если нет точного соответствия списка формальных и фактических параметров, то наивысший приоритет среди выполняемых преобразований имеют преобразования производного типа к базовому. Это относится как к самому типу, так и к указателю на него. Только в том случае, если это невозможно, компилятор пробует выполнить другие преобразования (например, стандартные преобразования указателей).

Пример:

```
class Base {…};
class Der: public Base {…};
func (Base*);
func (void*);
…
Der *dp;
float *fp;
func (dp);          // Вызов func (Base*).
func (fp);          // Вызов func (void*).
```

Если в программе используются несколько уровней производных классов, то при выполнении неявных преобразований типа указателя ищется класс “ближайшего” уровня:

```
class Base {…};
class A: public Base {…};
class B: public A{…};
func (Base*);
func (A*);
…
B *db;
func (db);          // Вызов функции func (A*).
```

20. Полиморфизм

Одно из самых коротких и выразительных определений полиморфизма таково: полиморфизм – это функциональная возможность, позволяющая старому коду вызвать новый. Это свойство дает возможность расширять и совершенствовать программную систему, не затрагивая существующий код. Осуществляется такой подход с помощью механизма виртуальных функций.

20.1. Раннее и позднее связывание

К механизму виртуальных функций обращаются в тех случаях, когда в базовый класс необходимо поместить функцию, которая должна по-разному выполняться в производных классах. Точнее, по-разному должна выполняться не единственная функция из базового класса, а в каждом производном классе требуется свой вариант этой функции.

Предположим, необходимо написать функцию-член CalculatePay() (Расчет), которая подсчитывает для каждого объекта класса Employee (Служащий) ежемесячные выплаты. Все просто, если зарплата рассчитывается одним способом: можно сразу вставить в вызов функции тип нужного объекта. Проблемы начинаются с появлением других форм оплаты. Допустим, уже есть класс Employee, реализующий расчет зарплаты по фиксированному окладу. А что делать, чтобы рассчитать зарплату контрактников – ведь это уже другой способ расчета! В процедурном подходе пришлось бы переделать функцию, включив в нее новый тип обработки, так как в прежнем коде такой обработки нет. Объектно-ориентированный подход благодаря полиморфизму позволяет производить различную обработку.

В таком подходе надо описать базовый класс Employee, а затем создать производные от него классы для всех форм оплаты. Каждый производный класс будет иметь собственную реализацию метода CalculatePay().

Другой пример: базовый класс figure может описывать фигуру на экране без конкретизации её вида, а производные классы triangle (треугольник), ellipse (эллипс) и т.д. однозначно определяют её форму и размер. Если в базовом классе ввести функцию void show () для изображения фигуры на экране, то выполнение этой функции будет возможно только для объектов каждого из производных классов, определяющих конкретные изображения. В каждый из производных классов нужно включить свою функцию void show() для формирования изображения

на экране. Доступ к функции show() производного класса возможен с помощью явного указания ее полного имени, например:

```
triangle::show ();
```

или с использованием конкретного объекта:

```
triangle t;
t.show();
```

Однако в обоих случаях выбор нужной функции выполняется при написании исходного текста программы и не изменяется после компиляции. Такой режим называется ранним или статическим связыванием.

Большую гибкость, особенно при использовании уже готовых библиотек классов, обеспечивает так называемое позднее, или отложенное, или динамическое связывание, которое предоставляется механизмом виртуальных функций.

20.2. Виртуальные функции

Рассмотрим сначала, как ведут себя при наследовании не виртуальные функции-члены с одинаковыми именами, сигнатурами и типами возвращаемых значений.

```
struct base {
void fun (int i) {
cout <<”base::i = ” << i << ’\n’;}
};

struct der: public base {
void fun (int i) {
cout << ” der::i = ” <<  i << ‘\n‘;}
};


void main () {
base B,         *bp = &B;
der D,          *dp = &D;
base *pbd = &D;   // Неявное преобразование от  der*  к  base*.
bp->fun (1);
dp->fun (5);
pbd->fun (8);
}
```

Результат:

```
base::i = 1
der::i = 5
base::i = 8
```

Здесь указатель pbd имеет тип base*, но его значение – адрес объекта D класса der. При вызове функции-члена по указателю на объект выбор функции зависит **только от типа указателя**, но не от его значения, что и иллюстрируется выводом base::i = 8. Настроив указатель базового класса на объект производного класса, **не удается** с помощью этого указателя вызвать функцию из производного класса. Таким способом не удается достичь **позднего** или **динамического** связывания.

Динамическое связывание обеспечивается механизмом виртуальных функций. Любая нестатическая функция базового класса может быть сделана виртуальной, если в ее объявлении использовать спецификатор **virtual**:

```
class base {
public:
int i;
virtual void print (){
cout << i << “  внутри base\n“;}
};

class D: public base{
public:
void print (){
cout << i << “  внутри D\n“;}
};

void main (){
base b;
base *pb = &b;
D f;
f.i = 1 + (b.i = 1);
pb->print ();
pb = &f;               // Неявное преобразование D*  к  Base*.
pb->print ();
}
```

Результат:

```
1  внутри  base
2  внутри  D
```

Здесь в каждом случае выполняется различная версия функции print(). Выбор динамически зависит от типа объекта, **на который указывает указатель**. Служебное слово virtual означает, что функция print() может иметь свои версии для различных порожденных классов. Указатель на базовый класс может указывать или на объект базового класса, или на объект порожденного класса. Выбранная функция-член зависит от класса, на объект которого указывается, но не от типа указателя. При отсутствии члена производного типа по умолчанию используется виртуальная функция базового класса. Отметим различие между выбором соответствующей переопределенной виртуальной функции и выбором перегруженной функции-члена (не виртуальной): перегруженная функция-член выбирается во время компиляции алгоритмом, основанным на правиле сигнатур. При перегрузке функции-члены могут иметь разные типы возвращаемого значения. Если же функция объявлена как virtual, то все её переопределения в порожденных классах должны иметь одну и ту же сигнатуру и один и тот же тип возвращаемого значения. При этом в производных классах слово virtual можно и не указывать.

В производном классе нельзя определять функцию с тем же именем и с той же сигнатурой, но с другим типом возвращаемого значения, чем у виртуальной функции базового класса. Отметим, что конструктор не может быть виртуальным, а деструктор – может.

Рассмотрим пример.

Вычисление площадей различных фигур.
Различные фигуры будем порождать от базового класса figure.

```
class figure {
protected:
double x, y;
virtual double area (){return 0;}          // Площадь по умолчанию.
};

class rectangle: public figure {
private:
double height, width;
. . .
```

```
public:
rectangle (double h, double w){height=h; width=w;}
double area () {return height * width;}
. . .
};
class circle: public figure {
private:
double radius;
. . .
public:
circle (double r){radius=r;}
double area () {
return  M_PI*radius*radius;}
. . .
};
```

Код пользователя может выглядеть так:

```
const int N = 30;
figure *p[N];
double tot_area = 0;
. . .              // Здесь устанавливаются указатели p[i], например,
. . .              // rectangle r(3, 5); p[0]=&r; circle c(8); p[1]=&c; и т.д.
for (int i = 0; i < N; i ++)
tot_area + = p[i]->area ();        // Код пользователя.
```

Главное преимущество состоит в том, что код пользователя не нуждается в изменении, даже если к системе фигур добавляются новые.

Рассмотрим еще пример для расчета заработной платы с классом Employee.

```
#include <iostream>
#include <fstream.h>
#include <conio.h>
#include <string.h>

class Employee{
protected:
char * firstName, * lastName;   // Имя, фамилия.
int age;                        // Возраст.
double payRate;                 // Размер оплаты.
public:
```

```
Employee (char* FN, char* LN, int a, double pay){
firstName = new char [strlen (FN) + 1];
strcpy(firstName, FN);
lastName = new char [strlen (LN) + 1];
strcpy(lastName, LN);
age = a;
payRate = pay;
}
virtual double CalculatePay (){
return 0;
}
void print(){cout<<firstName<<"  "<<lastName<<" this month has re-
ceived  ";}
virtual ~Employee (){};
};
class HourlyEmployee: public Employee {    // Почасовая оплата.
int hours;                                 // Проработано часов.
public:
HourlyEmployee (char* FN, char* LN, int a, double pay, int h):
Employee (FN, LN, a, pay){ hours=h; }
virtual ~HourlyEmployee (){delete firstName; delete firstName;}
virtual double CalculatePay (){
return hours*payRate;
}
};

class ContractorEmployee: public Employee { // Работа по
контракту.  public:
ContractorEmployee (char* FN, char* LN, int a, double pay):
Employee (FN, LN, a, pay){}
virtual double CalculatePay (){
return payRate;
}
virtual ~ContractorEmployee (){delete firstName; delete firstName;}
};

class DaypaymentEmployee: public Employee { // Поденная оплата.
int days;                                  // Отработано дней.
public:
DaypaymentEmployee (char* FN, char* LN, int a, double pay, int d):
Employee (FN, LN, a, pay){days=d;}
```

```
virtual double CalculatePay (){
return days*payRate/24.0;                  // Рабочих дней в месяце – 24.
}
virtual ~DaypaymentEmployee (){delete firstName; delete firstName;}
};
void loademploee (Employee* a[], int &n){
char FN[20], LN[20]; int age, arg; double pay;
char sel;                          // Селектор, задающий тип оплаты.
ifstream file("emp.txt");  // Создаем входной поток для чтения
                           // file  и связываем его с внешним
                           // файлом  emp.txt.
n = 0;
while ( file.peek ( ) != EOF ){   // Пока нет конца файла …
file >> sel;
file >> FN;
file >> LN;
file >> age;
file >> pay;
file >> arg;
switch (sel){
case 'h': a[n] = new HourlyEmployee      (FN, LN, age, pay, arg);
break;
case 'c': a[n] = new ContractorEmployee (FN, LN, age, pay);
break;
case 'd': a[n] = new DaypaymentEmployee (FN, LN, age, pay, arg);
break;
}
n++;
}
}
void main(){
int n;
Employee* a[20];
clrscr();
loademploee (a, n);
double s=0, r;
for(int i=0; i<n; i++){
s+=(r=a[i]->CalculatePay ());
a[i]->print();
cout.width (16); cout << r << "$\n";
}
```

```
cout<<"This month it is payd: ";
cout.width (16); cout << s << "$\n";
}
```

Пусть во входном файле emp.txt содержится следующая информация:

c	Jbanov	Ivan	32	4340	0
c	Muhin	Sergey	26	1320	0
h	Mazin	Petr	27	15.3	32
d	Bobrov	Mikhail	40	110	21

Тогда в результате работы программы на экране появится следующее:

```
Jbanov  Ivan this month has received            4340$
Muhin  Sergey this month has received            1320$
Mazin  Petr this month has received             489.6$
Bobrov  Mikhail this month has received          96.25$


This month it is payd:         6245.85$
```

20.3. Абстрактные классы

Снова рассмотрим пример с вычислением площадей фигур. В этой программе использована виртуальная функция area(). Эта функция должна была быть первый раз определена в классе figure. Поскольку при нормальной работе не должно существовать объектов типа figure, а только объекты производных от него типов, то версия area () была определена так:

```
double figure::area {return 0;}
```

Если бы тип возвращаемого значения у функции был void (например, при рисовании фигуры void show ( )), можно было бы написать:

```
void figure::show () {}
```

В обоих случаях эти функции фиктивны. Такого рода виртуальные функции можно было бы использовать для контроля ошибок, связанных с созданием объектов типа figure:

```
double figure::area () {
cout <<"Ошибка: попытка вычислить площадь ";
cout <<"несуществующего объекта!\n";
exit (1);
return 1;
}
```

В C++ существует более удобный и надежный способ. Версия виртуальной функции, которая, с одной стороны, должна быть определена, а с другой, никогда не должна использоваться, может быть объявлена как **чисто виртуальная функция**:

```
class figure {. . .
virtual double area () = 0;
};
```

В классах, производных от figure, при наличии своей версии виртуальной функции area () она должна либо быть определена, либо, в свою очередь, объявлена как чисто виртуальная функция. Во время выполнения программы при обращении к чисто виртуальной функции выдается сообщение об ошибке и программа аварийно завершается. Класс, содержащий хотя бы одну чисто виртуальную функцию, называется **абстрактным** классом. Запрещено создание объектов таких классов. Это позволяет установить контроль со стороны компилятора за ошибочным созданием объектов фиктивных типов, подобных figure. Заметим, что можно создавать указатели и ссылки на абстрактные классы.

21. Переопределение стандартных операций

21.1. Основные определения и свойства

В С++ есть возможность распространения действия стандартных операций на операнды абстрактных типов данных.

Для того, чтобы переопределить одну из стандартных операций для работы с операндами абстрактных типов, программист должен написать функцию с именем

operator α,

где α – обозначение этой операции (например, + – | += и т.д.).

При этом в языке существует несколько ограничений:

- нельзя создавать новые символы операций;
- нельзя переопределять операции
  :: * (– разыменование, не бинарное умножение) ?:
  sizeof **.** **.*** # ##
- символ унарной операции не может использоваться для переопределения бинарной операции и наоборот. Например, символ << можно использовать только для бинарной операции, ! – только для унарной, а & – и для унарной, и для бинарной;
- переопределение операций не меняет ни их приоритетов, ни порядка их выполнения (слева направо или справа налево);
- при переопределении операции компьютер не делает никаких предположений о ее свойствах. Это означает, что если стандартная операция += может быть выражена через операции + и =, т.е. a + = b эквивалентно a = a + b, то для переопределения операций в общем случае таких соотношений не существует, хотя, конечно, программист может их обеспечить. Кроме того, не делается предположений, например, о коммутативности операции +: компилятор не имеет оснований считать, что a + b, где a и b – абстрактных типов – это то же самое, что и b + a;
- никакая операция не может быть переопределена для операндов стандартных типов.

Функция operator α() является обычной функцией, которая может содержать от 0 до 2 явных аргументов. Она может быть, а может и не быть функцией-членом класса.

```
class cl { int i;
public:
int get () {return i;}
int operator + (int );              // Бинарный плюс.
};
int operator + (cl&, float);        // Бинарный плюс.
```

В первой форме бинарного плюса не один, а два аргумента. Первый – неявный. Его имеет любая нестатическая функция-член класса; этот аргумент является указателем на объект, для которого она вызвана. Реализация обеих функций может выглядеть так:

```
int cl::operator + (int op2){
return  i + op2;}
int operator + (cl &op, float op2){
return  op.get() + op2;}
```

Что будет, если в глобальной функции ::operator+() второй аргумент будет иметь тип не float, а int? В этом случае компилятор выдаст сообщение об ошибке, так как он не сможет сделать выбор между функциями cl::operator + () и ::operator +() – обе подходят в равной степени.

Для выполнения переопределенной унарной операции α**x** (или **x**α), где x – объект некоторого абстрактного типа Class, компилятор пробует найти либо функцию Class::operator α(void), либо ::operator α(Class). Если найдены одновременно оба варианта, то фиксируется ошибка.

Интерпретация выражения осуществляется либо как x.operator α (void), либо как operator α(x).

Для выполнения переопределенной бинарной операции **x** α **y**, где **x** обязательно является объектом абстрактного типа Class, компилятор ищет либо функцию Class::operator α(type y), либо функцию ::operatr α(Class, type y), причем type может быть как стандартным, так и абстрактным типом.

