ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

И.М. Егоров

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Учебное пособие

ТОМСК — 2007

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра промышленной электроники (ПРЭ)

И.М. Егоров

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Учебное пособие

2007

Егоров И.М.
Объектно-ориентированное программирование на C++: Учебное пособие. — Томск: Томский государственный университет систем управления и радиоэлектроники, 2007. — 180 с.

Учебное пособие предназначено для студентов специальности 210106 «Промышленная электроника», изучающих дисциплину «Объектно-ориентированное программирование». Материал, касающийся элементов прграммирования на C++, включенный в текст пособия, может использоваться студентами 1 курса в рамках изучения дисциплины «Информатика».

Пособие построено на базе лекционных курсов «Процедурно- ориентированное программирование» и «Объектно-ориентированное программирование» читаемых на протяжении 5 лет студентам специальности «Промышленная электроника».

© Егоров И.М., 2007
© Томский государственный университет систем управления и радиоэлектроники, 2007

ОГЛАВЛЕНИЕ

1 ВВЕДЕНИЕ

Существует такое определение: программирование — это создание типов и их интерпретация. На самом деле, любая программа оперирует только с битовыми последовательностями, преобразует их, порождает новые и т.д. Интерпретация значений битовых последовательностей может быть самая различная, соответственно, различны и правила выполнения операций над ними.

Длина битовой последовательности, интерпретация состояния ее битов и правила операций на ними целиком определены *типом данных*. Типы данных или, более кратко, просто *типы*, вводятся в программу заранее, до первого использования данных с тем, чтобы операции ними обрели необходимый смысл.

В программах принято выделять объекты двух категорий — данные (или переменные) и функции. С любым из них связан *адрес* его расположения в памяти, а с объектами — данными связано еще и его *значение* — битовая последовательность определенной длины, начинающаяся с заданного адреса.

Язык программирования выполняет коммуникационную функцию, обеспечивая связь между человеком и компьютером. С этой точки зрения он должен быть в достаточной мере удобен для пользователя и в то же время должен быть достаточно формальным для однозначного компилирования в машинную программу.

2 ЭЛЕМЕНТЫ ПРОГРАММИРОВАНИЯ НА С++

Язык Си был разработан в 70-е годы как язык системного программирования. При этом ставилась задача получить язык, обеспечивающий реализацию идей процедурного и структурного программирования и возможность реализации специфических приемов системного программирования. Такой язык позволил бы разрабатывать сложные программы на уровне, сравнимом с программированием на Ассемблере, но существенно быстрее. Эти цели, в основном, были достигнуты. Большинство компиляторов для Си сами написаны на этом языке, почти полностью написана на Си операционная система UNIX.

Недостатком Си оказалась низкая надежность разрабатываемых программ из-за отсутствия контроля типов. Попытка поправить дело включением в систему программирования Си отдельной программы, контролирующей неявные преобразования типов, решила эту проблему лишь частично.

На основе Си в 80-е годы был разработан язык Си++, вначале названный «Си с классами». Си++ практически включает язык Си и дополнен средствами объектно-ориентированного программирования. Рабочая версия Си++ появилась в 1983 г. С тех пор язык продолжает развиваться и опубликовано несколько версий проекта стандартов Си и Си++.

С++ — универсальный язык программирования, задуманный так, чтобы сделать программирование удобным для опытного программиста. За исключением второстепенных деталей С++ является надмножеством языка программирования Си.

Помимо возможностей, которые дает Си, С++ предоставляет гибкие и эффективные средства определения новых типов. Используя определения новых типов, точно отвечающих концепциям приложения, программист может разделять разрабатываемую программу на легко поддающиеся контролю части. Такой метод построения программ часто называют абстракцией данных. Информация о типах содержится в некоторых объектах типов, определенных пользователем. Такие объекты просты и надежны в использовании в тех ситуациях, когда их тип нельзя установить на стадии компиляции. Программирование с применением таких объектов часто называют объектно-ориентированным. При пра-

вильном использовании этот метод дает более короткие, проще понимаемые и легче контролируемые программы.

Ключевым понятием C++ является класс. Класс — это тип, определяемый пользователем. Классы обеспечивают скрытие данных, гарантированную инициализацию данных, неявное преобразование типов для типов, определенных пользователем, динамическое задание типа, контролируемое пользователем управление памятью и механизмы перегрузки операций. C++ предоставляет гораздо лучшие, чем в Си, средства выражения модульности программы и проверки типов. В языке C++ есть также усовершенствования, не связанные непосредственно с классами, включающие в себя символические константы, **`inline`** — подстановку функций, параметры функции по умолчанию, перегруженные имена функций, операции управления свободной памятью и ссылочный тип.

В C++ сохранены возможности языка Си по работе с основными объектами аппаратного обеспечения (биты, байты, слова, адреса и т.п.). Это позволяет весьма эффективно реализовывать типы, определяемые пользователем.

C++ и его стандартные библиотеки спроектированы так, чтобы обеспечивать переносимость. Имеющаяся на текущий момент реализация языка будет идти в большинстве систем, поддерживающих Си. Из C++ программ можно использовать Си библиотеки, и с C++ можно использовать большую часть инструментальных средств, поддерживающих программирование на Си.

2.1 Алфавит и лексемы языка C++

Алфавит языка C++ образован подмножеством символов ASCII. Это заглавные **A...Z** и строчные **a...z** буквы латинского алфавита, символ подчеркивания _ , воспринимаемый как дополнительная буква латинского алфавита, арабские цифры 0...9, специальные символы:

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

В литеральных строках (см. ниже) возможно использование символов второй части кодовой таблицы ASCII (символов с кодом 128 и выше), где обычно располагаются буквы национальных алфавитов, в частности, кириллицы, символы псевдографики и т.п.

Компилятор C++ ***регистрочувствительный***, т.е. способен различать в лексемах заглавные и строчные буквы. Отметим, что компилятор PASCAL таким свойством не обладает.

Из символов алфавита формируются смысловые единицы языка — ***лексемы***.

Существуют лексемы пяти видов:

– идентификаторы,
– служебные слова,
– литералы,
– операции,
– разделители.

В тексте программы лексемы отделяются друг от друга т.н. «обобщенными» пробелами, к которым относят собственно символы пробелов, символы табуляции и перевода строки. Сюда же относятся и комментарии в тексте программы. Сколько бы ни было подряд идущих обобщенных пробелов, они игнорируются компилятором и воспринимаются им лишь как единый разделитель лексем.

2.2 Идентификаторы

Идентификатор — это *имя типа* или *имя объекта* в программе. Идентификатором служит последовательность латинских букв и цифр произвольной длины. К буквам относится и символ подчеркивания _. Обязательное условие: *первый символ идентификатора должен быть буквой или символом подчеркивания*. Напомним, что заглавные и строчные буквы различаются. Хотя длина символьной последовательности идентификатора не ограничена, но, например, компилятор Borland C различает только первые 32 символа идентификатора.

Являясь именем вводимого в программу объекта или определяемого пользователем типа, идентификаторы могут строиться в достаточной мере произвольно, однако, необходимо учитывать, что идентификатор не должен совпадать со служебным словом

(см. ниже). Следует избегать применения идентификаторов с двумя лидирующими знаками подчеркивания, например **__TIME**, т.к. в этом случае есть риск попасть на имя т.н. *переменной окружения*, используемой в среде программирования.

Практический совет. Вводя идентификатор в качестве имени объекта или типа, желательно отразить в нем некоторое смысловое содержание, это поможет пользователю (не компилятору!) легче ориентироваться в тексте программы.

Примеры идентификаторов:

May_ident — правильный идентификатор

AbraKadabra — правильный идентификатор

Это_моеN — ***ошибка!,*** *применение русских букв в идентификаторах недопустимо*

5Ris ***ошибка!****, первым символом должна быть буква*

_alfa — правильный идентификатор

2.3 Литералы

К литералам относят константы, значение которых задается в программе. В составе языка C++ имеется следующий набор литералов:

- ***символьные константы,***
- ***целые арифметические константы,***
- ***арифметические константы с плавающей точкой,***
- ***строковые константы или строковые литералы.***

Иногда в литературе символьные и арифметические константы выделяют как самостоятельный вид лексем.

2.3.1 Символьные константы

Символьная константа задается символом ASCII, заключенным в одинарные кавычки. Например: **'A', 'g', '9','Ф', 'я'**. Значение символьной константы — код ASCII символа, ука-

занного в апострофах. Символьная константа занимает 1 байт памяти и имеет тип **`char`** (см. ниже).

В ряде случаев возникает необходимость задать символьной константой символ ASCII не имеющий графического представления. Такие символы, как правило, являются управляющими, их вывод на экран монитора, принтер или в текстовый файл вызывает изменение обычного режима вывода (осуществляется переход на следующую строку, переход к началу строки, сдвиг точки вывода на позицию табуляции и т.п.). Символьные константы, соответствующие управляющим символам изображаются с использованием лидирующей обратной косой черты:

Константа	Обозначение	Действие
\n	NL (LF)	перевод на следующую строку
\t	HT	горизонтальная табуляция
\v	VT	вертикальная табуляция
\b	BS	шаг назад («забой» предыдущего символа)
\r	CR	возврат каретки (переход к началу строки)
\f	FF	перевод формата (авторегистр)
\a	BEL	звуковой сигнал

Кроме управляющих символов по аналогичной схеме строятся константы соответствующие некоторым символам, хотя и имеющим графическое представление, но уже задействованным в синтаксисе представления констант:

Символ	Изображение	Константа
обратная дробная черта	\	\\
одиночная кавычка	'	\'
двойная кавычка	"	\"

Задать символьную константу с любым кодом ASCII можно таким образом:

'**`\xHH`**' или '**`\ooo`**' ,

где **`HH`** — шестнадцатеричное число в диапазоне 0–FF,

`ooo` — восьмеричное число в диапазоне 0–377.