Интерпретируется выражение **x** α **y** либо как

x.operator α(y), либо как operator α (x, y).

Как для унарной, так и для бинарной операции число аргументов функции operator α() должно точно соответствовать числу операндов этой операци. Заметим, что часто удобно передавать значения параметров в функцию operator α () не по значению, а по ссылке.

Рассмотрим для примера операцию сложения, определенную над классом “комплектное число“:

```
class complex {
double re, im;
public:
double & real (){return re;}
double & imag(){return im;}
//. . .
};
complex operator + (complex a, complex b){
complex result;
result.real () = a.real () + b.real ();
result.imag () = a.imag () + b.imag ();
return result;}
```

Здесь оба аргумента функции operator + () передаются по значению, то есть выполняется копирование четырех чисел типа double. Подобные затраты могут оказаться слишком накладными, особенно если операция переопределяется над таким, например, классом, как “матрица“.

Можно было бы попытаться избежать накладных расходов, передавая по значению не сами объекты, а указатели на них:

```
complex operator + (complex* a, complex *b){. . .}
```

Но так поступать нельзя, так как оба аргумента теперь являются объектами стандартного типа – указателями, а переопределение операций для стандартных типов запрещено.

В этой ситуации необходимо использовать ссылки – они не изменяют тип операндов, а только влияют на механизм передачи параметров:

```
complex operator + (complex &a, complex &b){
complex  result;
result.real () = a.real () + b.real ();
result.imag () = a.imag () + b.imag ();
return result;
}
```

Тело функции operator + () при этом не изменилось.

Пример: определение операции + для класса stroka:

```
class stroka {
char *c;                        // Указатель на строку.
int len;                        // Длина строки.

public:
stroka (int N = 80): len (0) // Строка, не содержащая информацию;
{i = new char[N +1];         // выделение памяти для массива.
c[0] = '\0';
}  // Конструктор выделяет память для строки и делает ее пустой.

stroka (const char * arg){
len = strlen (arg);
c = new char [len + 1];
strcpy (c, arg);
}

int & len_str ()             // Возвращает ссылку на длину строки.
{return len;
}

char * string( )             // Возвращает указатель на строку.
{return c;}

void display()               // Печать информации о строке.
{cout << "Длина строки: "<< len << ".\n";
 cout << "Содержимое строки: " << c << ".\n";
}

~ stroka(){delete c;}
};

stroka & operator + (stroka &a, stroka &b){
int ii = a.len_str() + b.len_str();         // Длина строки – результата.
stroka * ps = new  stroka (ii);
strcpy (ps->string (), a.string ());        // Копирует строку из  a;
strcat ( ps->string (), b.string ());       // присоединяет строку из  b;
ps->len_str() = ii;          // записывает значение длины строки;
return *ps;                  // возвращает новый объект  stroka.
}
```

```
void main () {
stroka X (“Вася“);
stroka Y (“  едет“);
stroka Z;
Z = X + Y + “  на велосипеде“;
Z.display ();
}
```

Результат выполнения программы:

```
Длина строки: 23.
Содержимое строки: Вася едет на велосипеде.
```

Заметим, что вместо Z = X+ Y + “на велосипеде“ возможна и такая форма обращения к operator +():

```
Z = operator + (X, Y);
Z = operator + (Z, “  на велосипеде“);
```

21.2. Операции *new* и *delete* при работе с абстрактными типами

Операции new и delete реализуются через функции и вне зависимости от того, описаны или нет operator new() и operator delete как static, они всегда являются статическими функциями. Операция new предопределена для любого типа, в том числе и для абстрактного типа, определенного через механизм классов. Можно переопределять как глобальную функцию operator new (), так и функцию class x::operator new(). Глобальные new и delete переопределяются обычным образом через механизм соответствия сигнатур.

Как и при переопределении глобальной функции operator new(), переопределенная функция classX::operator new() должна возвращать результат типа void*, а ее первый аргумент должен иметь тип size_t (то есть unsigned), в которой хранится размер выделяемой памяти. Замети, что при использовании операции new этот аргумент не указывается, а размер необходимого участка памяти вычисляется автоматически исходя из указанного типа.

21.3. Использование *new* при создании динамического объекта абстрактного типа

Рассмотрим фрагмент:

```
class C{ . . .
public:
C(int arg ){ . . .}
};
. . .
C *cp = new C(3);
```

Создание динамического объекта типа C можно разбить на две стадии:

Собственно создание объекта – это выполняет конструктор.

Размещение этого объекта в определенной области памяти – это делает операция new.

При этом вначале выполняется функция operator new(), а затем уже конструктор размещает создаваемый объект в выделенной памяти.

Операцию new можно переопределить:

```
class cl{ . . .
public:
cl() {cout << "Конструктор класса cl.\n";}
void* operator new (unsigned);
};
void* cl::operator new (unsigned size){
cout <<"Функция operator new () класса cl;\n";
void* p = new char [size ];        // Глобальная new!
if (p) return p; else {
cout << "Нет памяти для объекта типа cl!;\n"; exit(1);}
}

void main (){
cl * cp = new cl;}
```

Результат:

```
Функция operator new () класса cl;
Конструктор класса cl.
```

21.4. Операция *delete*

Выполнение операции delete применительно к указателю на объект абстрактного типа приводит к вызову деструктора для этого объекта.

```
cl * clp = new cl (5);        // Вызов конструктора cl (5);
. . .
delete clp;    // вызов деструктора ~ cl () перед освобождением
               // динамической памяти.
```

Функцию x::operator delete () можно переопределить в классе x, причем она может иметь только две формы:

```
void operator delete (void * );
void operator delete (void *, size_ t );
```

Если присутствует вторая форма данной операци, то компилятор использует именно ее.

21.5. Преобразование типов

Преобразование типов можно разделить на 4 группы:

1) стандартный к стандартному;
2) стандартный к абстрактному;
3) абстрактный к стандартному;
4) абстрактный к абстрактному.

Первые преобразования уже были нами рассмотрены. Преобразования второй группы основаны на использовании конструкторов – как явном, так и неявном.

Снова рассмотрим касс complex:

```
class complex {
double re, im;
public: complex (double r = 0, double i = 0){ re = r;  im = i ; }
. . .
};
```

Объявления вида

```
complex c1;
complex c2 (1.8);
complex c3 (1.2, 3.7);
```

обеспечивают создание комплексных чисел.

Но конструктор может вызываться и неявно, в том случае, когда в выражении должен находиться операнд типа complex, а на самом деле присутствует операнд типа double:

```
complex operator + (complex  & op, complex & op2 );
complex operator – (complex  & op, complex & op2 );
complex operator * (complex  & op, complex & op2 );
complex operator  /  (complex  & op, complex & op2 );
complex operator –  (complex  & op );          // Унарный минус.
complex res;
res = – (c1 + 2) * c2 / 3 + .5 * c3;
```

Интерпретация, например, выражения –(c1 + 2) будет следующей:

operator –((operator + (c1, complex (double (2)))).

При выполнении этого выражения неявные вызовы конструкторов создадут временные константы типа complex: (2.0, 0.0), (3.0, 0.0), (4.5, 0.0), которые будут уничтожены сразу же после того, как в них отпадет надобность. Заметим, что здесь не только происходит неявный вызов конструктора complex, но и неявное стандартное преобразование значения типа int к типу double. Число уровней неявных преобразований ограничено. При этом правила таковы: компилятор может выполнить не более одного неявного стандартного преобразования и не более одного неявного преобразования, определенного программистом.

Пример:

```
class A{public:
A(double d){. . .}
};
class B{ public:
B(A va){. . .}
};
class C{ public:
C(B vb){ . . .}
};
A var1 (1.2);          // A(double)
B var2 (3.4);          // B(A(double))
B var3 (var1);         // B(A)
C var4 (var3);         // C(B)
C var5 (var1);         // C(B(A))
C var6 (5.6);          // Ошибка! Неявно вызывается C(B(A(double)))
C var7 (A(5.6));       // C(B(A))
```

Ошибка при создании переменной var6 связана с тем, что необходимо два уровня неявных нестандартных преобразований, выполняющихся с помощью вызова конструкторов: double к A, а затем A к B.

При создании переменной var7 одно из этих преобразований – double к A – сделано явным, и теперь все будет нормально.

Таким образом, конструктор с одним аргументом Class::Class(type) всегда определяет преобразование типа type к типу Class, а не только способ создания объекта при явном обращении к нему.

Для преобразования абстрактного типа к стандартному или абстрактного к абстрактному в C++ существует средство – функция, выполняющая преобразование типов, или **оператор–функция преобразования типов**.

Она имеет вид

Class::operator type (void);

Эта функция выполняет определенное пользователем преобразование типа Class к типу type. Эта функция должна быть членом класса Class и не иметь аргументов. Кроме того, в ее объявлении не указывается тип возвращаемого значения. Обращение к этой функции может быть как явным, так и неявным. Для выполнения явного преобразования можно использовать как традиционную, так и «функциональную» форму.

Пример 1:

```
class X{int a, b;
public:
X (X & vx){a = vx.a; b = vx.b;}
X (int i, int j){a = 2*i, b = 3*j;}
operator double(){ return (a + b)/2.0;}  // Преобразование
                                         // абстрактного типа к стандартному.
};
int i = 5;
double d1 = double (i); // Явное преобразование типа  int  к  double;
double d2 = i ;         // неявное преобразование  int  к  double;
X xv(5, -8);
double d3 = double(xv);  // явное преобразование типа  X  к  double;
double d4 = xv;          // неявное преобразование X  к  double.
```

Пример 2:

```
// Преобразование абстрактного типа к абстрактному.

class Y {
char * str1;                // Строки  str1  и  str2  хранят символьное
char * str2;                // представление целых чисел.
public:
Y (char *s1, char *s2): str1(s1), str2 (s2){}
operator X() { return X (atoi (str1), atoi (str2));}
};
. . .
Y  yvar (“12“,“-25“);
X  xvar = yvar;
```

При создании переменной xvar перед вызовом конструктора копирования X::X(X&) будет выполнено неявное преобразование значения переменной yvar к типу X. Это же преобразование в явном виде может выглядеть так:

```
X  xvar = X (yvar);
X  xvar = (X) yvar;
```

Для выражения

```
X xvar = X (“12“, “-25“);
```

компилятор выдаст сообщение об ошибке «не найден конструктор с указанными аргументами». Дело в том, что в отличие от конструктора, оператор-функция преобразования типа не может создать объект абстрактного типа. Она способна только выполнить преобразование значения уже созданного объекта одного типа к значению другого типа.
В последнем же примере объект типа Y еще не существует.

22. Некоторые особенности переопределенных операций

Ограничениями при переопределении операций =, [], (), -> является то, что реализующие их функции operator=() и т.д. обязательно должны являться членами класса и не могут быть статическими функциями. Если говорить о механизме наследования, то обычно производный класс наследует все свойства класса базового. Из этого правила есть два исключения.

1) Производный класс не может наследовать конструкторы своего базового класса.

2) Операция присваивания, переопределенная для базового класса, не считается переопределенной для его производных классов.

Все остальные операции наследуются обычным образом, то есть, если для производного класса нужная операция не переопределена, но она переопределена в его базовом классе, то будет вызвана операция базового класса.

22.1. Операция =

Операция присваивания = является предопределенной для любого абстрактного типа данных.

При этом такая предопределенная операция присваивания интерпретируется не как получение побитовой копии объекта, а как последовательное выполнение операции присваивания над его членами (как стандартных так и абстрактных типов). Побитовое копирование происходит только тогда, когда операция = не определена. Предопределенную операцию = можно переопределить.

```
struct memberone { int i;
memberone & operator = (memberone & a){
cout << "Операция копирования  класса  memberone\n\n";
return a;
}
};
struct membertwo { int j;
membertwo & operator = (membertwo & a){
cout << "Операция копирования  класса  membertwo\n\n";
return a; }
};
```

```
struct contain { int k;
memberone  mo;
membertwo  mt;
};
void main () {
contain from;
from.mo.i = 1;
from. mt.j = 2;
from.k = 3;
contain  to;
to.mo.i = 0;
to.mt.j  = 0;
to.k = 0;
to = from;
cout   << “to.mo.i = “ << to.mo.i  << “\n\n“
        << “to.mt.j = “  << to.mt.j   << “\n\n“
        << “to.k = “ << to.k << “\n\n“;}
```

Результат работы программы:

```
Операция копирования класса memberone
Операция копирования класса membertwo
to.mo.i = 0
to.mt.j = 0
to.k = 3
```

Пример 2.
Рассмотрим снова класс stroka:

```
class stroka {
char *c;
int  len;
public:
 . . .
stroka & operator = ( stroka & str );
 . . .
};
stroka & stroka::operator = (stroka & str) {
if ( str.len > len ){
cout << “Длина строки мала! Копирование невозможно!\n“;}
else { strcpy ( c, str.c );  len = str.len;}
return * this;
}
```

```
void main( ) {
stroka A (“Строка A“),  B (“Строка“), C(“Str“);
A = B; A.display ( );
B = C;  B.display ( );
C = A; C.display ( );
}
```

В результате выполнения этой программы на мониторе появится:

```
Длина строки: 6
Содержание строки: Строка
Длина строки: 3
Содержание строки: Str
Длина строки мала! Копирование невозможно!
Длина строки: 3
Содержание строки: Str
```

22.2. Операция [ ]

Выражение **x[y]**, где x – объект абстрактного типа Class, интерпретируется как

x.operator[ ](y).

Заметим, что массив объектов абстрактного типа, как и любого стандартного, имеет стандартный тип – указатель.

Даже если array – массив элементов абстрактного типа Class, выражение array[i] по-прежнему означает *(array + i), вне зависимости от того, переопределена операция [ ] для типа Class или нет.

Пример:

```
class A{
int a [10];
public:
A(){ for ( int i = 0; i < 10; i ++ ) a [i] = i + 1; }
int operator [ ] ( int j ) {
return a [j]; }
};
void main( ){
A  array [20];
cout << “array[3][5] = “ << array [3][5] << “.\n“;
}
```

Результатом работы программы будет: array[3][5] = 6.

Очевидно, что операция [ ], использованная в конструкторе класса A, является стандартной, так как она выполняется над именем массива.

Рассмотрим теперь выражение array [3][5]. Результат его вычисления является таким, как и ожидалось, по следующей причине: операция [ ] выполняется слева направо. Следовательно, выражение array[3][5] интерпретируется как (array[3]).operator[ ](5).

Первая из двух операций [ ] является стандартной, так как выполняется над именем массива. При этом неважно, какой тип имеют его элементы.

Вторая операция [ ] – переопределенная, так как результатом первой операции [ ] является объект типа A.

Встает вопрос: когда имеет смысл переопределять операцию [ ]?

Попробуем создать АТД, который можно было бы использовать в программе подобно массиву.

Чтобы создание такого типа имело смысл, необходимо преодолеть основные недостатки, свойственные обычным массивам С++, а именно:

- необходимость задания размера массива на стадии компиляции;
- отсутствие контроля выхода за границы массива;
- невозможность задания произвольных границ изменения индекса;
- отсутствие предопределенных операций присваивания массивов, выполнения над ними арифметических операций и т.д.

Создадим класс **Array**, являющийся формализацией концепции “**одномерный массив целых**“.