Числа, следующие за знаком обратной черты — код ASCII соответствующего символа.

2.3.2 Целочисленные константы

Целочисленные константы могут быть заданы в виде целых чисел со знаком в десятичном, восьмеричном и шестнадцатеричном представлении.

Примеры:

`28953, -451` — десятичные константы;

`0347, -0176` — восьмеричные константы;

`0x1AF3, -0xAF89` — шестнадцатеричные константы.

Таким образом, признаком восьмеричной константы служит лидирующий нуль, а шестнадцатеричной — префикс 0x или 0X. Целочисленные константы могут иметь тип **`int, long, unsigned int, unsigned long`**.

2.3.3 Константы с плавающей точкой

Константы с плавающей точкой представляются в десятичной системе в двух формах:

в виде последовательности десятичных цифр с точкой, оделяющей целую часть числа от дробной:

314.156 –0.2718 0.0654;

в экспоненциальной форме с мантиссой и порядком:

`1.62E-10 ,` что эквивалентно 1.62×10^{-10}

`-3.173e4 ,` что эквивалентно -3.173×10^{4}

В экспоненциальном представлении разделитель мантиссы и порядка может быть представлен как заглавной буквой **`E`**, так и строчной **`e`**.

2.3.4 Литеральные константы

Литеральные константы или строки представляют собой последовательность символов ASCII, заключенных в двойные кавычки, например:

"Это литеральная строка" "C:\\Dir1\\dir2"

В составе литеральных констант могут содержаться любые символы ASCII, включая управляющие. Правила формирования символов аналогичны правилам построения символьных констант, рассмотренных выше.

Для обеспечения работы некоторых функций, оперирующих со строками, используется соглашение о завершении строки не отображаемым «нуль — терминатором» — байтом со значением 0 *(Внимание: не кодом символа '0'.!)*.

При записи в тексте программы длинных строк удобно их фрагменты размещать на разных строках при этом нужно уведомить компилятор о том, что строка продолжается символом обратной косой черты \ , например, запись

```
"Такая длин\
ная строка"
```

Будет воспринято компилятором как литерал

"Такая длинная строка"

2.4 Служебные слова

Служебные слова — это зарезервированные лексемы. Они служат в качестве имен предопределенных типов и объектов, являются компонентами операторов и т.п. В языке C++ имеются следующие служебные слова:

_asm	_ds	int	_seg
asm	else	interrupt	short
auto	enum	_interrupt	signed
break	_es	_loadds	sizeof
case	_export	long	_ss
catch	extern	_near	static
_cdecl	_far	near	struct
cdecl	far	new	switch
char	fastcall	operator	template
class	float	_pascal	this
const	for	pascal	typedef

continue	friend	private	union
_cs	goto	protected	unsigned
default	_huge	public	virtual
delete	huge	register	void
do	if	return	volatile
double	inline	_saverages	while

Здесь нет смысла пояснять значения каждого из перечисленных служебных слов, разъяснения будут приводиться далее по мере необходимости. Этот список зарезервированных слов C++ приведен для того, чтобы случайно не использовать эти слова в качестве имен переменных, поскольку, напомним еще раз, *вводимые пользователем идентификаторы не должны совпадать со служебными словами*. Здесь идет речь о полном совпадении слов с учетом регистра. Например, пользовательский идентификатор **`static`** недопустим из-за его совпадения со служебным словом, а идентификаторы **`Static`** и **`stAtic`** вполне приемлемы.

В редакторах исходного кода, используемых в интегрированных средах разработки (IDE), таких как Borland C++, Visual C и т.д. служебные слова в тексте программы автоматически выделяются либо цветом, либо насыщенностью шрифта.

2.5 Операции

Все преобразования значений объектов — переменных осуществляются с помощью операций. В зависимости от количества участвующих в операции объектов аргументов (***операндов***) операции подразделяют на ***унарные***, применяемые к одному операнду, ***бинарные***, использующие два операнда и ***тернарные*****,** работающие с тремя операндами. На C++ используются в основном унарные и бинарные операции. Тернарная операция на C++ всего одна — условное арифметическое выражение. Синтаксис записи основных операции с кратким пояснением их смысла приведен в следующей таблице.

Тернарная операция

Оператор	Описание	Пример
a?b:c	Условное выражение: если ***a*** не нуль, то ***b,*** иначе ***c***	x = a?b:c

Бинарные операции

Арифметические операции

Знак орерации	Описание	Пример
+	Сложение	x = x + z;
-	Вычитание	x = y - z;
*	Умножение	x = y * x;
/	Деление	x = y / z;
%	Остаток от деления целых чисел	r = i % j;

Поразрядные операции (применяются только к целым типам)

Знак орерации	Описание	Пример
&	Поразрядное И	c = a & b
|	Поразрядное ИЛИ	c = a | b
^	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ	c = a ^ b
<<	Поразрядный сдвиг кода влево	c = a << 3
>>	Поразрядный сдвиг кода вправо	c = a >> 4

Операции присваивания

Знак орерации	Описание	Пример
=	Присваивание	x = 10;

@=	Составное присваивание (здесь символом **@** обозначен любой из операторов: **+ , -, *, / ,& , |.**)	x @= a;
		то же, что и
		x = x @ a;

Логические операции

Знак орерации	Описание	Пример
&&	Логическое И	if (x && OxFF) {...}
||	Логическое ИЛИ	if (x || OxFF) (...)

Операции отношения

Знак орерации	Описание	Пример
==	Равно	if (x == 10) {...}
!=	Не равно	if (x != 10) {...}
<	Меньше	if (x < 10) {...}
>	Больше	if (x > 10) {...}
<=	Меньше или равно	if (x <= 10) {...}
>=	Больше или равно	if (x >= 10) {...}

Унарные операции

Знак орерации	Описание	Пример
*	Косвенная адресация (разыменование)	int x = *y;
&	Взятие адреса	int* x = &y;
~	Поразрядное НЕ	x &= ~0x02;
!	Логическое НЕ	if (!v) {...}
++	Инкремент	x++; или ++x;
--	Декремент	x--; или --x;

Операции доступа к элементам объектов

Знак орерации	Описание	Пример
::	Разрешение области видимости	MC::F();
->	Доступ к элементам объекта через указатель (косвенный доступ)	MC->F();
.	Доступ к элементам объекта через его имя (прямой доступ)	MC.F();

Из приведенной таблицы видно, что один и тот же символ может использоваться для обозначения различных операций. В этом случае смысл операции определяется ее контекстом. Для обозначения операций на C++ используются как одиночные символы, так и группы из двух символов, причем группа из двух одинаковых символов, обозначает совершенно иную операцию, нежели одиночный символ.

Признаком того, что лексема определяет операцию является наличие знака или группы знаков операции. Если операция опре-

делена двухсимвольной группой знаков, то ее знаки нельзя разрывать обобщенным пробелом. Например, вместо **a && b** нельзя записать **a & & b**, это будет воспринято как две последовательно идущие операции.

Существуют три операции, обозначаемых служебными словами:

sizeof — определение размера области памяти в байтах, занятой объектом (не функцией!) и определение фактического размера памяти, отводимого под тип

new — запрос на выделение памяти на этапе выполнения программы, достаточной для размещения объекта указанного типа.

delete — освобождение памяти, выделенной операцией **new;**

Примеры записи операций **sizeof**, **new** и **delete** приведены ниже:

s=sizeof A; — в **s** будет размер памяти в байтах, занимаемый объектом с именем A;

s=sizeof(int); — в **s** будет размер памяти в байтах, отводимой под тип **int**;

a=new double; — в **a** будет помещен начальный адрес динамически выделенной памяти, достаточной для размещения объекта типа **double**;

a=new double[10]; — в **a** будет помещен начальный адрес динамически выделенной памяти, достаточной для размещения массива из 10 объектов типа **double**;

delete a; — произойдет освобождение выделенной динамически памяти под объект с начальным адресом, записанным в **a**;

На C++ есть унарная операция явного преобразования типа, синтаксис которой имеет вид:

```
(type_new) T
```

Здесь **type_new** — имя любого типа, известного в программе к моменту выполнения операции, **T** — выражение типа, отличного от **type_new** . Эта операция предназначена для вре-

менной смены типа операнда. Так, например, если объявлена переменная **`int A`**, а по логике программы требуется представить ее значение как **`double`**, то значение выражения **`(double)A`** и даст желаемый результат.

2.6 Разделители

К символам разделителям языка C++ относятся:

Символ	Название	Назначение
{ }	Фигурные скобки	Выделение границ блоков (составных операторов)
[ ]	Квадратные скобки	Выделение индексов в массивах
()	Круглые скобки	Изменение порядка операций в арифметических выражениях. Выделение списка формальных и фактических параметров в функциях. Выделение параметров в заголовках условных операторов и операторов циклов.
;	Точка с запятой	Обозначение конца оператора.
,	Запятая	Разделение элементов в различных списках, например в списках параметров функций, инициализаторов и т.п.
:	Двоеточие	Отделяет метку от оператора. В заголовке производного класса обозначает начало списка базовых классов (только на C++). В заголовке функций — конструкторов обозначает начало списка инициализаторов (только на C++). В описаниях битовых полей отделяет имя поля от константы, задающей размер битового поля.

2.7 Комментарии в программе

Комментарии в программе предназначены для пользователя, компилятором текст комментариев полностью игнорируется. На C++ существует два способа выделения комментариев:

/* Текст комментария произвольной длины*/

// Текст комментария до конца строки.

Первый способ выделения комментариев унаследован от языка Си, второй используется только в C++. В первом случае текст комментария может занимать как часть строки, так и произвольное количество строк. Например:

```
int a=1, /*объект а инициализировали 1*/ b=100;
...
float d=3.7; /* Далее следует длинный
многострочный
комментарий, после которого
программа продолжается */ d *= 16.01;
...
```

Во втором случае текст комментария простирается от // до конца текущей строки, например:

```
f = A*cos(w*t); // далее до конца строки следует текст комментария
```

Кроме функций пояснения текста программы, комментарии удобно использовать при отладке программ. Закомментировав фрагмент текста программы, мы исключаем его из входного потока компилятора, без фактического удаления из файла.

Практический совет. Не пренебрегайте комментариями в тексте программы, они существенно помогают впоследствии разобраться в собственной программе, упрощают процесс сдачи программы преподавателю.

2.8 Типы и их описание

В C++ существуют *предопределенные* типы и типы, *определяемые пользователем*. Предопределенные типы «понимаются» компилятором без специального описания, имена предопределенных типов — это зарезервированные слова. Пользовательские

типы нуждаются в специальном *определении* — их описании через предопределенные типы, либо через ранее созданные пользовательские типы. Слово «пользовательский» в данном контексте отнюдь не означает, что тип введен пользователем в данный момент составляющим программу. Пользовательский тип может быть создан кем-то и когда-то, но его описание должно быть обязательно включено в текст программы. Любой объект программы должен быть описан до его первого использования в операторах, т.е. должен быть определен его тип. При описании переменных возможно проведение инициализации — присвоения значения непосредственно при создании объекта.

Типы в C++ подразделяют на основные и производные. Разница между ними проявляется при создании пользовательских типов: *в описании нуждается только основной тип, производные типы порождаются автоматически*.

По элементному составу типы делят на *скалярные*, *векторные* и *объектные*. Самыми элементарными являются скалярные предопределенные типы. Векторные типы (или массивы) образованы индексированным набором однотипных элементов. Объектные типы в общем случае представляют собой агрегат из элементов разного типа. Каждый элемент объектного типа имеет свое имя. На C++ элементами объекта могут быть функции.

В этом разделе мы рассмотрим предопределенные скалярные и векторные типы, рассмотрение объектных типов отложим до раздела, посвященного объектно-ориентированному программированию.

2.9 Основные скалярные предопределенные типы

Имена основных скалярных предопределенных типов C++ таковы:

Имя типа	Название
char	символьный тип
short	короткий целый
int	целый
long	длинный целый

float	плавающий обычной точности
double	плавающий двойной точности
long double	длинный плавающий двойной точности
void	«пустой» тип

Первые четыре типа используются для представления целых чисел, следующие три — для представления чисел с плавающей точкой (чисел с дробной частью).

2.9.1 Тип void

Тип **`void`** — специфический тип C++, он не применяется в качестве основного при создании объектов, используются лишь производные от него типы. Синтаксически **`void`** эквивалентен основным типам, но использовать его можно только в производном типе. Объектов типа **`void`** не существует.

С его помощью задаются указатели на объекты неизвестного типа или функции, не возвращающие значение.

```
void f();       // функция f не возвращает значения

void* pv;       // указатель на объект неизвестного типа
```

Указатель произвольного типа можно присваивать переменной типа **`void*`**. На первый взгляд этому трудно найти применение, поскольку для **`void*`** недопустимо косвенное обращение (разыменование). Однако, именно на этом ограничении основывается использование типа **`void*`**. Он приписывается параметрам функций, которые не должны знать истинного типа этих параметров. Тип **`void*`** имеют также бестиповые объекты, возвращаемые функциями.

Для использования таких объектов нужно выполнить явную операцию преобразования типа. Такие функции обычно находятся на самых нижних уровнях системы, которые управляют аппаратными ресурсами. Приведем пример:

```
void*  malloc(unsigned  size);// функция библиотеки Си,
                                  //предназначенная для выделения
                                  // динамической памяти на этапе
                                  // выполнения программы
void free(void*););// функция библиотеки Си,
                   //предназначенная для освобождения
                   //ранее выделенной динамической памяти

void f()             // распределение памяти в стиле Си
    {
      int* pi = (int*)malloc(10*sizeof(int));
              // выделеляется память под массив из 10 int
      char* pc = (char*)malloc(10);
              // выделеляется память под массив из 10 char

      //...
      free(pi);
      free(pc);
    }
```

На C++ все скалярные предопределенные типы, исключая void, являются арифметическими, т.е. соответствующие им значения могут интерпретироваться как числа и к ним применимы арифметические операции.

Обозначение: *(тип) выражение* — используется для задания операции преобразования выражения к типу (см. ниже), поэтому перед присваиванием pi тип void*, возвращаемый в первом вызове malloc(), преобразуется в тип int. Пример записан в архаичном стиле; лучший стиль управления размещением в свободной памяти реализован в C++ операцией new (см. ниже).

2.9.2 Объявление скалярных предопределенных типов

Директива объявления переменной состоит из имени типа и имени — идентификатора объявляемой переменной. Например:

```
int A, i; // объявлены переменные знакового целого
  // типа с именами A и i;
float ff1; // объявлена переменная вещественного типа
   // обычной точности с именем ff1;
char hh0,cd; // объявлены переменные символьного типа
```

// с именами **hh0** и **cd**;

long VV // объявлена переменная типа знакового длинного

// целого с именем **VV**;

float ff1; // объявлена переменная вещественного типа

// обычной точности с именем **ff1**;

На C++ все скалярные предопределенные типы, исключая void, являются арифметическими, т.е. соответствующие им значения могут интерпретироваться как числа и к ним применимы арифметические операции.

Отрицательные целые числа во внутреннем формате представляются в дополнительном коде, при этом их старший бит интерпретируется как знаковый: для отрицательных чисел он 1, для положительных 0. Такую интерпретацию состояния старшего бита можно включать и выключать, добавляя в объявлении перед именем типа прилагательное **signed** (знаковый) и **unsigned** (беззнаковый). При применении прилагательных к типу **int** имя типа можно опускать. Например:

unsigned UI, T; // объявлены переменные **UI** и **T**

//беззнакового целого типа;

signed I, k; // объявлены переменные **I** и **k** знакового целого типа;

unsigned long V // объявлена переменная **V** беззнакового

// длинного целого типа;

unsigned char ch; // объявлена переменная **ch** беззнакового

// символьного типа.

Прилагательное **signed** употребляют редко, т.к. этот режим включен по умолчанию. В переменных объявленных как **unsigned** старший бит интерпретируется обычным образом, представляемое число положительное, двоичный код — прямой.

В памяти компьютера переменные типов **int** и **long** располагаются так, что в первом байте, адрес которого совпадает с адресом переменной, располагаются младшие разряды двоичного кода числа, во втором и последующих байтах — старшие.

Можно запретить изменение значения объекта — данных в программе, употребив при его описании модификатор **const:**

```
const int c = 17;
const float e = 2.71828;
```

После такого объявления компилятор сам будет блокировать любую попытку изменения значений объекта, например, путем использования его в левой части оператора присваивания.

Очевидно, что значение константный объект может получить только путем инициализации при объявлении, далее это значение остается неизменным. Попытка создать константный объект без его инициализации вызывает ошибку на этапе компиляции.

2.9.3 Размеры памяти, занимаемые предопределенными типами

Переменная типа **`char`** имеет размер, естественный для хранения символа на данной машине, обычно, это — байт. Переменная типа **`int`** имеет размер, соответствующий целой арифметике на данной машине: 2 байта для 16-разрядных компьютеров и 4 байта для 32-разрядных.

Все перечисленные типы являются арифметическими, к ним применимы арифметические операции и операции сравнения, значения переменных этих типов в этом случае интерпретируются как числа. Диапазон целых чисел, которые могут быть представлены соответствующим типом, зависит от его размера:

Тип	Размер в байтах	Диапазон
char	1	$[-2^7, 2^7-1]$ или [–128, 127]
short	2	$[-2^{15}, 2^{15}-1]$ или [–32768, 32767]
int	2 или 4	см. **`short`** или см. **`long`**
long	4	$[-2^{31}, 2^{31}-1]$ или [–2147483648, 2147483647]
unsigned char	1	$[0, 2^8-1]$ или [0, 255]
unsigned short	2	$[0, 2^{16}-1]$ или [0, 65535]
unsigned int	2 или 4	см. unsigned short или unsigned long
unsigned long	4	$[0, 2^{32}-1]$ или [0, 4294967295]

Числа с плавающей точкой во внутреннем формате представляются в экспоненциальном формате: $\pm M\times 2^P$, где M — мантисса, при этом, если число $\neq 0$, то $1\le M<2$, P — порядок, целое число. Если число равно 0, то и мантисса и порядок равны нулю. Число F в плавающей форме выражается через биты своего представления следующим образом:

$$F=(-1)^s\cdot 2^p\left(1+\sum_{k=0}^{n-1}2^{k-n}\cdot q_k\right),\ \text{где}\ p=\sum_{k=0}^{m-1}2^k\cdot t_k-2^{m-1}-1$$

Здесь **p —** двоичный порядок,

n — число разрядов мантиссы,

m — число разрядов порядка,

s — значение знакового разряда,

$\mathbf{q_k}$ и $\mathbf{t_k}$ — значения двоичных разрядов мантиссы и порядка соответственно.

Параметры **n** и **m** для разных типов имеют следующие значения

float:	n = 23	m=8
double:	n = 52	m = 11
long double:	n = 63*	m = 15

*в представлении значений **`long double`** самый старший 64-й разряд мантиссы занят разрядом с весом 2^0, всегда установленным в 1. Для **`float`** и **`double`** этот разряд виртуальный, его наличие подразумевается, но физически он не существует.