Для простоты предполагаем, что массив типа Array имеет тот же диапазон изменения индексов, что и обычный массив С++, а изменение его размера после создания невозможно.

Определим над типом Array операции присваивания, сложения и вывода. Чтобы обращаться к элементам такого массива, переопределим операцию [ ].

```
// Файл Array.cpp
# include < iostream >
# include < stdlib.h >
class Array {
int *pa;        // Массив целых;
int   sz;       // размер массива.
public:
```

```
Array( const Array &v );
Array( const int a[ ], int s );
Array( int s );
virtual ~Array( ){ delete pa; }
int size ( ) { return  sz; }
int & operator [ ] ( int );
Array & operator = ( Array& );
// Результат возвращается по ссылке для возможности
// множественного присваивания типа   a=b=c;
Array & operator + ( Array& );
ostream & print ( ostream& );
};
Array::Array (const int a[], int s ){
// Инициализация массива типа Array  обычным массивом.
if ( s <= 0 ) { cout << "Неверный размер массива: "<< s <<"\n";
exit (1);}
if (!( pa = new  int [ sz = s ] )){
cout<<"Неудача при выделении памяти \n"; exit (1); }
for ( int i = 0; i<sz; i++) pa [ i ] = a [ i ];}
Array::Array ( const Array &v ){         // Конструктор копирования.
if (!( pa = new  int [ sz = v.sz ] )){
cout << "Неудача при выделении памяти \n";  exit(1);}
for ( int i = 0; i < sz; i++) pa [ i ] = v.pa [ i ];
}
Array::Array ( int s ){
// Создание неинициализированного массива размером  s.
if ( s<= 0) { cout <<"Неверный размер массива \n";  exit(1); }
if (!(pa = new  int [ sz = s ] )) {
cout <<"Неудача при выделении памяти \n";  exit (1); }
}
int & Array::operator [ ] ( int  index ) {
if ( index < 0 || index >= sz){
cout <<"Выход за границу массива!\n";  exit (1); }
return pa[index];
}
```

/* Так как результат возвращается по ссылке, то возвращается не значение элемента, а сам этот элемент и поэтому выражение вида c[i], где c – типа Array, может находиться в левой части операции присваивания. */

```
ostream & Array::print ( ostream& out) {
out << ’\n’;
for ( int i = 0; i < sz; i++) out << pa[ i ]<<”  ”;
out <<”\n”;
return out; }
ostream & operator << ( ostream & out, Array & v ){
v.print ( out);
return out; }
Array & Array::operator + ( Array & op2 ) {
if ( sz != op.sz ){
cout << “Попытка сложить массивы разных размерностей!\n“;
exit (1);}
Array & tmp = *( new  Array (sz));
for ( int i = 0 ; i < sz ; i ++ ) tmp[ i ] = pa[ i ] + op2.pa[ i ];
return tmp;
}
Array & Array::operator = ( Array &v ){
if ( sz != .sz ) {
cout <<”Разные размеры массивов при присваивании!\n”; exit (1);}
for ( int i = 0; i < sz ; i ++ ) pa[ i ] = v[ i ];
return ( *this );
}
// Конец файла  Array.cpp.
```

Теперь можно написать следующую программу:

```
# include “Array.cpp“

void main ( ){
int a [ ] = { 1, 7, 3, 15, 6, 20, 7 };
Array  mas (a, sizeof  a / sizeof (int));
```

```
Array b (7);            // Неопределенный массив.
Array c = mas;          // Конструктор копирования.
b = mas + c ;
mas = b + ( c = mas );
for ( int i=0; i < 7; i ++)cout << a[i] << "  ";  cout << "\n";
cout << mas << b << c; // Сравните эти два вывода!
}
```

Создадим теперь класс Matrix, являющийся формализацией концепции двумерного массива.

```
// Файл  matrix.cpp

class Matrix { Array **pm;       // Массив указателей на  Array.
int r, c;                        // Размерности матрицы.
public:
Matrix ( int, int );
virtual ~Matrix ( );
int row( ) { return r;}
int col ( ) { return c;}
Array & operator [ ] (int);
Matrix & operator = (Matrix&);
Matrix & operator + (Matrix&);
Matrix & operator * (Matrix&);
ostream & print (ostream & s);
};

// Результатом операции  [ ], примененной к объекту типа Matrix,
// должен быть объект типа Array:

Array & Matrix::operator [ ] ( int index ){
if ( index < 0 || index >= r ) {
cout << " Выход за границу массива ! \n"; exit (1);}
return * pm [index];
}
```

```
// Конструктор:
Matrix::Matrix (int row, int col ){
pm = new  Array * [ row ];
for (int i = 0; i < row; i ++ ) pm [ i ] = new Array (col);
r = row; c = col;
}
Matrix:: ~Matrix (){for ( int i = 0; i<r; i ++)delete pm[i]; delete pm; }
ostream & Matrix::print ( ostream &s ){
s<<"\n";
for ( int i =0; i < r; i++ ){
Array &v = *pm [ i ];
for ( int j = 0; j < c; j++ ) s << v [ j ] <<"  ";
s << "\n"; }
return s ;
}
Matrix & Matrix::operator = ( Matrix & tmp ) {
if ( r != tmp.r ) { cout << "Разные размеры массивов!\n"; exit (1);  }
for ( int i = 0; i < r; i++ ) *pm [ i ] = tmp [ i ];
return *this;
}

Matrix & Matrix::operator + ( Matrix & op2 ){
if ( r != op2.r ) { cout << "Разные размеры массивов!\n"; exit (1); }
Matrix & tmp = *( new  Matrix ( r, c ));
for ( int i = 0; i < r ; i++ ) tmp [ i ] = *pm[ i ] + op2[ i ];
return tmp;
}
Matrix & Matrix::operator *( Matrix & op2 ) {
if ( c != op2.r ) {
cout << "Нельзя перемножить матрицы!\n"; exit (1);}
Matrix & tmp = *( new Matrix ( r, op2.c));
for ( int i = 0; i < r ; i++ )
for ( int j = 0; j<op2.c; j++){ tmp[ i ][ j ] =0;
for ( int k = 0; k < c ; k++ ) tmp [ i ][ j ]+=(*this)[ i ][ k ]*op2[k ][ j ];
}
return tmp;}
```

```
ostream & operator << ( ostream &s, Matrix & m ){ m.print ( s );
return s;
}
// Конец файла  Matrix.cpp
```

Теперь можно написать программу, в которой используются новые типы данных Array и Matrix.

```
#include < iostream >
#include “array.cpp“
#include “matrix.cpp“

void main ( ){
Matrix tbl ( 3, 5), tbl2( 3, 5 );
for ( int i = 0; i < 3; i++ )
for ( int j = 0; j < 5; j++ ) {
tbl[ i ][ j ] = i + j;
tbl2[ i ][ j ] = ( i + j )*10;
}

Matrix tbl3 = tbl + tbl2;
cout << tbl3;
Array arr (10), arr2 (10);
for ( i = 0; i < 10; i++ ) { arr[i] = i; arr2[i] = i*10; }
Array arr3 = arr + arr2;
cout << arr3;
Matrix  mas ( 5, arr.size ());
for ( i = 0; i < 5; i++ ) mas[i] = arr;
Matrix mas2 ( 3, arr.size ());
mas2 = tbl*mas;
cout << mas << mas2;
}
```

23. Классы и шаблоны

Шаблон семейства классов определяет способ построения отдельных классов подобно тому, как класс определяет правила построения и формат отдельных объектов. Шаблон семейства классов определяется так:

template < список параметров шаблона > определение класса

В определении класса, входящего в шаблон, особую роль играет имя класса. Оно является не именем отдельного класса, а параметризованным именем семейства классов.

Определение шаблона может быть только глобальным.

Рассмотрим класс vector, в число данных которого входит одномерный массив. Независимо от типа элементов этого массива в классе должны быть определены одни и те же базовые операции, например, доступ к элементу по индексу и тому подобное. Если тип элементов вектора задать в качестве параметра шаблона класса, то система будет формировать вектор нужного типа и соответствующий класс при каждом определении конкретного объекта.

```
// Файл vec.cpp
template < class T >        // T – параметр шаблона;
class vector {
T *pv;                      // одномерный массив;
int size;                   // размер массива.
public:
vector ( int );
~vector ( ) { delete []pv; }
T & operator [ ] ( int i ) { return pv [ i ]; }
. . .
};
template < class T >
vector < T >::vector ( int n ){
pv = new T[n];
size = n;
}
// Конец файла  vec.cpp
```

Когда шаблон введен, появляется возможность определить конкретные объекты конкретных классов, каждый из которых параметрически порожден из шаблона. Формат определения объекта одного из классов, порожденного шаблоном, следующий:

имя параметризованного класса
< **фактические параметры шаблона** >
имя объекта (**параметры конструктора**).

В нашем случае определить вектор, имеющий 100 компонент типа double, можно так:

```
vector < double > x ( 100 );
```

Программа может выглядеть так:

```
#include < iostream >
#include “vec.cpp“
void main ( ){
vector < int > x ( 5 );
vector < char > c( 5 );
for ( int i = 0; i < 5; i++ ){
x [ i ] = i; c [ i ] = ’A’ + i;}
for ( i = 0; i < 5; i++ ) cout << “  “ << x [ i ] << “  “ << c [ i ];
cout << “/n“;
}
```

Результат:

```
0  A  1  B  2  C  3  D  4  E
```

В списке параметров шаблона могут присутствовать формальные переменные, не определяющие тип. Точнее, это параметры, для которых тип фиксирован:

```
template < class T, int size = 64 >
class row {
T * data;
int length;
public: row ( ) {
length = size;  data = new T [ size];
}
```

```
~row ( ) { delete T[] data;
}
T & operator [ ] ( int i ) { return data [i]; }
};

void main ( ){
row < float, 7 > rf;
row < int, 7 > ri;
for ( int i = 0; i < 7; i++ ){ rf[i] = i ; ri[i] = i * i; }
for ( i = 0; i < 8; i++ )
cout << "  " << rf[i] << "  " << ri[i];
cout << "\n";
}
```

Результат:

```
0 0   1 1   2 4   3 9   4 16   5 25   6 36
```

В качестве фактического аргумента для параметра size взята константа. В общем случае может быть использовано **константное** выражение, однако использовать выражения, содержащие переменные, нельзя.

24. Списки

Рассмотрим структуру

```
typedef  int  ETYPE;
struct elem { ETYPE data;
elem *next;
};
```

Назовем data – информационным элементом. У нас он – типа int, но может быть любого сложного необходимого нам типа ETYPE. Указатель next указывает на объект типа elem. Объекты типа elem можно упорядочить с помощью указателя next следующим образом (рис. 2):

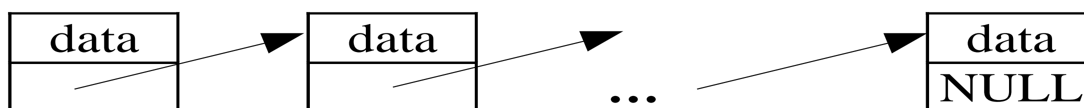


Рис. 2. Структура односвязного списка

Такая структура данных называется однонаправленным, или односвязным списком, иногда – цепочкой.

Объекты типа elem из этого списка называют элементами списка или звеньями. Каждый элемент цепочки, кроме последнего, содержит указатель на следующий за ним элемент. Признаком того, что элемент является последним в списке, служит то, что член типа elem* этого звена равен NULL.

Вместе с каждым списком рассматривается переменная, значением которой является указатель на первый элемент списка. Если список не имеет ни одного элемента, то есть пуст, значением этой переменной должен быть NULL.

Рассмотрим методы работы с такими списками.

Пусть переменные p, q имеют тип elem*:

```
elem *p, *q;
```

Построим список из двух элементов, содержащих числа 12 и –8.

Значением переменной p всегда будет указатель на первый элемент уже построенной части списка. Переменная q будет использоваться для выделения с помощью new места в памяти под размещение новых элементов списка.

Выполнение оператора

p = NULL;

приводит к созданию пустого списка. После выполнения операторов

q = new elem; q->data = –8; q->next = p; p = q;

имеем список, состоящий из одного элемента, содержащего число –8 в информационной части. Переменные p, q указывают на этот элемент (рис. 3а):

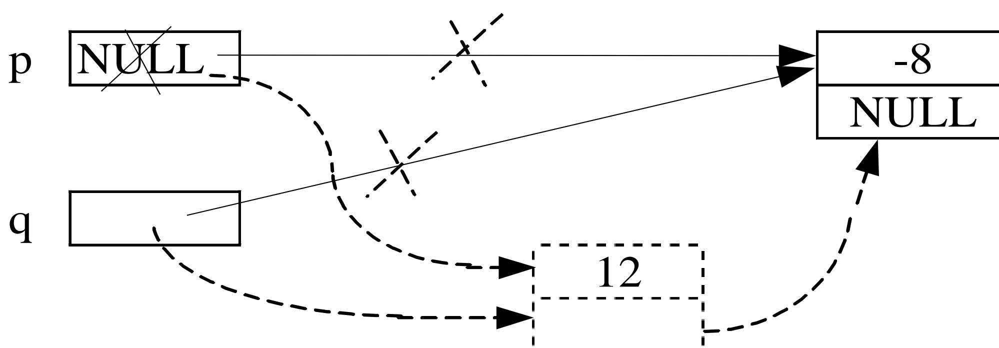


Рис. 3. Создание списка а) из одного элемента (сплошные линии); б) из двух элементов (пунктир)

Далее, выполнение операторов (рис. 3б)

q = new elem; q->data = 12; q->next = p; p = q;

приводит к тому, что в начало цепочки добавляется новый элемент, одержащий число 12. В результате получится список, изображенный на рис. 4. Значением переменных p и q снова является указатель на первый элемент списка:

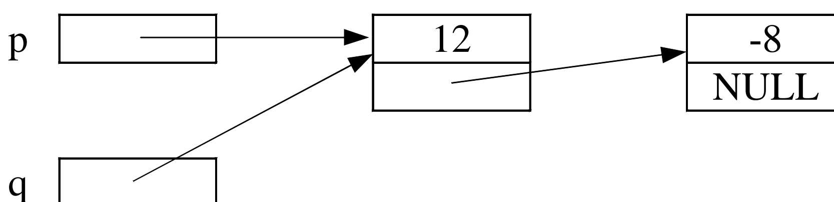


Рис. 4. Список из двух элементов

Фактически мы рассмотрели операцию включения нового элемента в начало, или голову списка, а формирование списка состоит в том, что начинают с пустого списка и последовательно добавляют в начало элементы.

Пример:

Построим список, элементы которого содержат целые числа 1, 2, 3, . . ., n.

```
p = NULL;
while ( n > 0 ){ q = new elem;
q->data = n;  q->next = p;  p = q ;
n – –;}
```

Заметим, что при включении элемента в голову списка порядок элементов в списке обратен порядку их включения.

Основная операция при работе со списком – это проход по списку.

Предположим, что с каждым информационным элементом звена нужно выполнить некоторую операцию, которая реализуется функцией void P(ETYPE). Пусть также опять p указывает на начало списка. Тогда проход по списку осуществляется так:

```
q = p;
while (q) {
P ( q->data );
q = q->next; }
```

Пример. Во входном файле num.txt находится последовательность, содержащая нечетное количество целых чисел.