Диапазоны представления чисел с плавающей точкой:

Тип	Размер в байтах	Диапазон
float	4	$[-2^{127},2^{127}]$ или $[-1.70\times10^{38}, 1.7\times10^{38}]$
double	8	$[-2^{1024}, 2^{1024}]$ или $[-1.8\times10^{308}, 1.8\times10^{308}]$
long double	10	$[-2^{16384}, 2^{16384}]$ или $[-1.19\times10^{4932}, 1.19\times10^{4932}]$

2.10 Производные типы

Если в программе введен тип, например, с именем **`type`** , то компилятор C++ сразу же начинает «понимать» производные от него типы:

type[]	вектор (массив) элементов типа **type**
type*	указатель на тип с именем **type**
type&	ссылка на тип с именем **type**
type()	функция, возвращающая значение типа **type**

Примеры объявлений объектов производных типов:

```
char v[10]   //объявлена переменная v - вектор (массив)
             //из 10 элементов типа char;
char* p      //объявлена переменная p - указатель на объект типа char;
int& u       //объявлена переменная u - ссылка на объект типа int;
int Fn()     //объявлена функция с именем Fn, возвращающая
             //значение типа int, не принимающая никаких аргументов
```

Техника работы с объектами основных и производных типов будет обсуждаться ниже в соответствующих разделах.

2.10.1 Векторные типы (массивы)

Вектором или массивом называется совокупность данных одинакового типа, объединенных одним именем с возможностью обращения к отдельному элементу по его номеру (индексу). Так, например, при объявлении объекта

type M[n];

где **type** – имя типа,

M – имя объекта,

n — положительная целочисленная константа

компилятор создает в памяти непрерывный сегмент, в котором последовательно один за другим расположены **n** элементов типа **type**. Доступ к элементам массива производится по индексу — целочисленной переменной или константе, указывающей номер элемента, например, **M[0], M[1], M[10].** *В C++ индексация массивов начинается с 0,* так что в нашем примере имеет смысл изменять индекс от 0 до n–1.

Внимание! На C++ нет автоматического контроля выхода индекса за границы массива, указанные при объявлении. Такой контроль возложен на программиста.

С именем массива M связывается адрес его первого элемента **M[0],** адрес i-го элемента определяется путем прибавления к этому адресу величины **i×sizeof (type).** Массивы могут

быть образованы из элементов любого типа, как предопределенного типа, так и типов, определяемых пользователем. Элементом массива может служить другой массив. По такому принципу на C++ строятся многомерные массивы. Синтаксис объявления двумерного массива:

int IM [12][10]

Даже по структуре объявления видно, что вводится 12 массивов по 10 элементов в каждом (или, что эквивалентно, 10 массивов по 12 элементов).

При объявлении массива обязательно в квадратных скобках указывается количество элементов в нем. Исключение из этого правила — ситуация, когда размер массива определяется автоматически из инициализирующего выражения. В этом случае в квадратных скобках можно ничего не писать, но наличие самих скобок обязательно. Например:

```
int C[] = {1,2,4,7};
char S[] = "Всем привет!";
```

Первой директивой создается массив из 4-х элементов типа **int** , которые сразу же после создания инициализируются значениями 1, 2, 4 и 7. Во втором случае создается массив элементов типа **char**, инициализированный литеральной константой. Размер массива, созданного второй директивой равен числу символов литеральной константы + 1 с учетом того, что последний символ литеральной константы — «нуль-терминатор».

2.10.2 Указатели C++

Указателем на C++ называют объекты значение, которых интерпретируется как адрес другого объекта. Можно выделить две категории указателей: ***указатели данных*** *и* ***указатели функций***. В пределах этого раздела рассматриваются только указатели на данные, указатели функций рассмотрены ниже.

Хотя адреса и представляют собой числа тождественные по формату числам типа **unsigned int** или **unsigned long**, указатели имеют свои собственные правила и ограничения на присваивания, преобразования и выполнение с ними арифметических действий.

2.10.2.1 Объявления указателей

На C++ практически не употребляется просто термин «указатель», а используется термин «указатель на тип». Тем самым подчеркивается, что указатель каким-то образом «типизирован», т.е. некоторые важные свойства указателя зависят от типа объекта, на который он указывает. Синтаксис объявления указателей следующий:

имя_типа *имя_указателя;

где ***имя_типа*** — имя типа объектов, адреса которых может хранить указатель;

имя_указателя — имя самого указателя, его идентификатор.

Важно, чтобы символ * располагался между именем типа и именем указателя. При этом неважно «прижат» ли этот символ к имени типа или к имени указателя и, вообще, не имеет значения сколько пробелов отделяют * от имени типа и от имени указателя.

Примеры объявления указателей на данные:

`char *ucc;` // объявлен указатель на тип **`char`** с именем **`ucc`**

`int* ui;` // объявлен указатель на тип **`int`** с именем **`ui`**

Поскольку указатель сам по себе является объектом, можно создать указатель на указатель, например:

`int ** uui;` // в **`uui`** может содержаться адрес указателя на тип **`int`**

Процесс «наращивания» числа звездочек можно продолжить, создавая указатель указателя на указатель и т.д.

Являясь производным типом, указатель может быть объявлен на любой ранее определенный тип. Более того, в самом определении пользовательского типа на основе структуры допустимо использовать в качестве поля указатель на тип, определяемый этой структурой, например:

`struct List` // определение пользовательского типа **`List`**

{

char IT[100];

List *next; // указатель на определяемый тип в
//составе полей этого типа
};

Размер памяти, отводимой под указатель, зависит от используемой модели памяти, размера адресного пространства компьютера. В 16-разрядных компьютерах указатели могут быть «ближними» — **near** и «дальними» — **far**, например:

char near* uc;
int far* U;

Указатель **near** имеет размер 2 байта и может хранить адрес в пределах 64 килобайтного сегмента, указатель **far** занимает 4 байта и содержит полный 32-х разрядный адрес. В 32-разрядных системах используются только 4-х байтовые указатели. Модификаторы **near** или **far** не обязательно указывать в объявлениях — один из них устанавливается по умолчанию, какой именно — указывается в опциях настройки компилятора.

2.10.2.2 Операции с указателями данных и адресная арифметика

Указатели на разные типы не совместимы. Это значит, что автоматического преобразования типов не предусмотрено и попытка простого присвоения значения указателя на один тип указателю на другой тип будет блокирована компилятором. Исключением из этого правила является указатель на тип **void.** Указателю на **void** можно присваивать значение указателя на любой тип, поэтому этот указатель называют родовым. Обратное — неверно: попытка присвоить значение указателя на **void** указателю на любой другой тип без явного преобразования типа невозможна. Ввиду того, что автоматическое приведение типов для указателей отсутствует, в случае необходимости присвоения значения указателя на тип **type1** указателю на тип **type2** следует использовать явное преобразование типов.

Пример:

type1 *T1; // объявлен указатель **T1** на тип **type1**
type2 *T2; // объявлен указатель **T2** на тип **type2**
void *V; // объявлен указатель **V** на тип **void**

T1 = T2; // **ошибка!** указатели на различные типы несовместимы

T1 = (type1*) T2; // присвоение значения указателя **T2** указателю

// **T1** с использованием операции явного

// преобразования типа

V = T1;// указателю на **void** можно присваивать

V = T2; // значения указателей на любой тип

T1 = V; // ошибка! если type1 не void

T1 =(type1*)V; // возможно с явным преобразованием типа

С указателями связаны две унарные операции: операция взятия адреса именованного объекта **&** и операция разыменования указателя *. Операция взятия адреса применяется для занесения значения адреса ранее введенного объекта в указатель, например:

int A; // введен объект с именем **A** типа **int;**

...

int *ua = &A; // введен **ua** - указатель на **int**

//и инициализирован адресом объекта **A**;

Разыменование указателя — это ссылка на значение переменной, адрес которой содержится в указателе. По сути, это операция косвенной адресации. Например, смысловое содержание директивы ***u = 34;*** таково: «объекту, адрес которого находится в **u** присвоить значение 34»

Из арифметических операций над указателями на один и тот же тип допустима только операция вычитания. Над указателями можно выполнять операции инкремента и декремента, к ним можно также прибавлять (вычитать) значения переменных и констант целого типа. Операции инкремента / декремента значения указателя и их сложение / вычитание с целыми числами ***выполняются специфически***. Пусть объявлен указатель **Ut** на тип с именем **type**: **type* Ut**, тогда операция **Ut++** вызывает увеличение значения указателя на величину **sizeof(type)** , а операция **Ut+=n**, где **n** — целое число, увеличивает значение указателя на **n*sizeof(type).**Обратим внимание на сходство арифметических операций с указателями с операциями над индексами мас-

сивов: добавление арифметической константы к указателю вызывает такое же изменение адреса, как увеличение на ту же величину индекса массива.

Внимание! Созданные и неинициализированные указатели опасны. В неинициализированном указателе содержится случайный код, воспринимаемый как адрес. Модификация значения объекта по такому адресу может привести к непредсказуемым последствиям.

В ряде случаев значение указателя нужно установить на такой адрес, которого заведомо не существует в адресном пространстве. В качестве такого адреса «в никуда» используют нулевое значение кода в указателе. Для этого специфического адреса введено мнемоническое обозначение **NULL**. Значение **NULL** эквивалентно значению арифметической константы 0.

2.10.2.3 Модификатор const и указатели

Модификатор **const** с указателями может использоваться в следующих контекстах.

```
const type * u1;            // объявлен указатель на константу
type *const u2 = &A;        // объявлен указатель - константа
const type *const u3=&A;    // объявлен указатель - константа
                            // на константу
```

В первом случае будет блокироваться попытка изменения значение объекта, на который указывает **u1**. Во втором случае запрещено изменение значения указателя **u2** (изменение адреса находящегося в нем). Поскольку изменение значения **u2** невозможно, он обязательно должен быть инициализирован при объявлении, что и сделано в примере. Отсутствие инициализирующего выражения при объявлении указателя — константы вызывает ошибку компиляции. Третий случай объединяет предыдущие два: нельзя изменять ни адрес, ни значение объекта, на который указывает указатель. В последнем случае фактически запрещено изменять значение объекта **A,** ссылаясь на него через указатель **u3**, хотя изменение значения **A** при ссылке на него по имени вполне допустимо.

2.10.3 Ссылки

На C++ существует производный тип ***ссылка на тип***. Объявление ссылки на тип с именем **type** выглядит следующим образом:

```
type & RT = AA;
```

где **type** — имя типа,

RT — имя ссылки,

AA — имя ранее определенной переменной типа **type**.

В отличие от указателя ссылку невозможно объявить без инициализации. После объявления ссылка становится еще одним именем инициализирующей переменной.

Например:

```
int V1;          // описание объекта V1 типа int
int &r = V1;     // описание объекта r типа ссылка на
                 // int с инициализацией V1
...
r = 12;          // действия этих операторов
V1=12;           // эквивалентны
```

Рассматривая ссылки вне связи с функциями, трудно понять (и объяснить), зачем одному и тому же объекту может понадобиться еще одно имя. Далее, рассматривая передачу параметров при вызове функций, покажем, сколь велики различия в механизме передачи параметров объявленных как ссылка.

2.11 Объектные типы: классы, структуры, объединения, битовые поля

Наряду с массивами, являющимися агрегатами элементов одного и того же типа, на C++ возможно построение агрегатных типов с элементами разного типа, включая элементы — функции.

Такие типы называют объектными. Существуют четыре вида объектных типов C++:

– класс;
– структура;
– объединение;
– битовые поля.

Объектные типы создаются пользователем и нуждаются в обязательном описании, т.к. необходимо указать из каких элементов они состоят. Такое описание типа, или, как говорят, его *определение* должно быть сделано до первого их использования в программе.

Классы C++ требуют отдельного рассмотрения, что будет сделано в разделах, посвященных объектно-ориентированному программированию. Сейчас же начнем рассмотрение только со структуры, причем в том их представлении, которое используется в Си. Дело в том, что классы и структуры C++ являются развитием структур Си. Класс и структура C++ — одно и то же, и различаются регламентом доступа к элементам по умолчанию.

2.11.1 Структуры

Структура на Си предназначена для описания составных объектных типов, имеющих в своем составе *поля* различного типа. В описание объектного типа, представленного структурой на C++, могут включаться и функции, но эту ситуацию мы рассмотрим позже, здесь же ограничимся рассмотрением структур, содержащих только поля данных. Описание структуры с полями имеет следующий вид:

```
struct тег_структуры
{
тип1 поле1;
тип2 поле2;
...
типN полеN;
};
```

Это т.н. *протокол описания структуры*. За ключевым словом **struct** следует *тег_структуры* — идентификатор, который впоследствии будет являться именем определяемого типа. Далее в блоке из фигурных скобок следуют операторы описаний полей, входящих в структуру. Протокол описания структуры должен обязательно заканчиваться точкой с запятой, следующей за завершающей закрывающей фигурной скобкой.

Поля могут быть образованы как основными, так и производными типами. Они могут быть скалярными, векторными (массивами), другими определенными ранее структурами, указателями и ссылками на различные типы.

Например:

```
struct Mst
{
int n;
double *D;
char Na [20];
};
```

После такого описания в программу вводится новый основной тип с именем **Mst,** вместе с ним возникают и все производные от него типы. Теперь это имя типа можно точно так же как использовались имена предопределенных типов **int, float, char** и т.п.

Например, объявлением **Mst A, B;** создаются две переменных типа **Mst** с именами **A** и **B**. Объявление **Mst *U;** создает переменную с именем **U** — указатель на тип **Mst**. У каждого из созданных объектов основного типа будут свои собственные поля с именами **n, D** и **Na**, обратиться к которым можно по составному имени, например:

```
A.n = 3;   // полю n объекта A присваивается значение 3
B.n = 156;   // полю n объекта B присваивается значение 156
A.Na[10] ='a';   // 10-му  элементу поля Na объекта A
                 // присваивается значение 'a'.
```

Таким образом, синтаксис обращения к полю объекта типа структуры по имени имеет вид:

имя_объекта.имя_поля

Если объект имеет поля объектного типа, то количество элементов, записываемых через точку, увеличивается. Например, пусть имеется описание типа с именем **FW**:

```
struct FW
{
struct {int IA; int JB} ST; // поле с именем ST,
                            // созданное на основе структуры
int fib;
};
```

и в дальнейшем создан объект с именем **OB** типа **FW**:

FW OB;

тогда добраться до составляющих **IA** и **JB** поля **ST** можно следующим образом:

```
OB.ST.IA = 67;   OB.ST.JB = 53;
```

Обратим внимание на описание типа поля **ST** в протоколе описания структуры **FW**: там использована структура без тега. Использование такой безымянной структуры не приводит к созданию нового типа, структура лишь описывает тип конкретного объекта с именем **ST**.

К полям переменной объектного типа можно обращаться через указатель. Пусть определен тип с именем **DF:**

```
struct DF
{
 char str[30];
 float F;
};
```

и созданы переменная объектного типа **DF** с именем **T** и указатель на тип **DF** с именем **UU**:

```
DF T ;
DF *UU= &T;
```

при этом указатель инициализирован адресом объекта **T**.

Теперь в **UU** содержится адрес объекта **T**, и к его полям можно обратиться через указатель, например:

UU -> F = 3.1415; UU -> str[14] = 'g';

Таким образом, синтаксис доступа к полям переменной объектного типа, адрес которой помещен в указатель, имеет следующий вид:

имя_указателя -> имя_поля;

Знак операции доступа здесь состоит из двух символов — (минус) и > (больше). В случае, когда поле объекта тоже является указателем на объект, определенный структурой, стрелок в выражении доступа может быть более одной.

Инициализация объектов типа структуры может быть произведена аналогично инициализации массива (см. выше), но с учетом типов и длины полей. Например, пусть определен тип **FF**:

```
struct FF
{
char N[6];
int A;
float ff;
char c;
};
```

Создание переменной с именем **TY** типа **FF** с одновременной инициализацией ее полей заданными значениями может выглядеть так:

```
  FF TY={'П','о','л','е',' ','N',321,7.895,'y'};
```

Список констант, перечисленных через запятую в фигурных скобках, представляет значения полей и их компонентов, которые

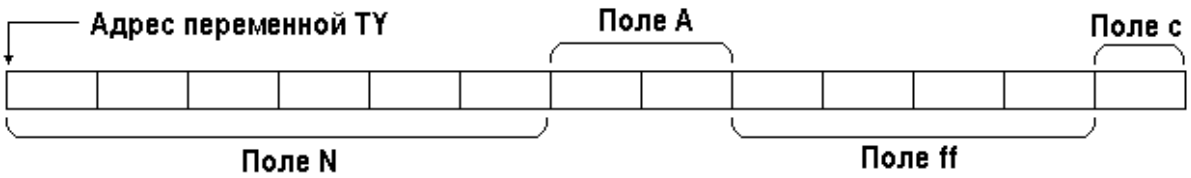
будут установлены при создании переменной **TY**. Поле, образованное символьным массивом можно инициализировать одной литеральной константой, например:

```
FF TY={"Поле N", 321, 7.895, 'y'};
```

Результат будет тот же, что и в первом примере инициализации. В данном случае литеральная константа поместится в поле **TY.N** без завершающего нулевого байта. Потеря завершающего нуля может привести к проблемам, если в дальнейшем поле **TY.N** будет использовано как строка. Если же размер поля будет недостаточен для занесения информационных символов литеральной константы, то на этапе компиляции будет выдана ошибка:

```
FF TY={"Поле N23",321,7.895,'y'};// ошибка!
                                // строка не помещается в поле TY.N
```

Поля переменной типа структуры располагаются в памяти последовательно. В нашем примере компоненты переменной **TY** будут расположены в памяти следующим образом:



Таким образом, поля структуры располагаются в байтах памяти последовательно друг за другом в соответствии порядком их записи в описании структуры и в соответствии с типом и длиной каждого поля. Начальный адрес первого по порядку записи поля совпадает с адресом самой объектной переменной. Размер памяти, занимаемой объектной переменной равен суммарному размеру всех ее полей.

В нашем примере

```
sizeof TY=sizeof TY.N+sizeof TY.A+sizeof TY.ff+sizeof
TY.c =13
```

2.11.2 Объединения

Объединения или, как их еще называют *смеси*, имеют синтаксис определения сходный со структурой, различие лишь в том, что вместо ключевого слова **`struct`** используется ключевое слово **`union`**:

`union` ***тег_объединения***

{

тип1 поле1;

тип2 поле2;

...

типN полеN;

};

Здесь также тег объединения становится именем типа, синтаксис доступа к элементам полностью аналогичен доступу к элементам структуры. Различие структуры и объединения в том, что *все поля объединения начинаются с одного и того же адреса*. Этот адрес совпадает с адресом объектной переменной. Поля в объединении перекрываются, их значения как бы «смешиваются», отсюда и происхождение второго названия объединений.

Пример описания объектного типа **`union`**:

```
union UN
{
char BY[4];
int I[2];
unsigned long L;
float F;
};
```

В этом примере произведено описание объектного типа с именем **`UN`**. Поля имеют различный тип, но одинаковую длину. Начинаясь с одного и того же адреса и имея одинаковую длину, все четыре поля расположены в одном и том же участке памяти и

адресуются к одной и той же битовой последовательности. Интерпретация же этой последовательности разная: если обращаться к ней через поле **BY**, последовательность будет интерпретирована как массив из 4-х символов, если обращение произведено через поле **I**, то содержимое последовательности будет воспринято как массив из 2-х целых чисел. Соответственно обращение к полю **L** — даст интерпретацию как значения переменной типа **unsigned long**, к полю **F** — как значения переменной типа **float**. Такая различная интерпретация одной и той же битовой последовательности может быть полезна в некоторых приложениях.

В приведенном примере все поля типа **UN** имели одинаковую длину, в общем случае, разумеется, длина полей может быть различной.