Напишем программу, в результате выполнения которой выводится число, занимающее в этой последовательности центральную позицию.

```
#include < fstream.h >     // Для работы с файлом ввода.
#include < stdlib.h >
struct elem { int data;
elem *next; };
void main ( ){
ifstream infile ( “num.txt“ );
/* Создается входной поток с именем infile для чтения данных, разыскивается файл с именем “num.txt“; если такой файл не существует, то конструктор завершает работу аварийно и возвращает для infile нулевое значение.
*/
if ( !infile ) {
cout << “Ошибка при открытии файла num.txt!\n“; exit (1); }
elem *p = NULL, *q;
int  j = 0;
```

```
while ( infile.peek() != EOF ) {
/* Функция-член  peek()  класса  ifstream  возвращает очередной символ из входного потока  infile, фактически не извлекая его оттуда. Если встретится конец файла, то будет возвращено значение  EOF, то есть  -1. */
q = new elem;
infile >> q->data;
q->next = p;  p = q;
j ++;
}
for ( int i = 1; i <= j/2; i++ )
q = q->next;
cout << q->data << "\n";
}
```

24.1. Операции над односвязными списками

Основных операций над списками – три.

1) Проход по списку, или переход от элемента к следующему.

Как мы уже рассмотрели, это осуществляется с помощью присвоения типа q = q -> next;

2) Включение в список.

Пусть q, r – переменные типа elem*.

Предположим, что необходимо включить новый элемент в список после некоторого элемента, на который указывает q. Создадим этот новый элемент с помощью указателя r и занесем в его информационную часть число 19.

Такое включение осуществляется следующими операторами:

```
r = new elem;  r->data = 19;
r->next = q->next;  q->next = r;
```

Проиллюстрируем это на рис. 5.

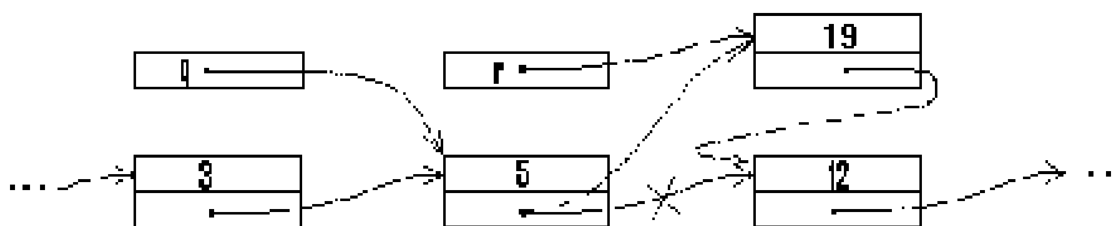


Рис. 5. Включение элемента в список

3) Исключение из списка.

Пусть теперь значением переменной q является указатель на некоторый, не последний, элемент списка и требуется исключить из списка элемент, следующий за ним. Это можно сделать так:

```
r = q->next;
q->next = q->next->next;
r->next = NULL;
```

Второе из приведенных присваиваний – это собственно исключение из списка, а первое выполняется для того, чтобы сохранить указатель на исключенный элемент, т.е. чтобы после исключения из списка он оставался доступным и с ним можно было бы выполнять некоторые действия. Например, вставить его на другое место или освободить занимаемую им память с помощью операции delete r.

Третье присваивание выполняется для того, чтобы сделать исключение окончательным, т.е. чтобы из исключенного элемента нельзя было бы попасть в список, из которого он исключен.

Реализация списка

Реализуем понятие списка через механизм классов.

```
// Файл ”list.cpp”
#include < iostream >
#include < stdlib.h >
typedef int ETYPE;
struct elem {
ETYPE data;
elem * next;
elem ( ETYPE e, elem * n ) { data = e; next = n;}
};
class list {
elem *h;                         // Адрес начала списка.
public:
list ( ) { h = NULL; }
~list ( ) {release ( ); }
void create ( ETYPE );           // Добавляет элемент в начало
списка.
void release ( );                // Удаляет список.
void insert (elem* q, ETYPE c); // Вставляет в список c после q.
void del0 ( ) {                  // Удаляет первый элемент.
elem *t = h; h = h->next; delete t;}
```

```
void del ( elem * q );                    // Удаляет элемент после q.
void print ( );                           // Распечатывает список.
friend class iter;
elem *first ( ) { return h; }
};
class iter {
elem * current;
public:
iter ( list & l )  { current = l.h; }
elem * operator ++ ( );                   // Продвижение по списку.
};
void list::create ( ETYPE c ){ h = new elem ( c, h ); }
void list::insert ( elem *q, ETYPE c ){
q->next = new elem ( c, q->next );}
void list::del ( elem *q ){ if ( q->next = = NULL ){
cout << ”Конец списка! ”<<
”Удаление следующего элемента невозможно!\n”; exit (1); }
elem * r = q->next;  q-> next = q->next->next;
r->next = NULL;
delete  r;}
elem iter::operator ++ ( ){
/* Возвращает указатель на текущий элемент списка. Осуществля-
ет продвижение по списку. Запоминает новый текущий элемент списка.
*/
if ( current ) { elem * tmp = current;
current = current->next;
return tmp; }
return NULL;
}
void list::release ( ) {
iter t ( *this );
elem *p;
while (( p = ++t )!= NULL ) {h = h->next; delete  p;}}
void list::print ( ) {
iter t ( *this );
elem *p;
while (( p = ++t )!= NULL )
cout << p->data << “  “;
cout << ’\n’;
}
// Конец файла  list.cpp
```

Здесь реализован односвязный список. Это одна из простых моделей структур управления данным. Класс list, реализующий эту модель, – представитель так называемых содержательных или **контейнерных** типов. Класс iter создан специально для перебора элементов произвольного списка типа list. Объекты, предназначенные для перебора элементов внутри некоторого набора данных, обычно называют **итераторами**.

Приведем пример использования односвязного списка.

В файле int.txt расположена непустая последовательность целых чисел $A_1, A_2,, \dots, A_n$. Определить количество этих чисел n и вывести их в порядке возрастания.

Для решения этой задачи будем строить список, элементы которого упорядочены по возрастанию содержащихся в них целых чисел. Построение выполняется за n шагов. Первый шаг – это создание списка, состоящего из одного элемента, который содержит A_1. Очевидно, этот список упорядочен. На i-м шаге (i = 2, 3, … , n) переходим от упорядоченного списка, элементы которого содержат числа $A_1,\dots, A_{i-1}$, к упорядоченному списку, элементы которого содержат $A_1,\dots, A_{i-1}, A_i$. Для выполнения такого перехода достаточно включить в список новый элемент, содержащий A_i. Его надо вставить непосредственно за последним по порядку элементом, содержащим число, меньшее, чем A_i.

Если же все элементы исходного списка содержат числа, не меньшие, чем A_i, то новый элемент нужно вставить в начало списка.

```
#include “list.cpp“
#include < fstream.h >
void main ( ) {
ifstream file ( “int.txt” );
list  lst;  int i, n;
file >> i; n = 1;
lst.create ( i );
while ( file.peek ( )!= EOF )
{ file >> i;  n ++;
iter tmp ( lst );              // Создаем объект-итератор для перебора
                               // элементов списка  lst.
elem *p, *q;
while (( p = ++tmp)!= NULL )
if ( p->data < i ) q = p;
else  break;
if ( p = = lst.first ( )) lst.create ( i );
else lst.insert ( q, i );}
```

```
cout <<" В файле чисел:  " << n << "\n";
cout <<" Упорядоченный список:\n";
lst.print ( );
}
```

В последнем операторе if – else делается проверка p = = lst.first (). Это необходимо из-за того, что механизм вставки звена в начало списка и в список после указателя p различен. Различие возникает из-за того, что у первого элемента нет предыдущего. Иногда для единообразия в начало списка помещают так называемый заглавный элемент, который никогда не удаляют, и перед которым никогда не вставляют элемент. Его информационная часть обычно не используется.

24.2. Двунаправленные и кольцевые списки

Чтобы в списках был удобный способ доступа к предыдущим элементам, добавим в каждый элемент списка еще один указатель, значением которого будет адрес предыдущего звена списка:

```
struct elem{
ETYPE data;
elem * next;
elem * pred;
elem ( ETYPE c, elem * n, elem * p ){ data = c; next = n; pred = p; }
};
```

С помощью элементов такого типа (рис. 6) можно сформировать так называемый двунаправленный список (с заглавным элементом):

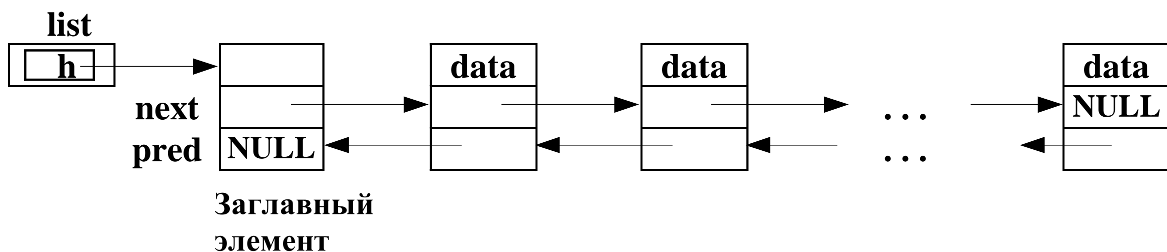


Рис. 6. Двунаправленный список

Здесь в поле pred заглавного звена содержится пустой указатель NULL, означающий, что у заглавного элемента нет предыдущего. Часто двунаправленные списки обобщают следующим образом (рис. 7, 8): в качестве значения next последнего звена принимают указатель на заглавное (или первое) звено, а в качестве значения поля pred заглавного (соответственно первого) звена – указатель на последнее звено:

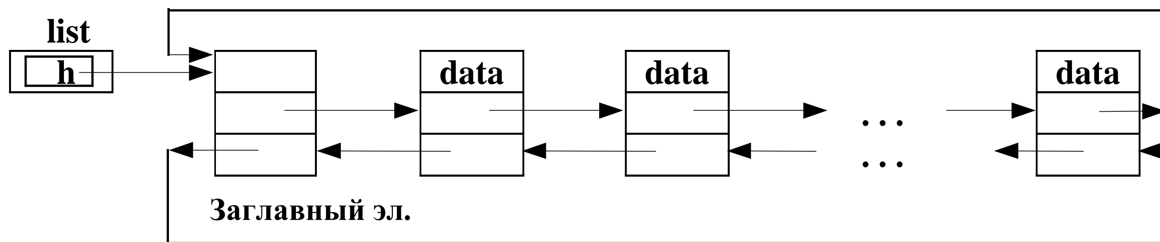


Рис. 7. Первый вариант двунаправленного кольцевого списка

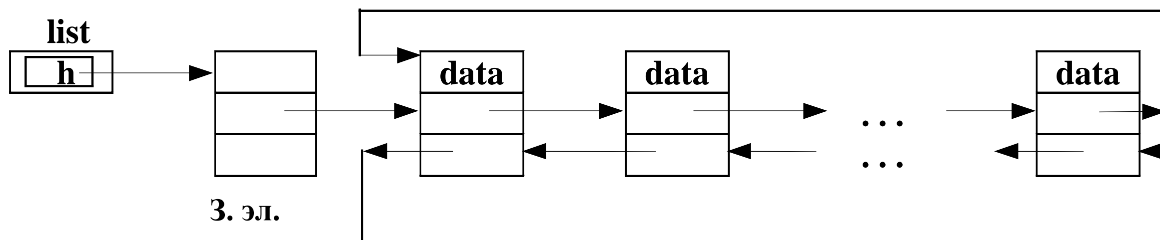


Рис. 8. Второй вариант двунаправленного кольцевого списка

Здесь список замыкается в кольцо, поэтому списки такого вида называют двунаправленными кольцевыми.

В первом варианте (рис. 7) очень просто реализуется вставка нового звена как в начало списка (после заглавного звена) так и в его конец, так как вставка звена в конец списка эквивалентна его вставке перед заглавным элементом. Но здесь при циклической обработке элементов придётся каждый раз проверять, не является ли очередное звено заглавным. Этого недостатка лишён второй вариант списка (рис. 8), но в этом случае труднее реализуется добавление в конец списка.

Рассмотрим основные операции с кольцевыми двунаправленными списками в первом варианте (рис. 7).

24.3. Операции над кольцевыми списками

Вставка элемента

Пусть h, p, q – переменные типа elem*, а k – переменная типа int. Значение k нужно занести в информационную часть элемента, который должен быть вставлен после звена, на которое указывает указатель p.

Эту вставку можно осуществить так:

```
q = new elem (k, p->next, p);
p->next->pred=q;  p->next=q;            // В таком порядке!
```

Для вставки нового элемента в начало списка достаточно, чтобы указатель p принял значение адреса заглавного элемента списка: p=h;

Удаление элемента

Возможность двигаться по указателям в любом направлении позволяет задавать исключаемое звено указателем p непосредственно на само это звено:

```
p->next->pred = p->pred;
p->pred->next = p->next;
delete  p;
```

Поиск элемента

Пусть h – указатель на заглавный элемент списка, r – указатель, который будет указывать на найденное звено, содержащее k. Пусть также p, q – переменные типа elem*, b – типа int. Поиск элемента, содержащего число k, осуществляется так:

```
b=1; h->data = k+1;     // В информационную часть заглавного звена
                        // заведомо занесём число, отличное от k.
p=h->next;              // Сначала p указывает на первое звено.
r=NULL;
q=p;                    // q указывает на первое звено.
do {
  if (p->data= =k){
  b=0;
  r=p;
  }
  p=p->next;
}while ((p!=q) && b);
```

Заметим, что если в списке вообще нет звена, содержащего k, то после поиска значение b останется равным единице, указатель r будет равен NULL, а p примет значение q, т.е. снова будет указывать на первое звено (после заглавного).

25. Стеки

В программировании часто используется структура данных, которая называется **очередью**. Над очередью определены две операции – занесение элемента в очередь и выбор элемента из очереди. При этом выбранный элемент исключается из очереди. В очереди доступны две позиции – ее начало (из этой позиции выбирается элемент из очереди) и конец (в эту позицию помещается заносимый в очередь элемент). Различают два основных вида очередей, отличающихся по дисциплине обслуживания. При первой из дисциплин элемент, поступивший в очередь первым, выбирается первым и удаляется из очереди. Эту дисциплину обслуживания очереди принято называть FIFO (First In – First Out → первый в очередь – первый из очереди).

Остановимся более подробно на очереди с такой дисциплиной обслуживания, при которой на обслуживание первым выбирается тот элемент очереди, который поступил в нее последним. Эту дисциплину обслуживания принято называть LIFO (Last In – First Out → последний в очередь – первый из очереди). Очередь такого вида в программировании называют **стеком** или магазином. В стеке доступна единственная его позиция, называемая вершиной стека. Это позиция, в которой находится последний по времени поступления элемент. Отобразим стек на подходящую структуру данных языка C++.