Размер памяти, занимаемый объектом типа **union,** равен максимальному размеру входящему в него поля.

2.11.3 Битовые поля

Минимальной единицей адресации в программах является байт. Непосредственная адресация к отдельному биту или произвольной группе битов невозможна. В некоторых важных приложениях потребность в таком доступе возникает. На C++ имеются средства, позволяющие организовать доступ к битовым группам — это битовые поля.

Объектный тип с битовыми полями имеет синтаксис определения почти тот же, что и тип, определяемый структурой, даже ключевое слово то же самое — **struct**:

struct ***тег_структуры***

{

целый_тип1 поле1 : целая_константа1;

целый_тип2 поле2 : целая_константа2;

...

целый_типN полеN : целая_константаN;

};

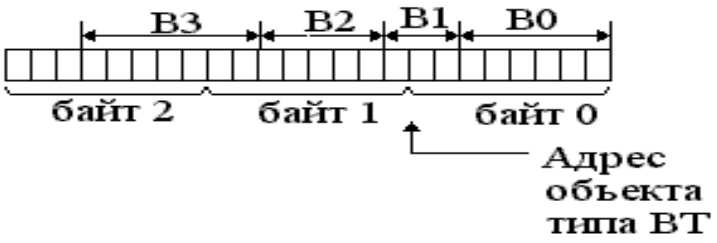
В отличие от описания объектного типа «структура», рассмотренного ранее здесь имеются следующие особенности:

- допускаются только следующие скалярные типы полей: **`char,unsigned char, int, unsigned int`**;
- после имени поля ставится разделитель : (двоеточие) и далее в форме целочисленной константы указывается ***длина битового поля в битах***.

Размер битового поля не может превышать числа битов в типе, имя которого указано слева . Это 8 для **`char`**, для **`int`** — либо 16, либо 32, в зависимости от реализации компилятора. Пусть определен тип **`BT`**, представляющий собой битовое поле:

```
struct BT
{
unsigned B0  : 6;
unsigned B1  : 3;
unsigned B2  : 5;
unsigned B3  : 7;
};
```

Расположение битовых полей типа BT в байтах адресного пространства показано на рисунке:



Первое по порядку следования в протоколе описания битовое поле (в нашем примере **`B0`**) располагается в *младших* битах первого байта адресного пространства, занимаемого объектом типа битового поля.

При размещении в памяти объекта типа битовое поле его размер выравнивается до целого числа байт. В нашем примере под объект типа **BT** отведено 3 байта.

2.12 Объявления и определения C++

На C++ существуют описания двух типов:

– ***объявления;***

– ***определения.***

Объявление содержит краткую характеристику объекта. При объявлении переменной приводится имя соответствующего ей типа и ее идентификатор. При объявлении функций дается только описание ее интерфейса (см. ниже). Определение дает более развернутую информацию. В определении переменной может присутствовать описание типа, если он определяется пользователем, и инициализатор, задающий начальные значения элементам объекта. Определение функции содержит текст ее программы (тело функции).

Примеры объявлений:

`int A;`	// объявлена переменная с именем **A**, // имеющая тип **int** (знаковый целый)
`char C[10];`	// объявлен вектор (массив) с именем **C**, состоящий из 10 элементов типа **char** (знаковый)
`float d, d1;`	// объявлены две переменные (списком), имеющие тип **float** (вещественный с плавающей точкой)
`struct SW;`	// объявлена структура с именем **SW**
`int Fn(float);`	// объявлена функция с именем **Fn**, возвращающая значение типа **int** и принимающая один параметр типа **float**.
`typedef unsigned WORD`	// для типа **unsigned int** объявлено новое имя типа **WORD**

Примеры определений:

`int A = 32;`	// определена переменная с именем **A**,
	// имеющая тип **int** и инициализирована
	// значением 32 при создании.

`float d=2.33,d1=1.75;`	// определены и инициализированы две // переменные типа **float**

`struct SW`
`{`
`  char C;`
`  float fl;`
`};`
	// определен пользовательский тип
	// - структура с именем **SW** с двумя
	// элементами: **C** - типа **char** и
	// **fl** - типа **float**

`int Fn(float x)`
`{`
`return x/7;`
`}`
	// определена функция **Fn**, возвращающая
	// целую часть от деления значения
	// параметра **x** типа **float** на 7.

Как следует из приведенных примеров, объявления переменных предопределенных типов **int** и **float** отличаются от соответствующих определений только отсутствием инициализаторов. Это различие не столь значимое, поскольку инициализацию можно заменить операцией присваивания. Для типов, создаваемых пользователем (в примере это структура **SW**) и функций (в примере это функция **Fn**), наличие определения обязательно и это определение должно быть только одно.

Необходимость введения объявлений вызвана тем, что они, как правило, короче определений, а информации в них достаточно, чтобы встроить объект в программу. Лаконичные объявления размещают впереди программы, обеспечивая предварительный уровень описания участвующих в ней объектов.

2.12.1 Объявление `typedef`

Директива **typedef** используется для создания ***нового имени типа***.

Например:

```
typedef unsigned long UINT;

typedef float VECT[3];
```

После отработки этой директивы в программу будут введены новые имена типов **UINT** и **VECT**. Теперь описание вида **UINT a, a1;** будет эквивалентно описанию **unsigned long a, a1,** а описание **VECT C1,C2;** будет эквивалентно директиве **float C1[3],C2[3].** Директива **typedef** *не создает новый тип*, типы объектов в примерах как были **unsigned long** и массив **float**, так и остались, сохранилась и интерпретация операций.

2.13 Выражения

Выражением называется последовательность операций, операндов и разделителей, задающих определенное вычисление. В выражениях образуются новые значения и изменяются значения переменных.

Выражения C++, подобно объектам, *имеют тип и значение* интерпретируемое в соответствии с этим типом. Примеры выражений:

`(a + d)/c - dd*k`	— арифметическое выражение;
`!(fn && Rd) || tp`	— логическое выражение;

`GG > T`
`H <= y`
`a == b`
`c != d`
	} — выражения отношения;

`F = d`	— выражение присваивания;
`a & b | c<<3`	— выражение с поразрядными операциями;
`cos(x)`	— выражение вызова функции;
`a`	— выражение — значение объекта;

Тип выражения зависит от типов операндов и смысла операций с этими типами.

2.13.1 Арифметические выражения

Арифметические выражения внешне похожи на обычные алгебраические формулы. Приоритет и порядок выполнения арифметических операций тот же, что и в алгебре. Высшим приоритетом пользуются *мультипликативные* операции: умножение * и деление / , сюда же относится и специальная операция нахождения остатка от деления целых чисел %. Низшим приоритетом обладают *аддитивные* операции: сложение + и вычитание -. Выполнение операций одинакового приоритета осуществляется слева направо в порядке их записи в выражении. При необходимости изменить порядок выполнения операций в выражениях применяют круглые скобки.

Следует сделать замечание об особенности выполнения операции деления операндов целого типа. *Если в выражении A/B оба операнда целого типа, то значение выражения будет целой частью от деления A на B.* Отметим, что на PASCAL такая операция недопустима.

Арифметические выражения с операндами различных преопределенных типов вычисляются с приведением всех операндов к «старшему» типу. Один арифметический тип считается *старшим* по отношению к другому, если его диапазон представляемых им значений полностью включает в себя диапазон младшего типа. В этом случае преобразование к старшему типу выполняется без потери информации и производится компилятором автоматически. Например, при **`double D; int z; char cc`**; выражение **`D+z-cc`** будет иметь тип **`double`**.

2.13.2 Выражения отношения и логические выражения

Аналогичное приведение к старшему типу производиться в выражениях отношения: сначала операнды приводятся к старшему типу и лишь затем производится сравнение. По логике вещей, тип выражений отношения и логических выражений должен быть логическим с двумя значениями «истина»/«ложь». Но на C++ нет специального предопределенного типа для логических данных.

Логическое значение объекта любого типа интерпретируется как «ложь», если все биты в его битовом представлении уставлены в 0 и, соответственно, «истина», если хотя бы один бит представления находится в состоянии 1.

Это правило действует во всех случаях, когда по контексту требуется логическое «значение» объекта или выражения. Логические выражения и выражения отношения имеют тип **int**, при этом состояние «истина» соответствует значению единица, «ложь» — нуль.

2.13.3 Выражения с поразрядными операциями

Поразрядные операции характерны для программирования на низком уровне, их присутствие в языках Си и C++ дает возможность работать непосредственно с битовыми последовательностями. Это особенно важно при современной тенденции готовить программы для микроконтроллеров на языке высокого уровня.

Рассмотрим примеры выражений с поразрядными операциями. Пусть имеются два объекта беззнакового целого типа, инициализированные некоторым кодом:

```
unsigned a = 0xA7F2, b = 0x70D3;
```

Формирование значений выражений с поразрядными операциями с участием a и b легко понять из приводимых ниже таблиц.

Поразрядное «И»: **a&b**

Выражение	Значение		
	двоичное	шестнадцатеричное	десятичное
a	**1010011111110010**	**A7F2**	**42994**
b	**0111000011010011**	**70D3**	**28883**
a&b	**0010000011010010**	**20D2**	**8402**

Поразрядное «ИЛИ»: **a|b**

Выражение	Значение		
	двоичное	шестнадцатеричное	десятичное
a	**1010011111110010**	**A7F2**	**42994**
b	**0111000011010011**	**70D3**	**28883**
a|b	**1111011111110011**	**F7F3**	**63475**

Поразрядное «ИСКЛЮЧАЮЩЕЕ ИЛИ»: **a^b**

Выражение	Значение		
	двоичное	шестнадцатеричное	десятичное
a	**1010011111110010**	**A7F2**	**42994**
b	**0111000011010011**	**70D3**	**28883**
a^b	**1101011100100001**	**D721**	**55073**

Поразрядное «НЕ»: ~**a** (инверсия кода)

Выражение	Значение		
	двоичное	шестнадцатеричное	десятичное
a	**1010011111110010**	**A7F2**	**42994**
~a	**0101100000001101**	**580D**	**22541**

Как следует из приведенных примеров, действие поразрядных операций сводится к применению логических операций к **i**-му разряду операндов и записи результата в **i**-й разряд результата.