25.1. Реализация стека через массив

```
// Файл stack0.cpp
typedef char ETYPE;
class stack {
enum {EMPTY = –1};
char *s;
int max_len;
int top;
public:
stack ( ) {s = new ETYPE [100];
max_len = 100;
top = EMPTY;                    // Стек пуст.
}
stack (int size) {s = new ETYPE [size]; max_len = size;
top = EMPTY; }
stack (const ETYPE a [ ], int len){     // Инициализация массивом.
max_len = len;
s = new ETYPE [max_len];
```

```
for (int i = 0 ; i < max_len; i ++ ) s[i] = a[i];
top = max_len – 1;
}

stack (const stack & a) {                    // Инициализация стеком.
s = new ETYPE [a.max_len];
max_len = a.max_len;  top = a.top;
for (int i = 0 ; i < max_len; i ++ ) s[i] = a.s[i];}
~ stack () { delete  s ;}
void reset (){ top = EMPTY; }  // Сброс стека в состояние ПУСТ.
void push (ETYPE c ) { s [ ++ top ] = c ; }    // Занесение в стек.
ETYPE pop () { return (s [ top – – ]); }         // Извлечение из стека.
ETYPE top_show () const { return (s [top]); }
```

/* Возвращает элемент из стека, фактически не извлекая его. Модификатор const гарантирует, что эта функция не будет менять данные-члены объектов типа stack */

```
int empty () const { return (top = = EMPTY); }
// Проверяет, пуст ли стек. Возвращает
// 1, если стек пуст, 0 – если не пуст.
int full () const { return (top = = max_len – 1); }
// Проверяет, есть ли в стеке еще место.
};
// Конец файла  stack0.cpp
```

Теперь в программе могут появиться такие операторы:

```
stack  data (1000);  // Создание стека длиной 1000.
stack  d[5]               // Конструктор по умолчанию создает массив
                          // из 5  стеков по 100  элементов каждый.
stack  w (“ABCD“, 4);     // w.s[0] = ‘A’ . . . s[3] = ‘D’.
stack  cop ( w);          // cop – копия стека  w.
```

В качестве примера рассмотрим задачу вывода строки в обратном порядке.

```
# include <iostream>
# include “stack.cpp“
void main (){
char str [ ] = “Дядя  Вася!“;
stack  s;
int i = 0;
cout << str <<‘\n’;
```

```
while ( str [ i ] )
if ( !s.full ()) s.push (str [ i++]);
else {cout << “Стек заполнен!“ <<‘\n’;  break;}
while (!s.empty ())cout<<s.pop();        // Печать в обратном порядке.
cout <<’\n’;
}
```

Результат выполнения программы:

```
Дядя Вася!
!ясаВ ядяД
```

Можно решить эту задачу и так:

```
char str [ ] = “Дядя Вася!“;
stack s (str, 10);
while (!s.empty ()) cout <<s.pop(); cout <<’\n’;
```

25.2. Реализация стека через динамическую цепочку звеньев

Пусть значением указателя, представляющего стек в целом, является адрес вершины стека. Как и в случае односвязного списка, каждое звено будет содержать указатель на следующей элемент, причем «дно» стека (т.е. элемент, занесенный в стек раньше всех) содержит указатель NULL.

```
// Файл  stack.cpp
typedef  char  ETYPE;
struct elem {
ETYPE  data;
elem* next;
elem (ETYPE d, elem* n){ data = d;  next = n; }
};
class stack {
elem*h;                              // Адрес вершины стека.
public:
stack () {h = NULL;}                 // Создание пустого стека.
```

```
stack (ETYPE a [ ], int  len){     // Инициализация стека массивом.
h = NULL;
for (int i = 0; i< len; i++) h = new elem (a[i], h);}
stack (stack & a){              // Инициализация стека другим стеком.
elem *p,*q;
p = a.h;  q = NULL;
while (p) { q = new elem (p->data, q);
if (q->next = = NULL) h = q;
else  q->next->next = q;
p = p->next;
}
q->next = NULL;
}
~stack () {reset ();}
void push (ETYPE c) { h = new elem (c, h);
}                                                   //Поместить в стек.
ETYPE pop () { elem *q = h; ETYPE a = h->data;
                                                    // Извлечь из  стека.
h = h->next;  delete q;
return  a;
}
ETYPE  pop_show () {return  h->data;}        // Показать вершину.
void reset () { while (h ){ elem *q = h;  h = h->next;  delete  q; }
}
int empty () { return h ? 0:1;}
};
// Конец файла  stack.cpp
```

Приведем задачу, в решении которой удобно использовать стек.

В файле задана строка литер. Требуется проверить баланс скобок в этой строке.

Баланс соблюдается, если выполнено каждое из следующих условий:

1) Для каждой открывающей скобки справа от нее есть соответствующая закрывающая скобка. Наоборот, для каждой закрывающей скобки слева от нее есть соответствующая открывающая скобка.

2) Соответствующие пары скобок разных типов правильно вложены друг в друга.

Так в строке

{[(a*b)+(n-4)]/[7-sin(x)]+exp(d)}*s

баланс скобок соблюдается, а в строке

[{a+b[i]((x-sin(x))]d)) – нет.

В качестве результата необходимо вывести сообщение о соблюдении баланса, а также начало строки до первого по порядку нарушения баланса скобок.

Для решения задачи сформируем сначала пустой стек. Затем будем последовательно просматривать литеры строки, и открывающие скобки поместим в стек. Если очередной символ окажется закрывающей скобкой, выберем последнюю из занесенных в стек открывающих скобок и сравним эти скобки на соответствие друг другу. Если соответствие есть, то эффект должен быть такой же, как если бы этой пары скобок в строке вообще не было. Если эти скобки не соответствуют друг другу, то не выполнено второе условие соблюдения баланса скобок. Если же в момент выбора из строки очередной закрывающей скобки стек оказался пуст или по завершении просмотра стек оказался не пуст, то не выполнено первое условие соблюдения баланса скобок.

Обозначим через **s** – стек, **sym** – обрабатываемый символ, а через **b** – целую переменную, фиксирующую факт соответствия закрывающей скобки из строки с открывающей скобкой из вершины стека. Для проверки на соответствие закрывающей скобки, являющейся значением переменной **sym**, и открывающей скобки в вершине стека, опишем специальную функцию **accord**(), которая возвращает целое значение и удаляет открывающую скобку из стека.

```
# include <fstream.h>
# include <stdlib.h>
# include “stack.cpp“
int accord (stack & s, char sym){
char  r = s.pop ( );
switch (sym) {
case  ‘)’ :     return  r = = ‘(‘;
case  ‘]’ :     return  r = = ‘[‘;
case  ‘}’ :     return  r = = ‘{‘;
default   :     break;
}
}

void main ( ) {
ifstream  file ( “a.txt“ );
if ( !file ){ cout << “Ошибка при открытии файла a.txt!\n“; exit(1);
}
stack  s;
char  sym;  int  i, n, b = 1;
while ( file.peek () != EOF && b){
file >> sym;  cout <<sym;
switch (sym){
case ‘(’ : case ‘[’ : case ‘{’ : s.push(sym);  break;
case ‘)’ : case ‘]’ : case ‘}’ :
if ( s.empty () || !accord (s, sym)) b = 0;  break;
}
}
if (!b || !s.empty ()) cout <<”\nБаланса скобок нет!\n”;
else  cout <<”\nСкобки сбалансированы\n”;
}
```

26. Двоичные деревья

26.1. Определение и построение

Связанный граф, в котором нет циклов, называется деревом. Дерево называется ориентированным, если на каждом его ребре указано направление.

Двоичное дерево – это такое ориентированное дерево, в котором:

1) Имеется ровно одна вершина, в которую не входит ни одного ребра. Эта вершина называется корнем двоичного дерева.

2) В каждую вершину, кроме корня, входит одно ребро.

3) Из каждой вершины исходит не более двух ребер.

Будем изображать двоичные деревья так, чтобы корень располагался выше других вершин.

Для ребер, выходящих из любой вершины, имеется две возможности – быть направленными влево вниз и вправо вниз. При этом, если из вершины выходит одно ребро, то указание направления ребра влево вниз или вправо вниз является существенным. При таких соглашениях нет необходимости указывать направление на ребрах – все ребра ориентированы вниз.

При решении некоторых задач бывает удобно представить наборы объектов в виде двоичных деревьев.

Рассмотрим задачу кодирования непустой последовательности целых чисел.

Пусть дана последовательность целых чисел $a_1, \dots , a_n$ и функция целочисленного аргумента $F(x)$, принимающая целые значения. Значение $F(x)$ будем называть кодом числа x. Пусть требуется закодировать данные числа, т.е. вычислить значения $F(a_1), \dots , F(a_n)$, и пусть в последовательности $a_1, \dots , a_n$ имеются частые повторения значений элементов.

Чтобы избежать повторных вычислений одного и того же значения, следует по ходу кодирования строить таблицу уже найденных кодов. Организация этой таблицы должна быть такой, чтобы, во-первых, в ней довольно быстро можно было находить элементы с данным значением (или устанавливать факт отсутствия его в таблице), и, во-вторых, без особых затрат добавлять в таблицу новые элементы.

Этим требованиям удовлетворяет представление таблицы в виде двоичного дерева, в вершинах которого расположены различные элементы из данной последовательности и их коды.

Процесс построения двоичного дерева идет так: первое число и его код образуют корень. Следующее число, которое нужно кодировать, сравнивается с числом в корне, и если они равны, то дерево не разрастается и код может быть взят из корня. Если новое число меньше первого, то проводится ребро из корня влево вниз, иначе – вправо вниз. В образовавшуюся вершину помещаются это новое число и его код.

Пусть уже построено некоторое дерево и имеется некоторое число **x**, которое нужно закодировать. Сначала сравниваем с **x** число из корня. В случае равенства поиск закончен, и код **x** извлекается из корня. В противном случае переходим к рассмотрению вершины, которая находится слева внизу, если **x** меньше только что рассмотренного, или справа внизу, если **x** больше рассмотренного. Сравниваем **x** с числом из этой вершины и т.д. Процесс завершается в одном из двух случаев:

1) Найдена вершина, содержащая число **x**.

2) В дереве отсутствует вершина, к которой надо перейти для выполнения очередного шага поиска.

В первом случае из найденной вершины извлекается код числа **x**. Во втором необходимо вычислить F(x) и присоединить к дереву вершину, в которой расположены **x** и F(x). Пусть последовательность начинается с чисел 8, 4, 13, 10, 14, 10. Тогда вначале дерево будет разрастаться следующим образом (рис. 9 а-д):

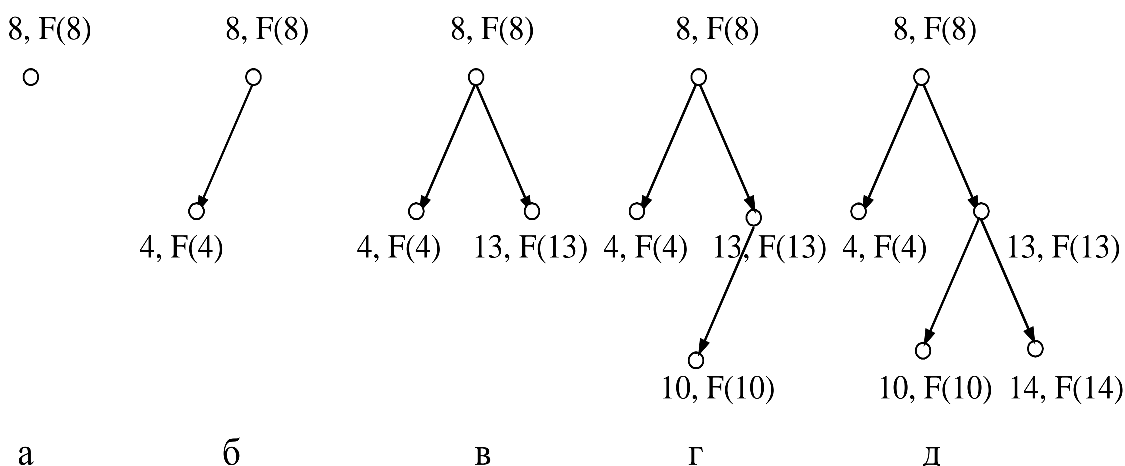


Рис. 9. Разрастание двоичного дерева в задаче кодирования

При появлении числа 10 в качестве шестого элемента последовательности к дереву не добавляется новой вершины, а значение F(10) извлекается из имеющейся вершины (рис. 9д).

Определим в программе построения последовательности кодов следующую структуру:

```
struct node {
int  num,  code;
node* left, * right;
node (int n,  int c,  node *l,  node *r){
num = n;  code = c;
left = l;  right = r;}
};
```

Объекты типа **node** являются структурами, в которых каждое из полей **left** и **right** есть либо NULL, либо указатель на место в памяти, отведенное с помощью new для объекта типа **node**. Дерево можно представить в виде множества объектов типа **node**, связанных указателями. Сами эти объекты будут вершинами дерева, а указатели на места в памяти, отведенные для объектов типа **node**, – ребрами дерева. Если при этом поле **left** (**right**) есть NULL, то это значит, что в дереве из данной вершины не исходит ребро, направленное влево вниз (вправо вниз). Изобразим на рис. 10 представление дерева, соответствующего рис. 9д, в памяти ЭВМ:

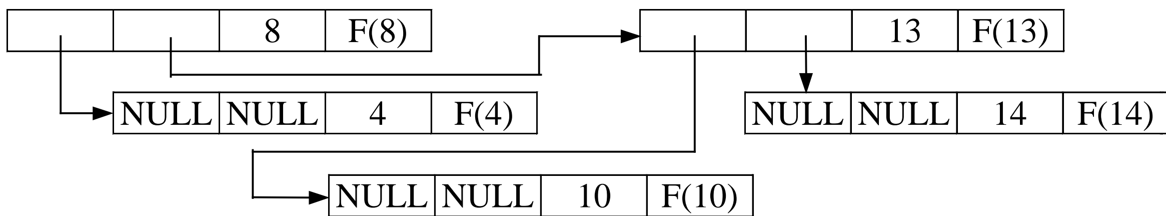


Рис. 10. Изображение дерева в памяти ЭВМ

Выполнение присваивания v = v->left (v = v->right) означает переход к вершине, расположенной непосредственно слева снизу (справа снизу) от данной, если, конечно, соответствующее поле данной вершины не есть NULL. Таким способом можно передвигаться от вершины к вершине сверху вниз. Включение новой вершины в дерево представляет собой изменение значений полей типа node* некоторых вершин данного дерева.

Вместе с каждым деревом рассматривается переменная, значением которой является указатель на корень дерева. Если в дерево не входит ни одной вершины, значение этой переменной должно равняться нулевому указателю NULL.

Напишем программу, по ходу выполнения которой кодируется последовательность натуральных чисел, расположенных во входном файле NUM.txt. Для кодирования используем файл COD.txt, компонен-

тами которого являются целые числа. Кодом F(k) числа k будем считать k-ю по порядку компоненту файла COD.txt.

```
struct node {
int  num,  code;
node* left, *right;
node (int  n, int  c, node* l, node* r){
num = n;  code = c;  left = l;  right = r;}
};
int f(int);
void insert(int n, node* root){
node* temp = root;
node* old;
while (temp !=0 ) {
old = temp;
if (temp->num = = n) { cout << temp->code <<”  ”; return; }
if (temp->num > n) temp = temp->left;
                 else     temp = temp->right;
}
int k = f(n);  cout << k << “  “;
if (old->num > n)   old->left = new  node(n, k, 0, 0);
else                old->right = new  node(n, k, 0, 0);
}
ifstream  num ( “num.txt“ ),  cod ( “cod.txt“ );
int f(int k) { int i, j;
cod.seekg(0);         // Устанавливает позицию чтения из файла  в  0.
for ( i = 1; i <=k; i++) cod >> j;
return j;
}

void main(){
int n;
num >> n;
node* root = new  node (n, f(n), 0, 0); cout << root->code << “  “;
while (num.peek() != EOF){  num >> n;
insert (n, root);
}
}
```

26.2. Таблицы

Построенное в последнем примере дерево часто называют деревом поиска. Дерево поиска иногда используют для построения таблиц, в которых хранятся различные сведения, обычно в виде структур. При этом для упорядочения структур они обычно именуются и чтобы организовать эффективный поиск структуры по ее имени, нужно иметь возможность сравнивать любые два имени и устанавливать, какое из них “больше”, а какое –“меньше”. Имя структуры в таблице часто называют ключом структуры. В качестве ключа часто используют целые числа или строки одинаковой длины.

Над таблицей как структурой данных определены операции:

- поиск в таблице структуры с заданным ключом;
- включение в таблицу структуры с заданным ключом;
- исключение из таблицы структуры с заданным ключом.

Мы рассмотрим организацию таблиц в виде двоичного дерева. В примере кодирования ключом служили сами числа из файла NUM.txt. Фактически мы уже рассмотрели операции поиска в дереве и включения элемента в дерево по заданному ключу. Сейчас оформим в виде функции операцию исключение элемента с заданным ключом из дерева.

Непосредственное удаление структуры реализуется просто, если удаляемая вершина дерева является конечной или из нее выходит только одна ветвь (рис. 11):

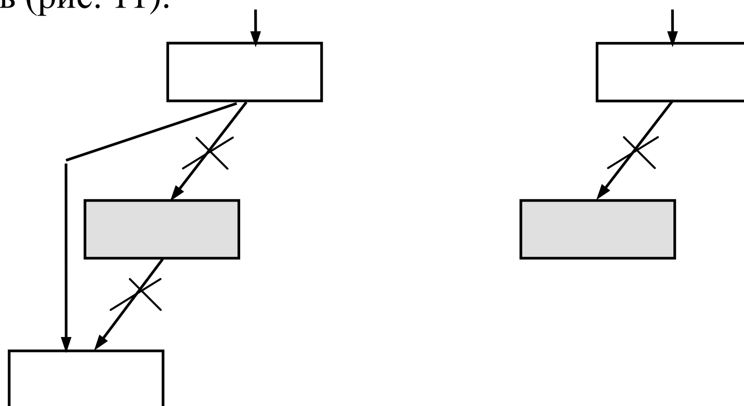


Рис. 11. Удаление элемента из дерева, когда удаляемая вершина является конечной или из нее выходит только одна ветвь

Трудность состоит в удалении вершины, из которой выходят две ветви. В этом случае нужно найти подходящее звено дерева, которое можно было бы вставить на место удаляемого, причём это подходящее звено должно просто перемещаться. Такое звено всегда существует: это либо самый правый элемент левого поддерева, либо самый левый эле-

мент правого поддерева. В первом случае для достижения этого звена необходимо перейти в следующую вершину по левой ветви, а затем переходить в очередные вершины только по правой ветви до тех пор, пока очередной указатель не будет равен NULL. Во втором случае необходимо перейти в следующую вершину по правой ветви, а затем переходить в очередные вершины только по левой ветви до тех пор, пока очередной указатель не станет равен NULL. Такие подходящие звенья могут иметь не более одной ветви. Ниже (рис. 12) схематично изображено исключение из дерева вершины с ключом 50:

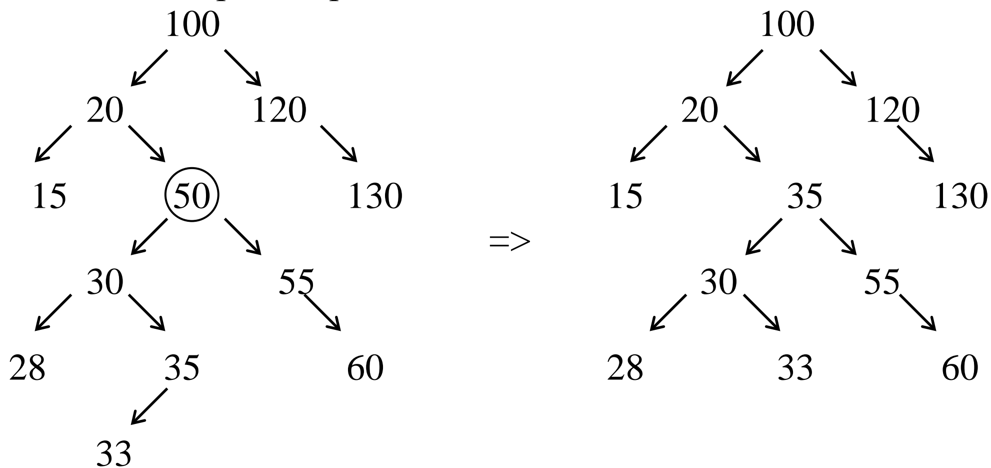


Рис. 12. Исключение из дерева вершины с ключом 50

Приведём программу, реализующую дерево поиска со всеми операциями с помощью класса **tree** и несколько изменённой функции **insert**.

```
#include <fstream.h>
struct node {
int  key,  code;
node *left,  *right;
node (int k, int c, node *l, node *r){ key = k; code = c; left = l;
right = r; }
};

int f(int);
class tree {
node *root;
void print (node *r);                    // Печать с указателя  r.
public:
int insert (int);
```

```
void del (int);
tree () { root = 0; }                // Создание пустого дерева.
node *&find (node *&r, int  key);    // Поиск элемента в поддереве
                                     // с вершиной  r.
node *&find (int key) {              // Поиск по всему дереву.
return ( find(root, key) );
}

void print () { print (root); }      // Печать всего дерева.
};

int tree::insert (int key){          // Возвращает код.
node* t = root;
node* old;
int k;
if (root = = 0 ) { k = f( key );
root = new node( key, k, 0, 0 );
return  k;
}
while (t !=0){
old = t;
if (t-> key = = key ){
return  t->code; }
if  ( t->key > key ) t = t->left;
else   t = t->right; }
k = f (key);

if ( old->key > key) old->left = new node ( key, k, 0, 0);
else  old->right = new node ( key, k, 0, 0);
return k; }

node *&tree::find ( node *&r, int  key){      // Рекурсивная функция
if ( r = = 0) return  r;                      // поиска в поддереве  r.
if ( r->key = = key ) return  r;
if ( r->key > key ) return ( find ( r->left, key ));
else
return ( find ( r->right, key ));
}

void del_el ( node *&r, node *q){    // Рекурсивная функция
if ( r->right = = 0 ) {              // удаления элемента,
q->key = r->key;                     // на который указывает  q.
```

```
q->code = r->code;
q = r;  r = r-> left;
} else
del_el ( r->righ, q );
}

void tree::del (int  key){   // Удаление элемента с ключом  key.
node *&q = find ( key);   // Установка указателя q на удаляемый
                          // элемент.
if ( q = = 0) { cout << “Элемента с ключом  “ << key <<“  нет\n“;
return; }
if ( q->right = = 0) q = q->left;
else
if ( q->left = = 0)    q = q->right;
else
del_el ( q->left, q);
}

void print ( node *r){              // Глобальная функция печати
                                    // элемента  дерева  r.
cout << r->key << “   “ << r->code << “   “;
}
void tree::print ( node *r) {       // Рекурсивная функция печати
if (r != 0) {                       // поддерева, начиная с  r.
::print ( r);
print ( r->left );
print ( r->right );
}
}
ifstream  numb (“num.txt“),  cod (“cod.txt“);
int f ( int k ){ int i, j;
cod.seekg ( 0 );
for ( i = 1; i <= k; i++) cod >> j;
return  j;
}
```

```
void main (){
cout << "----------------------------------------------------------------\n";
int  key;
tree  t;

while ( num.peek () != EOF) {
numb >> key;
cout << t.insert ( key ) << "   ";          // Вставка и печать элемента,
}                                            // или просто печать, если
cout << '\n';                                // элемент уже есть.

t.print ();
cout << "\n\n";
key = 50;
t.del ( key );
t.print ();
cout << '\n';
}
```

Заметим, что рекурсивная функция del_el() вызывается, если из удаляемой вершины направлены два ребра дерева. Она спускается до самого правого звена левого поддерева удаляемого элемента *q, а затем заменяет значения членов key и code в *q соответствующими значениями полей key и code вершины *r. После этого вершину, на которую указывает r, можно исключать с помощью оператора r = r->left. Можно видоизменить эту функцию, освобождая память, занимаемую удалённым звеном с помощью операции delete.

27. Потоковый ввод-вывод

Мы уже неоднократно пользовались различными потоками ввода/вывода. Здесь мы рассмотрим работу с потоками более подробно. Ввод/вывод потоков в C++ используется для преобразования типизированных объектов в читаемый текст и обратно.

Классы, связанные с потоками C++, содержат расширяемые библиотеки, позволяющие выполнять форматированный ввод/вывод с контролем типов как для предопределенных, так и для определяемых пользователем типов данных с помощью перегруженных операций и прочих объектно-ориентированных методов.

Потоком будем называть понятие, относящееся к любому переносу данных от источника (или поставщика данных) к приемнику (или потребителю) данных. Несмотря на свое имя, класс потока может быть использован для форматирования данных в ситуациях, не подразумевающих реального выполнения ввода/вывода. Так, форматирование в памяти можно применять к символьным массивам и прочим структурам.

В файле iostream имеется два параллельных класса: **streambuf** и **ios**. Оба они являются классами низкого уровня, и каждый выполняет свой круг задач.

Класс streambuf обеспечивает общие правила буферизации и обработки потоков в тех случаях, когда не требуется значительного форматирования этих потоков. Класс streambuf представляет собой базовый класс, используемый другими классами из iostream. Большинство функций-членов streambuf являются встраиваемыми (inline) для обеспечения максимальной эффективности. Классы strstreambuf и filebuf являются производными от streambuf.

Класс ios (и, следовательно, производные от него классы) содержит указатель на streambuf.

Класс ios имеет два производных класса: istream (для ввода) и ostream (для вывода). Другой класс, iostream, является производным классом сразу от istream и ostream вследствие множественного наследования:

```
class ios;
class istream: virtual public ios;
class ostream: virtual public ios;
class iostream: public istream, public ostream;
```

Кроме того, существует три класса _withassign, являющихся производными классами от istream, ostream и iostream:

```
class istream_withassign: public istream;
class ostream_withassign: public ostream;
class iostream_withassign: public iostream;
```

27.1. Классы потоков

Класс ios содержит переменные состояния для интерфейса с streambuf и обработки ошибок.

Класс istream поддерживает как форматированные, так и неформатированные преобразования потоков символов, извлекаемых из streambuf.

Класс ostream поддерживает как форматированные, так и неформатированные преобразования потоков символов, помещаемых в streambuf.

Класс iostream объединяет классы istream и ostream для двунаправленных операций, в которых один поток действует и как источник, и как приемник.

Производные классы _withassign обеспечивают четыре предопределенных "стандартных" потока: cin, cout, cerr и clog, описываемые в следующем разделе. Классы _withassign добавляют к соответствующим базовым классам операции присваивания следующим образом:

```
class istream_withassign: public istream {
istream_withassign();
istream& operator = (istream&);
istream& operator = (streambuf*);
}
```

и аналогично для ostream_withassign и iostream_withassign.

Классом потока называется любой класс, производный от классов istream и ostream.

27.2. Стандартные потоки

Выполнение любой программы C++ начинаются с четырьмя предопределенными открытыми потоками, объявленными как объекты классов _withassign в iostream следующим образом:

```
extern istream_withassign cin;
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;
```

Их конструкторы вызываются всякий раз при включении iostream, но фактическая инициализация выполняется только один раз.

Все эти предопределенные стандартные потоки по умолчанию связаны с терминалом.

Четыре стандартных потока предназначены для:
cin – стандартного ввода;
cout – стандартного вывода;
cerr – стандартного вывода ошибок;
clog –полностью буферизованного вывода ошибок.

В табл. 5 приведено назначение классов потокового ввода-вывода.

Таблица 5

Назначение классов потокового ввода-вывода

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
ofstream	Потоковый класс для вывода в файл
ostream_withassign	Потоковый класс ввода для cout, cerr, and 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
sir stream	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
s trs treambuf	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 операцию присваивания, что позволяет изменять указатель на используемый классом буфер.

Потоки ввода-вывода С++ предоставляют некоторые преимущества по сравнению с функциями ввода-вывода библиотеки С.

Безопасность типов. Сравним вызов функций библиотеки С и использование стандартных потоков С++. Вызов функции printf() выглядит следующим образом:

```
#include <stdio.h>
. . .
int n = 12;
char name[] = ”Вывод строки на экран\n”;
printf(”%d  %s”, i, name);
```

Этот вызов приведет к следующей правильной печати:

```
12  Вывод строки на экран
```

Но если по невнимательности поменять местами аргументы для printf(), ошибка обнаружится только во время исполнения программы. Может произойти все что угодно – от странного вывода до краха системы. Этого не может случиться в случае использования стандартных потоков:

```
#include <iostream>
cout << i <<' ' << name << '\n';
```

Так как имеются перегруженные версии операции сдвига operator<<(), правая операция всегда будет выполнена. 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 С++ является то, что определенные пользователем типы данных могут быть без труда в них встроены. Рассмотрим класс Data, данные которого необходимо печатать:

```
struct Data {
int x; char* y;
};
```

Все, что нужно сделать, это переопределить операцию << для нового типа Data. Соответствующая функция operator<<() может быть реализована так:

```
ostream &operator<<(ostream & out, const Data & p){
return out << p.x << ' ' << p.y;
}
```

После этого станет возможно осуществлять вывод:

```
#include <iostream>
struct Data {int x; char* y;
Data (int x, char* y){this->x = x;  this->y = y;}
};

ostream &operator<<(ostream & out, const Data & p){
return out << p.x <<'  '<< p.y;
}

void main(){
Data  p(1, "Error");
cout << p<<'\n';
}
```

27.3. Операции помещения и извлечения из потока

В ы в о д в п о т о к выполняется с помощью ***операции вставки*** (в поток), которая является перегруженной операцией сдвига влево <<. Левым ее операндом является объект потока вывода. Правым операндом может являться любая переменная, для которой определен вывод в поток (то есть переменная любого встроенного типа или любого определенного пользователем типа, для которого она перегружена). Например, оператор **cout << "Hello!\n"**; приводит к выводу в предопределенный поток **cout** строки "Hello!".

Операция << возвращает ссылку на объект типа ostream, для которого она вызвана. Это позволяет строить цепочки вызовов операции вставки в поток, которые выполняются слева направо:

```
int i = 5;
double d = 2.08;
cout << "i = " << i << ", d = " << d << '\n';
```

Эти операторы приведут к выводу на экран следующей строки:

```
i = 5, d = 2.08
```

Операция вставки в поток поддерживает следующие встроенные типы данных: char, short, int, long, char* (рассматриваемый как строка), float, double, long double, void*:

```
ostream& operator<<(short n);
ostream& operator<<(unsigned short n);
ostream& operator<<(int n);
ostream& operator<<(unsigned int n);
ostream& operator<<(long n);
ostream& operator<<(unsigned long n) ;
ostream& operator<<(float f);
ostream& operator<<(double f);
ostream& opera to r<<(long double f) ;
ostream& operator<<(const void *p);
```

Целочисленные типы форматируются в соответствии с правилами, принятыми по умолчанию, если они не изменены путем установки

различных флагов форматирования. Тип void* используется для отображения адреса:

```
int i;
// Отобразить адрес в 16-ричной форме:
cout << &i;
```

Отметим, что переопределение не изменяет нормального приоритета выполнения операции <<, поэтому можно записать

```
cout << “sum =“ << x+y << “\n“;
```

без круглых скобок. Однако, в случае

```
cout << (x & y) << “\n“;
```

круглые скобки нужны.

Для ввода информации из потока используется ***операция извлечения,*** которой является перегруженная операция сдвига вправо >>. Левым операндом операции >> является объект класса istream, который также является и результатом операции. Это позволяет строить цепочки операций извлечения из потока, выполняемых слева направо. Правым операндом может быть любой тип данных, для которого определен поток ввода.

```
istream& operator>>(short& n);
istream& operator>>(unsigned short& n);
istream& operator>>(int& n);
istream& operator>>(unsigned int& n);
istream& operator>>(long& n);
istream& operator>>(unsigned long& n);
istream& operator>>(float& f);
istream& operator>>(double& f);
istreaai& operator>>(long double& f);
istream& operator>>(void*& p);
```

По умолчанию операция >> пропускает символы-заполнители (по умолчанию – пробельные символы), 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;
```

Последний оператор приводит к тому, что программа пропускает ведущие символы-заполнители и считывает целое число в переменную i. Затем она игнорирует любые символы-заполнители, следующие за целым числом, и считывает переменную с плавающей точкой d.

Для переменной типа char* (рассматриваемой 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 '\n'.

27.4. Форматирование потока

Для управления форматированием ввода-вывода предусмотрены три вида средств: ***форматирующие функции, флаги*** **и** ***манипуляторы***. Все эти средства являются членами класса 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). Если же выводимое значение имеет больше символов, чем ширина отведенного ему поля, то поле будет расширено до нужного размера. Эта функция имеет следующие прототипы:

```
int width(int wide);
int width() const;
```

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 десятичной точки), с которой выводятся числа с плавающей точкой. По умолчанию числа с плавающей точкой выводятся с точностью, равной шести цифрам. Функция precision () имеет следующие прототипы:

```
int precision(int  prec);
int precision() const;
```

Функция с первым прототипом устанавливает точность в prec и возвращает предыдущую точность. Функция со вторым прототипом возвращает текущую точность.

Функция fill() позволяет прочесть или установить символ-заполнитель. Она имеет следующие прототипы:

```
char fill(char type ch);
char 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.

Рассмотрим пример программы, в котором используются форматирующие функции:

```
void main() { double x;
cout.precision(4);
cout.fill('0');
cout << "   x     sqrt(x)   x^2\n\n";
for (x=1.0; x< 6.5; x++) {
cout.width(7);
cout << x << " ";
cout.width(7);
cout << sqrt(x) << " ";
cout.width(7);
cout << x*x << '\n';
}
}
```

Эта программа выводит на экран небольшую таблицу значений переменной x, ее квадратного корня и квадрата:

```
   x        sqrt(x)    x^2
0000001  0000001  0000001
0000002  01.4142  0000004
0000003  01.7321  0000009
0000004  0000002  0000016
0000005  02.2361  0000025
0000006  02.4495  0000036
```

С каждым потоком связан набор ***флагов,*** которые управляют форматированием потока. Они представляют собой битовые маски, которые определены в классе ios как данные перечисления.

Флаги форматирования и их назначение приведены в табл. 6.

Таблица 6

Флаги форматирования и их назначение

Флаг	Назначение
hex	Значения целого типа преобразуются к основанию 16 (как шестнадцатеричные)
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 (то есть nnn.ddd)
scientific	Числа с плавающей точкой выводятся в так называемой научной записи (то есть n.xxxEyy)
showbase	Выводится основание системы счисления в виде префикса к целому числовому значению (например, число 1FE выводится как 0x1FE)
showpos	При выводе положительных числовых значений выводится знак плюс
upper case	Заменяет определенные символы нижнего регистра на символы верхнего регистра (символ ”е” при выводе чисел в научной нотации на ”Е” и символ ”х” при выводе 16-ричных чисел на ”Х”)
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 пе-

	ревода на новую строку) отбрасываются
stdio	Потоки stdout, stderr очищаются после каждой операции вставки
unitbuf	Очищаются все выходные потоки после каждой операции вставки в поток
stdio	Очищаются stdout, stderr после каждой операции вставки в поток

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:

long flags();

Функция flags() имеет и вторую форму, которая может использоваться для установки значений флагов. Для этого используется следующий прототип этой функции:

long flags(long fmtfl);

В этом случае битовый шаблон копирует 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:

long setf (long mask);

long setf (long fmtfl, long 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);

В качестве второго параметра функции setf() можно использовать следующие константы, определенные в классе ios:

```
static  const long basefield;      // = dec | oct | hex
static  const long adjustfield;    // = left | right | internal
static  const long floatfield;     // = scientific | fixed
```

Сбросить установленные флаги можно с помощью функции-члена unsetf() класса ios, имеющей следующий прототип:

```
void unsetf(long mask);
```

Она сбрасывает флаги, заданные параметром mask. Следующий пример демонстрирует некоторые флаги:

```
double  d = 1.321e9;
int  n = 1024;
void main(){
// Вывести значения
cout << ”d = ” << d << ‘\n’ << ”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’;
}
```

При выполнении программа выводит на экран:

```
d = 1.321e+09
n = 1024
d = +1.321E+09
n = 400
```

Система ввода-вывода С++ предусматривает еще один способ форматирования потока. Этот способ основан на использовании *манипуляторов ввода-вывода.* Список манипуляторов и их назначение приведены в табл. 7. 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 7

Манипуляторы ввода-вывода и их назначение

Манипулятор	Использование	Назначение
dec	Ввод-вывод	Устанавливает флаг dec
endl	Вывод	Вставляет символ новой строки и очищает буфер
ends	Вывод	Вставляет символ конца
flush	Вывод	Очищает буфер потока
hex	Ввод-вывод	Устанавливает флаг hex
oct	Ввод-вывод	Устанавливает флаг oct
resetiosflags (iosbase::long mask)	Ввод-вывод	Сбрасывает ios-флаги в соответствии с mask
Setbase (int base)	Ввод-вывод	Задает основание системы счисления для целых (8, 10, 16)
Setfill (int c)	Ввод-вывод	Устанавливает символ-заполнитель
setiosflags (iosbase::long mask)	Ввод-вывод	Устанавливает ios-флаги в соответствии с mask
setprecision (int n)	Ввод-вывод	Устанавливает точность чисел с плавающей точкой
setw(int n)	Ввод-вывод	Устанавливает минимальную ширину поля
ws	Ввод	Устанавливает пропуск символов-заполнителей

За исключением setw() , все изменения в потоке, внесенные манипулятором, сохраняются до следующей установки.

Для доступа к манипуляторам с параметрами необходимо включить в программу стандартный заголовочный файл iomanip.h. 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 <math.h>
void main() { double x;
cout << setprecision(4);
cout << setfill('0');
cout << "   x     sqrt(x)   x^2\n\n";
for (x=1.0; x < 6.5; x++) { cout << setw(7) << x << " ";
cout << setw(7) << sqrt(x) << " ";
cout << setw(7) << x*x << "\n";
}
}
```

Этот пример функционально полностью эквивалентен приведенному ранее, но для управления форматом вывода использует манипуляторы, а не функции форматирования.

Манипулятор setw(), как и форматирующая функция width(), может помочь избежать переполнения строки-приемника при вводе символьных строк:

```
const int SIZE = 50;
. . .
char array[SIZE];
cin>>setw(sizeof(array));  // Или cin.width(sizeof(array));
                           // Ограничивает число вводимых симво-
лов
. . .                      // и позволяет избежать выхода
                           // за границу массива.
cin >> array;
```

27.5. Файловый ввод-вывод с использованием потоков

Для осуществления операций с файлами предусмотрено 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.h.

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(). Можно 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:

```
ifstream();
ofstream();
fstream();
```

создать поток, открыть файл и связать поток с файлом:

```
ifstream(const char *name, ios::openmode mode = ios::in) ;
ofstream(const char* name, ios::openmode mode=ios::out | ios::trunc);
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.txt”);
```

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. 8.

Таблица 8

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, то есть удалить его содержимое
ios::binary	Двоичный режим операций

Режимы открытия файла представляют собой битовые маски, поэтому можно задавать два или более режима, объединяя их побитовой операцией ИЛИ. В следующем фрагменте кода файл открывается для вывода с помощью функции open():

```
ofstream ofs;
ofs.open(”FileName.txt”);
```

Обратим 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 оператора

```
ofstream ofs(”FName.txt”, ios::out | ios::app);
```

написать

```
ofstream ofs(”FName.txt”, ios::app);
```

Между режимами открытия файла ios::ate и ios::app имеется небольшая разница.

Если файл открывается в режиме добавления, весь вывод в файл будет осуществляться в позицию, начинающуюся с текущего конца файла, безотносительно к операциям позиционирования в файле. В режиме открытия ios::ate (от английского ”at end”) можно изменить позицию вывода в файл и осуществлять запись, начиная с нее. Для потоков вывода режим открытия эквивалентен ios::out | ios::trunc, то есть

можно 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, будет возвращать 0:

```
if (!ofs){ cout << ”Файл не открыт\n”;}
```

Проверить успешность открытия файла можно также с помощью функции-члена is_open(), имеющей следующий прототип:

```
int is_open() const;
```

Функция возвращает 1, если поток удалось связать с открытым файлом. Например,

```
if (!ofs.is_open()){ cout << ”Файл не открыт\n”; return; }
```

Если при открытии файла не указан режим ios::binary, файл открывается в текстовом режиме и после того, как файл успешно открыт, для выполнения операций ввода-вывода можно использовать операторы извлечения и вставки в поток. Для проверки, достигнут ли конец файла, можно использовать функцию ios::eof(), имеющую прототип int eof();

Завершив операции ввода-вывода, необходимо закрыть файл, вызвав функцию-член close() с прототипом void close():

```
ofs.close();
```

Закрытие файла происходит автоматически при выходе потокового объекта из области существования, когда вызывается деструктор потока.

Рассмотрим пример, демонстрирующий файловый ввод-вывод с использованием потоков:

```
#include <iostream>
#include <fstream.h>
void main(){
int n = 50;
// Открываем файл для вывода
ofstream ofs(”Test.txt”);
if (!ofs) {cout << ”Файл не открыт.\n”; return;}
ofs << ”Hello!\n” << n;
```

```
// Закрываем файл
ofs.close();
// открываем тот же файл для ввода ifstream file(”Test.txt”);
if (!file) {cout << ”Файл не открыт.\n”;
return;}
char str[80];
file >> str >> n;
cout << str << ” ” << n << endl;
// Закрываем файл
file.close();
}
```

27.6. Неформатируемый ввод-вывод

Когда файл открывается 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 необходимо использовать файл вывода для последующего ввода в программу (возможно, другую), лишние байты информации ни к чему. С этой целью система ввода-вывода предоставляет возможность осуществления неформатируемого ввода-вывода, то есть записи и чтения двоичной информации (иногда говорят – сырых данных). Для осуществления ввода-вывода в двоичном режиме нужно включить флаг ios::binary в параметр open_mode, передаваемый конструктору потока или функции open(). Чтение двоичной информации из файла осуществляется функцией read(), которая имеет следующие прототипы:

```
istream& read(char* s, int n);
istream& read(unsigned char* s, int n);
```

Здесь параметр s задает буфер для считывания данных, а параметр n – число читаемых символов.

Запись двоичных данных осуществляет функция-член write():

```
ostream& write(const char * s, int n);
ostream& write(const unsigned char * s, int 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.h>
void main() {
int x = 255;
char str[80] = ”Тестирование двоичного ввода-вывода.”;
// Открываем файл для вывода в двоичном режиме
ofstream  ofs(”Test.txt”);
if (!ofs) { cout << ”Файл не открыт.\n”; return; }
ofs.write((char*)&x, sizeof(int));
ofs.write((char*)&str, sizeof(str));
ofs.close();
// Открываем файл для вывода в двоичном режиме
ifstream  ifs(”Test.txt”);
if (!ifs) { cout << ”Файл не открыт.\n”; return; }
ifs.read((char*)&x, sizeof(int));
ifs.read((char*) str, sizeof(str));
cout << x << ‘\n’ << str << ‘\n’;
}
```

27.7. Часто применяемые функции

Помимо уже описанных функций, библиотека ввода-вывода С++ 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 следующие прототипы:

```
int get();
istream& get(signed char*, int len, char delim= '\n');
istream& get(unsigned char*, int len, char delim= '\n');
istream& get(unsigned char &);
istream& get(signed char &);
istream& get(streambuf &, char delim= '\n');
```

Функция get() в первой форме возвращает код прочитанного символа или -1, если встретился конец файла ввода (ctrl/z).

Приведем пример использования функции get ():

```
#include <iostream>
void 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;}
```

Для вставки символа в поток вывода используется функция put() с прототипом ostreaiu& put(char ch). Функция get () может также использоваться для чтения строки символов. В этом случае используются ее варианты, в которых эта функция извлекает из входного потока символы в буфер str, пока не встретится символ-ограничитель *delim* (по умолчанию – перевод строки) или не будет прочитано (len-1) символов, или не будет прочитан признак конца файла. Сам символ-ограничитель не извлекается из входного потока.

Ввиду того, что функция get() не извлекает из входного потока символ-ограничитель, она используется редко. Гораздо чаще используется функция **getline**(), которая извлекает из входного потока символ-ограничитель, но не помещает его в буфер. Она имеет следующие прототипы:

```
istream& getline (char* str, int len, char delim);
istream& getline(char * str, int len);
```

Здесь параметры имеют те же назначения, что и в функции get().

Функция **gcount**() (с прототипом int gcount()const;) возвращает число символов, извлеченных из потока последней операцией неформатируемого ввода (то есть функцией get(), getline() или read()).

Рассмотрим пример, в котором используются две последние функции:

```
#include <iostream>
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, так как
// getline() не помещает ограничитель в буфер.
cout << ”\nЧисло прочитанных символов: ” << count – 1;}
```

Для того, чтобы пропустить при вводе несколько символов, используется функция **ignore**():

istream & ignore(int n = l, int delim = EOF);

Эта функция игнорирует вплоть до n символов во входном потоке. Пропуск символов прекращается, если она встречает символ-ограничитель, которым по умолчанию является символ конца файла. Символ-ограничитель извлекается из входного потока.

Функция **peek**(), имеющая прототип int peek(), позволяет ”заглянуть” во входной поток и узнать следующий вводимый символ. При этом сам символ из потока не извлекается.

С помощью функции **putback**() (с прототипом istream &putback(char 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.

Она неявно используется манипулятором **endl**. Этот манипулятор вставляет в поток символ перевода строки и очищает буфер. Таким образом, оператор

cout << endl;

эквивалентен следующим:

cout << ‘\n’; cout.flush();

Функция **rdbuf**() позволяет получить указатель на связанный с потоком буфер. Эта функция имеет прототип streambuf * rdbuf() const;

Наконец, функция **setbuf**() с прототипом void setbuf(char * buf, int n) позволяет связать с потоком другой буфер. Здесь buf – указатель на другой буфер длины n.

27.8. Файлы с произвольным доступом

Произвольный доступ в системе ввода-вывода реализуется с помощью функций seekg() и seekp(), используемых для позиционирования, соответственно, входного и выходного потока. Каждая из них имеет по два прототипа:

```
istream& seekg(long pos);
istream& seekg(long pos, seek_dir dir);
ostream& seekp(long offset);
ostream& seekp(long offset, seek_dir 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 в соответствии с определением из класса ios:

```
enum seek_dir { beg, cur, end };
```

Здесь константы перечисления определяют:

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:

```
long tellg ();

long tellp();
```

Следующий пример демонстрирует возможность позиционирования потока ввода информации:

```
#include <iostream>
#include <fstream.h>
void main(int argc, char* argv[]){ int size = 0;
if (argc > 1){ const char *FileName = argv[1];
ofstream of;
of.open( FileName, ios::binary );
for(int i = 0; i<100; i++) of.put ((char)(i+27));
of.close();
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<<endl;
}}
else cout << "Вы не задали имя файла.";
}
```

Программа выводит на экран длину заданного файла.

27.9. 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. 9.

Таблица 9

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 прототип iostate rdstate() const.

Функция возвращает состояние операции ввода-вывода. Во-вторых, можно воспользоваться одной из следующих функций-членов:

```
int good() const;
int eof() const;
int fail() const;
int bad() const;
```

Каждая из этих функций возвращает 1, если установлен соответствующий бит состояния (точнее, функция fail() возвращает 1, 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 setsfate():

```
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 int operator !() const; (он возвращает 1, если установлен бит badbit). Это позволяет сравнивать выражения, в которые входит поток или его отрицание с нулем, то есть писать выражения вида:

```
while(!strm.eof()) {
... }
```

Следующий пример иллюстрирует получение информации о состоянии ввода-вывода.

```
#include <iostream>
#include <fstream.h>
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 \n”; 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.

27.10. Переопределение операций извлечения и вставки

Одним из главных преимуществ потоков ввода-вывода является их расширяемость для новых типов данных. Можно реализовать операции извлечения и вставки для своих собственных типов данных. Чтобы избежать неожиданностей, ввод-вывод для определенных пользователем типов данных должен следовать тем же соглашениям, которые ис-

пользуются операциями извлечения и вставки для встроенных типов данных. Рассмотрим пример переопределения операций извлечения и вставки в поток для определенного 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 член типа tm, который представляет собой структуру для хранения даты и времени, определенную в заголовочном файле time.h. Чтобы осуществить ввод-вывод пользовательского типа данных, какими являются объекты класса Date, нужно переопределить операции извлечения и вставки в поток для этого класса. Приведем соответствующее объявление класса Date:

```
class Date {
tm tm_date;
friend istream& operator>>(istreamfi &is, Date txt);
friend ostream& operator<<(ostream& os, const Date& txt);
public:
Date(int d, int m, int y);
Date(tm t);
Date();
tm tm_date;
friend istream& operator>>(istreamfi &is, Date txt);
friend ostream& operator<<(ostream& os, const Date& txt);
};
```

Реализуем операции извлечения и вставки для объектов класса Date.

Возвращаемым значением для операции извлечения (и вставки) должна являться ссылка на поток, чтобы несколько операций 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, операция извлечения должна быть объявлена как дружественная функция класса. Ниже приведена операция извлечения из потока для класса Date:

```
istream.& operator>>(istream& is, Date& txt){
is >> txt.tm_date.tm_mday;
is >> txt.tm_date.tm_mon;
is >> txt.tm_date.tm_year;
return is;
}
```

Те же самые замечания верны и для операции вставки. Она может быть построена аналогично. Единственное отличие заключается в том, что в нее нужно передать константную ссылку на объект типа Date, поскольку операция вставки не должна модифицировать выводимые объекты. Ниже приведена ее реализация для класса Date:

```
osiream& operator<<(ostream& os, const Date& txt){
os << txt.tm_date.tm_mday <<'/';
os << txt.tm_date.tin_mon << '/';
os << txt.tm_date.tm_year;
return os;
}
```

Следуя соглашениям о вводе-выводе для потоков, теперь можно 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 <time.h>
class Date {
tm  tm_date;
public:
Date(int d, int m, int y) {
tm_date.tm_mday = d; tm_date.tm_mon = m; tm_date.tm_year = y;
};

Date (tm t){tm_date = t;};
Date() { tm_date.tm_mday = 01;
tm_date.tm_mon = 00;
tm_date.tm_year = 00; }
friend ostream& operator<<(ostream& os, const Date& txt);
friend istream& operator>>(istream& is, Date& txt);
};
istream& operator>>(istream& is, Date& txt){
is >> txt.tm_date.tm_mday;
is >> txt.tm_date.tm_mon ;
is >> txt.tm_date.tm_year ;
return is;
}
ostream& operator<<(ostream& os,const Date& txt){
os << txt.tm_date.tm_mon << '/' ;
os << txt.tm_date.tm_mday << '/' ;
os << txt. 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 переопределение операций извлечения и вставки в поток, на этот раз для структуры:

```
struct info{
char *name;
double val;
char *units;
info(){
val=0;
name = new char [30];
units = new char [30];
name[0]=units[0]=0;
}
};
ostream& operator << (ostream& s, info &m){ // Вывод info в s
s << m.name <<"  "<<m.val <<"  "<< m.units<<"\n";
return s;
}
```

Операция >> может быть переопределена следующим образом:

```
istream& operator >> (istream& s, info &m){ // Ввод в info
s.width(30);   s >> m.name;
char c;
while((c = s.get())!= ' ' && c!='\t' && c!='\n');
s.putback(c);
s.width(30);
s >> m.val;
s.width(30);
s >> m.units;
return s;
}
```

Для считывания строки ввода, такой как "**Resistance 300 Ohm**", можно использовать следующую запись:

```
void main(){
clrscr();
info m;
cout<<"Введите наименование величины, ее значение \n";
```

```
cout<<" и единицу измерения (и нажмите Enter.):\n";
cin >> m;             // Переопределенная операция >>
cout << m;            // Переопределенная операция <<
}
```

При выполнении этой программы диалог на экране монитора может выглядеть следующим образом:

```
Введите наименование величины, ее значение
и единицу измерения (и нажмите Enter.):
Resistance 300 Ohm
Resistance 300 Ohm
```

27.11. Переадресация ввода-вывода

Можно 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.h>
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.

Литература

1. Г. Буч, Р. А. Максимчук, М. У. Энгл, Б. Дж. Янг, Д. Коналлен, К. А. Хьюстон. Объектно-ориентированный анализ и проектирование с примерами приложений. Третье издание. .– Объектно-ориентированный анализ и проектирование с примерами приложений. Третье издание. – М: Вильямс, 2010. – 720с.
2. Б. Керниган, Д. Ритчи. Язык программирования Си. 2-е издание. – М.: Вильямс , 2013. – 304 с. И. В.
3. Т. А. Павловская. С/С++. Процедурное и объектно-ориентированное программирование. СПб: Питер, 2015. – 495с.
4. Ашарина. Объектно-ориентированное программирование в С++: лекции и упражнения, 2-е изд. М: Горячая Линия – Телеком, 2015. –. 335 с.
5. П. Лукас. С++ под рукой. – Киев: ДиаСофт, 2003. – 480 с.
6. С. Прата. Язык программирования С++. 6 издание. – М. Вильямс, 2011. – 1244 с.
7. Х.М. Дейтел., П.Дж.Дейтел. Как программировать на Си++. - М.: ЗАО БИНОМ, 1999. – 1000 с.
8. У. Топп, У. Форд. Структуры данных в Си++. – М.: ЗАО БИНОМ, 1999. – 800 с.
9. Г. Шилдт. Самоучитель С++, 3-е издание. – СПб: БХВ-Петербург, 2006. – 688 с.
10. Р. Лафоре. Объектно-ориентированное программирование в С++. СПб: ПИТЕР, 2004.- 844 с.
11. Б. Карпов, Т. Баранова. С++: Специальный справочник. – СПб: Питер, 2001. – 480 с.
12. Борис Пахомов. С/С++ и MS Visual C++ 2012 для начинающих. – СПб: БХВ-Петербург, 2013. – 512 с.

13. Б. Страуструп. Язык программирования C++. Специальное издание. – М: Бином, 2011. – 1136 с.

14. Ирэ Пол. ООП с использованием C++. – Киев: ДиаСофт, 1995, 1999. – 480 с.

15. Цимбал А.А., Майоров А.Г., Козодоев М.А. Турбо C++: язык и применение. – М.: "Джен АЙ Лтд", 1993. – 512 с.

16. С. Дьюхарст, К. Старк. Программирование на C++. – Киев: НИПО "ДиаСофт", 1993. -272 с.

17. У. Сэвитч. C++ в примерах. – М.: ЭКОМ, 1997. – 736 с.

18. Р. Вайнер, Л. Пинсон. C++ изнутри. – Киев: НИПО "ДиаСофт", 1993. – 304 с.

Содержание

1. ВВЕДЕНИЕ 3

2. ЭЛЕМЕНТЫ ЯЗЫКА 5

2.1. Первые программы 5

2.2. Алфавит языка 6

2.3. Комментарии 8

2.4. Типы данных 8

2.5. Целые типы данных 9

2.6. Плавающие типы данных 12

2.7. Константы-строки, или литералы 12

2.8. Директива препроцессора *define* 13

2.9. Описания 13

2.10. Модификатор *const* 14

3. ВЫРАЖЕНИЯ 14

3.1. Операция и выражение присваивания 14

3.2. Арифметические операции 16

3.3. Операции отношения 16

3.4. Логические операции 16

3.5. Побитовые операции 17

3.6. Сдвиги 19

3.7. Операции автоувеличения и автоуменьшения ++ и – – 20

3.8. Тернарная или условная операция 21

3.9. Операция следования 21
3.10. Приоритеты операций и порядок вычисления 22
4. ОПЕРАТОРЫ 24
4.1. Пустой оператор 24
4.2. Оператор-выражение 24
4.3. Составной оператор 24
4.4. Объявления 24
4.5. Условный оператор 24
4.6. Оператор выбора *switch* 25
4.7. Оператор цикла *while* 28
4.8. Цикл с постусловием *do-while* 29
4.9. Оператор *for* 29
4.10. Оператор безусловного перехода 31
4.11. Оператор *break* 32
4.12. Оператор *continue* 32
4.13. Оператор *return* 32
5. УКАЗАТЕЛИ 33
5.1. Определение указателей 33
5.2. Указатели и массивы 34
5.3. Адресная арифметика 36
5.4. Символьные массивы и строки 37
5.5. Многомерные массивы 38
5.6. Указатели и многомерные массивы 40

6. ОПЕРАЦИЯ SIZEOF 41

7. ОПЕРАЦИИ ДЛЯ РАБОТЫ С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ 42

7.1. Операция выделения памяти *new* 42

7.2. Операция освобождения памяти *delete* 43

8. ОБЪЯВЛЕНИЯ И ОПРЕДЕЛЕНИЯ 45

9. ОБЛАСТЬ СУЩЕСТВОВАНИЯ ИМЕНИ 46

9.1. Компиляция, компоновка, библиотеки 46

9.1.1. Компиляция и ее фазы 48

9.1.2. Модульное программирование, компоновка 49

9.2. Виды областей существования имени 50

10. ОБЛАСТЬ ВИДИМОСТИ 52

11. КЛАССЫ ПАМЯТИ 54

12. ОБЪЯВЛЕНИЯ ОБЪЕКТОВ И ТИПОВ 56

13. ИМЕНА ТИПОВ 57

14. СИНОНИМ ИМЕНИ ТИПА 58

15. ПРАВИЛА ПРЕОБРАЗОВАНИЯ СТАНДАРТНЫХ ТИПОВ 59

15.1. Явные преобразования 59

15.2. Неявные преобразования стандартных базовых типов 60

15.3. Преобразование производных стандартных типов 61

16. ФУНКЦИИ 62

16.1. Определение и вызов функции 62

16.2. Функции. Передача аргументов 64
16.3. Передача многомерных массивов 67
16.4. Указатели на функции 70
16.5. Ссылки 72
16.6. Ссылки в качестве параметров функций 73
16.7. Рекурсивные функции 74
16.8. Аргументы по умолчанию 76
16.9. Перегрузка функций 77
16.10. Шаблоны функций 79

17. ПЕРЕЧИСЛЕНИЯ 82

18. КЛАССЫ 85
18.1. Объявление классов 85
18.3. Деструкторы 89
18.4. Статические члены класса 95
18.5. Указатель *this* 97
18.6. Статические функции-члены 98
18.7. Указатели на члены класса 99
18.8. Инициализация данных–членов класса 100
18.9. Конструктор копирования и операция присваивания 102
18.10. Дружественные функции 105
18.11. Конструктор и операция *new* 108
18.12. Вызов деструктора 109

19. ПРОИЗВОДНЫЕ КЛАССЫ 112

19.1. Построение производного класса 112

19.2. Защищенные члены класса 113

19.3. Управление уровнем доступа к членам класса 114

19.4. Последовательность вызова конструктора и деструктора при построении производного класса на основе одного базового 115

19.5. Преобразования типов 116

20. ПОЛИМОРФИЗМ 119

20.1. Раннее и позднее связывание 119

20.2. Виртуальные функции 120

20.3. Абстрактные классы 126

21. ПЕРЕОПРЕДЕЛЕНИЕ СТАНДАРТНЫХ ОПЕРАЦИЙ 128

21.1. Основные определения и свойства 128

21.2. Операции *new* и *delete* при работе с абстрактными типами 132

21.3. Использование *new* при создании динамического объекта абстрактного типа 133

21.4. Операция *delete* 134

21.5. Преобразование типов 134

22. НЕКОТОРЫЕ ОСОБЕННОСТИ ПЕРЕОПРЕДЕЛЕННЫХ ОПЕРАЦИЙ 138

22.1. Операция = 138

22.2. Операция [ ] 140

23. КЛАССЫ И ШАБЛОНЫ 147

24. СПИСКИ 150

24.1. Операции над односвязными списками 153

24.2. Двунаправленные и кольцевые списки 157

24.3. Операции над кольцевыми списками 158

25. СТЕКИ 160

25.1. Реализация стека через массив 160

25.2. Реализация стека через динамическую цепочку звеньев 162

26. ДВОИЧНЫЕ ДЕРЕВЬЯ 166

26.1. Определение и построение 166

26.2. Таблицы 170

27. ПОТОКОВЫЙ ВВОД-ВЫВОД 175

27.1. Классы потоков 176

27.2. Стандартные потоки 177

27.3. Операции помещения и извлечения из потока 180

27.4. Форматирование потока 182

27.5. Файловый ввод-вывод с использованием потоков 188

27.6. Неформатируемый ввод-вывод 192

27.7. Часто применяемые функции 193

27.8. Файлы с произвольным доступом 195

27.9. Опрос и установка состояния потока 197

27.10. Переопределение операций извлечения и вставки 198

27.11. Переадресация ввода-вывода 203

ЛИТЕРАТУРА 204

СОДЕРЖАНИЕ 206