Рассмотрим пример поразрядного сдвига кода беззнакового целого, представленный следующими таблицами.

Поразрядной сдвиг кода вправо на k разрядов **a>>k**

Выражение	Значение		
	двоичное	шестнадцатеричное	десятичное
a>>0	**1010011111110010**	**A7F2**	**42994**
a>>1	**0101001111111001**	**53F9**	**21497**
a>>2	**0010100111111100**	**29FC**	**10748**
a>>3	**0001010011111110**	**14FE**	**5374**
a>>13	**0000000000000101**	**5**	**5**
a>>14	**0000000000000010**	**2**	**2**
a>>15	**0000000000000001**	**1**	**1**

Поразрядной сдвиг кода влево на k разрядов **a<<k**

Выражение	Значение		
	двоичное	шестнадцатеричное	десятичное
a<<0	**1010011111110010**	**A7F2**	**42994**
a<<1	**0100111111100100**	**4FE4**	**20452**
a<<2	**1001111111001000**	**9FC8**	**40904**
a<<3	**0011111110010000**	**3F90**	**16272**
a<<13	**0100000000000000**	**4000**	**16384**
a<<14	**1000000000000000**	**8000**	**32768**
a<<15	**0000000000000000**	**0**	**0**

Сдвиг кода аргумента знакового типа при его отрицательном значении происходит с восстановлением единицы в знаковом разряде. Пусть объявлено:

```
int a = A7F2;
```

Тогда поразрядной сдвиг кода отрицательного числа вправо на k разрядов (значение выражения **a>>k**) будет иметь вид:

Выражение	Значение		
	двоичное	шестнадцатеричное	десятичное
a>>0	**1010011111110010**	**A7F2**	–22542
a>>1	**1101001111111001**	**D3F9**	**–11271**
a>>2	**1110100111111100**	**E9FC**	**–5636**
a>>3	**1111010011111110**	**F4FE**	**–2818**
a>>13	**1111111111111101**	**FFFD**	**–3**
a>>14	**1111111111111110**	**FFFE**	**–2**
a>>15	**1111111111111111**	**FFFF**	**–1**

Применение выражений с поразрядными операциями на C++ позволяет моделировать работу цифровых устройств на уровне регистров.

2.13.4 Выражения присваивания

В Си и C++ есть выражение присваивания, а не только оператор присваивания, как, например в PASCAL. Это «полноправное» выражение, имеющее тип и значение. Тип и значение выражения присваивания совпадает с типом и значением объекта, расположенным в левой части выражения. Например, если **int I** и **float f =3.78**, то выражение **I=f** , будет иметь тип **int** и значение **3**.

Наличие у выражения присваивания значения дает возможность употреблять его в несколько неожиданном контексте; например, **x=1.5*b+(a=3*x)**. Возможна организация цепочек: **A=b=f**, что означает присвоение значения объекта **f** сначала объекту **b**, а затем значение объекта **b** объекту **A**.

Еще одной особенностью операции присваивания C++ является то, что она может быть составной, т.е. совмещаться с боль-

шинством бинарных операций. Например, **`x[i+3]*=4`** означает **`x[i+3]=x[i+3]*4`**, за исключением того факта, что выражение **`x[i+3]`** вычисляется только один раз.

2.13.5 Выражения инкремента и декремента

На C++ существуют унарные операции инкремента (увеличения) и декремента (уменьшения) операндов. Выражения инкремента и декремента могут иметь ***постфиксную*** и ***префиксную*** формы. Эти операции могут применятся как операндам арифметического типа, так указателям. Пример действия операций для арифметических типов приведен в следующих таблицах.

ИНКРЕМЕНТ

Постфиксная форма: **`I++`**			Префиксная форма: **`++I`**		
значение операнда до операции	значение выражения	значение операнда после операции	значение операнда до операции	значение выражения	значение операнда после операции
Тип операнда целый					
10	10	11	10	11	11
Тип операнда плавающий					
0.95	0.95	1.95	0.95	1.95	1.95

ДЕКРЕМЕНТ

Постфиксная форма: **`I--`**			Префиксная форма: **`--I`**		
значение операнда до операции	значение выражения	значение операнда после операции	значение операнда до операции	значение выражения	значение операнда после операции
Тип операнда целый					
10	10	9	10	9	9
Тип операнда плавающий					
0.95	0.95	–0.05	0.95	–0.05	–0.05

Из приведенных примеров следует, что в применении к арифметическим типам операции инкремента и декремента вызывают соответственно увеличение и уменьшение значения операнда на единицу.

Постфиксная и префиксная формы операций отличаются значением самого выражения: в постфиксной форме значение выражения совпадает со значением операнда *до* операции, в префиксной — со значением операнда *после* операции.

С указателями операции инкремента и декремента выполняются в соответствии с правилами адресной арифметики, рассмотренной выше.

2.14 Операторы

Исходный текст любой программы, в том числе и программы на C++, состоит из набора директив — *операторов*, предписывающих компьютеру выполнить какое то ни было действие.

Оператор программы — законченная инструкция на исходном языке. Оператор в программе по логической завершенности аналогичен предложению на естественном языке. После компиляции оператор преобразуется в поток команд, переводящих вычислительную систему из одного состояния в другое. По завершении выполнения оператора внутреннее состояния вычислительной среды (состояние регистров процессора, регистров общего назначения, состояние стека и пр.) возвращается некоторому «стандартному» состоянию. Любой оператор C++ должен заканчиваться разделителем ; — точка с запятой.

Операторы C++ можно разделить на три группы, это

- ***операторы описания;***
- ***операторы выражения;***
- ***операторы управления.***

К группе операторов описания относятся рассмотренные выше директивы объявления и определения объектов программы с инициализацией данных или без таковой.

Операторы — выражения представляют наиболее многочисленную группу операторов. Именно операторы — выражения вы-

зывают преобразование значений переменных, осуществляют вызов функций и осуществляют доступ к объектам.

Операторы управления позволяют разветвлять вычислительный процесс в зависимости от условий, выполнять итерации (циклы) в программе и т.п.

2.14.1 Пустой оператор

На языке C++ возможна запись пустого оператора, т.е. такого оператора, с которым не связаны никакие действия. Синтаксис записи пустого оператора — это просто точка с запятой:

```
;   // пустой оператор
```

Пустой оператор — это синтаксически полноправный оператор, он может использоваться в качестве своего рода заглушки: там, где по соображениям синтаксиса должен находиться оператор, но никаких действий от программы не требуется.

Замечание. Неуместное использование знака ; воспринимаемого компилятором как пустой оператор, может приводить к семантическим ошибкам в программе.

2.14.2 Операторы описания

Оператор описания вводит в программу объект, связывая имя типа с идентификатором объекта. Таким образом, с помощью операторов описания реализуются объявления объектов. На C++ операторы описания имеют не только дескриптивный, но и конструктивный аспект: их действие вызывает создание объектов. Примеры операторов описания даны в п. 2.9.

2.14.3 Операторы-выражения

Самый обычный вид оператора — оператор выражение. Он состоит из выражения, за которым следует точка с запятой. Например:

```
a = b*3+c; // оператор присваивания с арифметическим выражением
lseek(fd,0,2); // оператор вызова функции
```

2.14.4 Составные операторы (блоки)

Составной оператор или блок — это последовательность операторов, заключенных в фигурные скобки:

```
{          // начало блока
a=b+2;
. . .      // еще операторы
b++;
}          // конец блока
```

Синтаксически блок рассматривается как один оператор.

Если в составе операторов блока встречаются операторы описания, то вводимые ими объекты локальны и имеют *область видимости* (см. ниже) только в пределах данного блока. В случае, если имя локальной переменной совпадает с именем переменной, описанной в охватывающем блоке, то происходит т.н. *маскировка имени* — внутри блока имя будет ссылаться *только* на локальную переменную. Рассмотрим пример.

Пусть имеется три вложенных друг в друга блока и в каждом из них объявлена переменная **`I`**:

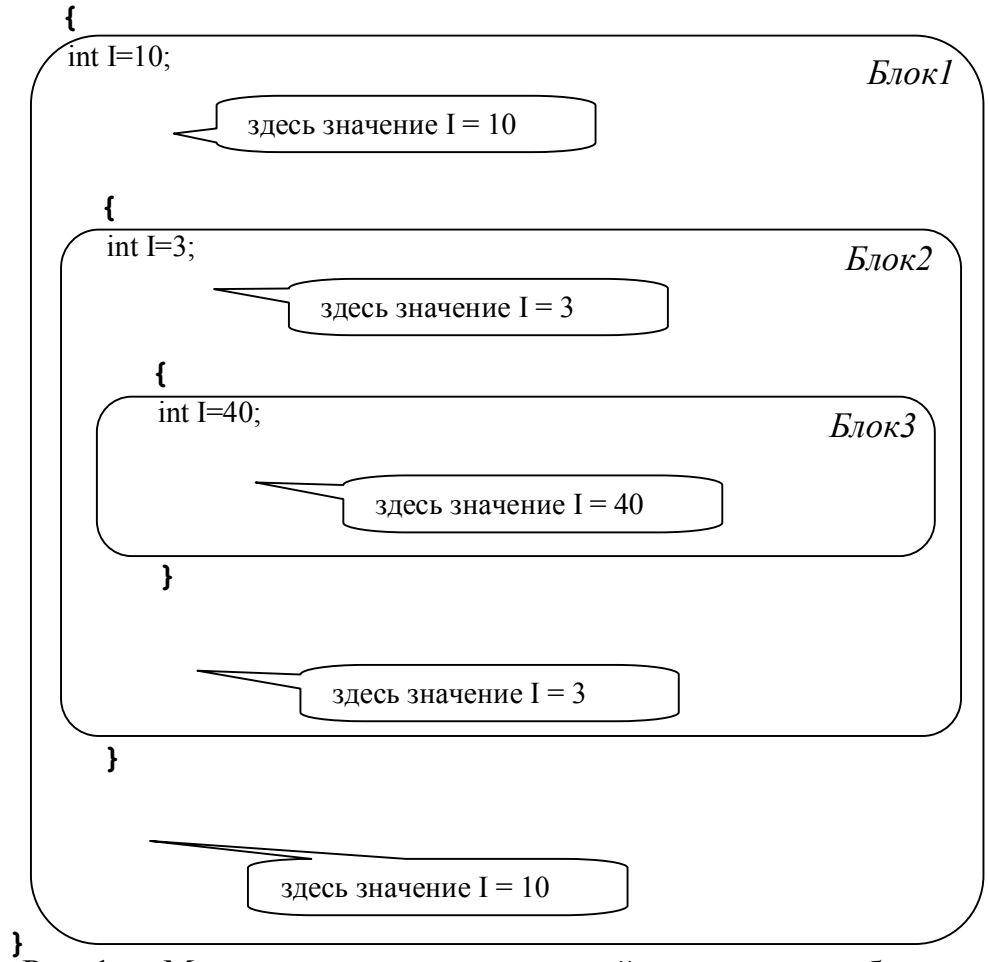


Рис. 1 — Маскировка имени переменной во вложенных блоках

Таким образом, операторы описания, содержащиеся внутри блока, создают локальный объект внутри этого блока. В случае, если имя локального объекта совпадает с именем объекта во внешнем охватывающем блоке, то имя локального объекта маскирует имя внешнего объекта. Образно говоря, на имя внутри блока «откликается» ближайший объект. Более подробно явления маскировки будут обсуждаться в разделе «Области видимости и время жизни объектов».

2.15 Операторы управления

2.15.1 Операторы if и if/else

Оператор **`if`** предназначен для реализации в программе следующей структуры:

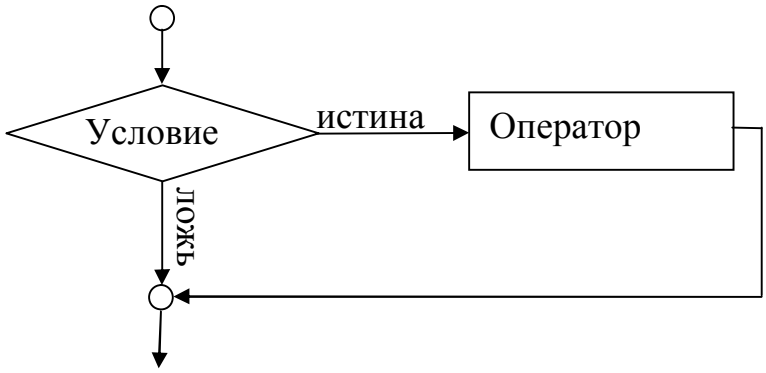


Рис. 2 — Логическая структура оператора **`if`**

Оператор исполняется, если условие истинно, иначе он пропускается. Синтаксис записи условного оператора:

`if (`*логическое выражение***`)`** *исполняемый оператор***`;`** …

Например:

```
if(a>5) a=5;

if(t) t=1/t;  // если t ≠ 0 значение t меняется на обратное
```

В последнем примере использована логическая интерпретация значения переменной арифметического типа.

Условный оператор состоит из заголовка, образованного зарезервированным словом **if** и последующим условным выражением в круглых скобках. За заголовком следует исполняемый оператор.

Исполняемый оператор может быть блоком, содержащем сколь угодно много операторов. При истинности условия будет выполняться весь этот блок.

Внимание! Не ставьте после заголовка оператора точку с запятой, это приведет к семантической ошибке: при истинности условия будет «выполняться» пустой оператор.

Условный оператор имеет альтернативную форму **if/else**, реализующую следующую алгоритмическую структуру:

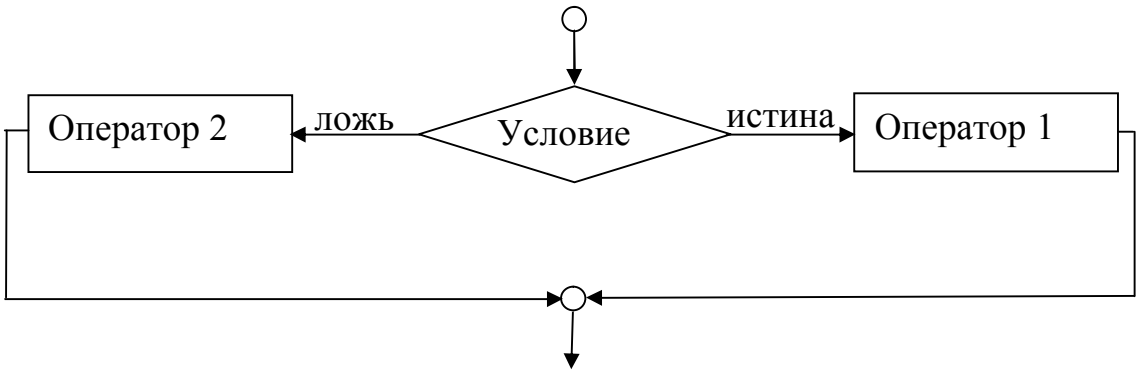


Рис. 3 — Логическая структура оператора **if \ else**

В отличие от оператора **if**, рассмотренного выше, оператор **if \else** производит альтернативный выбор: если условие истинно, выполняются действия оператора 1, иначе — действия оператора 2.

Синтаксис записи оператора на языке С++:

if(*логическое выражение*) *оператор1*; **else** *оператор2*;

Пример использования:

```
if(f<0) c=-1;  else c=1;
if(g) h='1'; else h='0';
```

Здесь также, как и в операторе **if** исполняемые операторы 1 и 2 могут быть составными и представлять собой блоки.

Условные операторы могут быть вложенными друг в друга и глубина вложения теоретически не ограничена. Это обстоятельство создает определенные неудобства при чтении текста программы с многократным вложением альтернативных операторов: последовательность перемежающихся **`if`** и **`else`** затеняет структурность текста, так что становиться трудно определить к какому же **`if`** относиться тот или иной **`else`**. Во избежание подобной путаницы не следует употреблять глубоко вложенные конструкции альтернативных операторов.

2.15.2 Операторы `switch`

Оператор многоальтернативного выбора **`switch`** предназначен для реализации в программе структуры ветвления следующего вида:

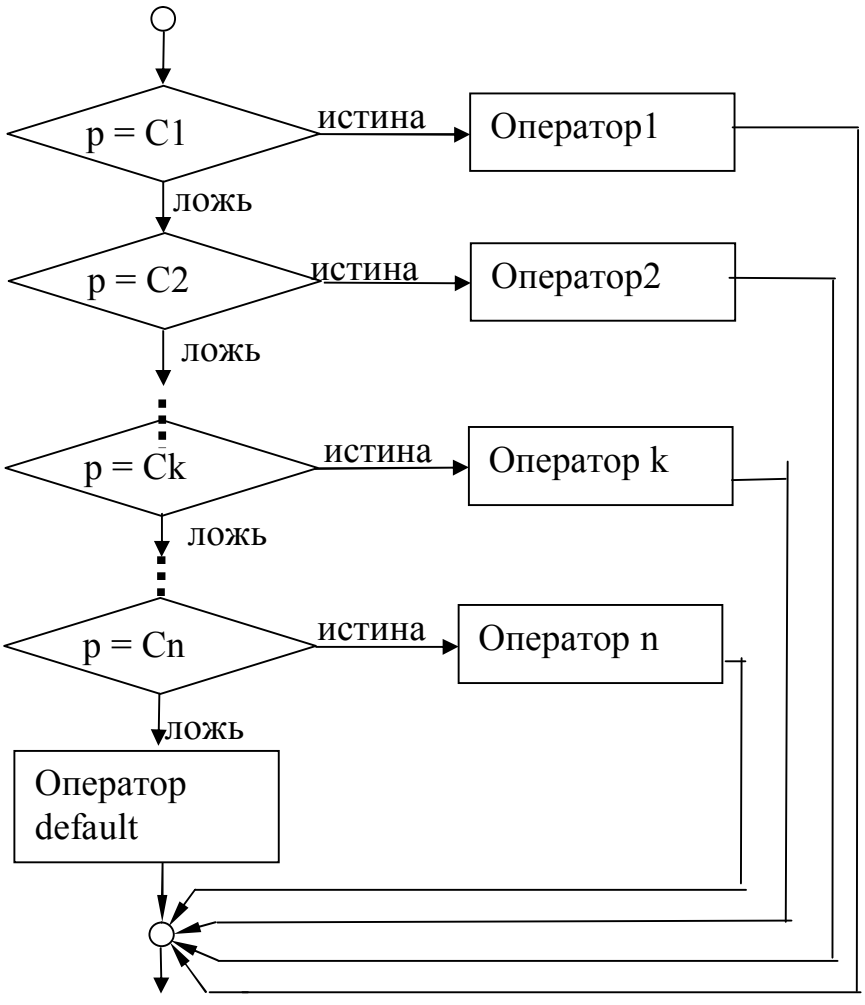


Рис. 4 — Логическая структура оператора **`switsh`**

Здесь **p** — параметр оператора, значение которого сравнивается множеством констант **C1, C2, ..Cn**. При равенстве значения **p** какой-либо из констант вычислительный процесс направляется на выполнение соответствующего оператора (блока операторов). В случае, когда значение p оказывается отличным от всего набора констант исполняется оператор по умолчанию (оператор default)

```
switch (выражение для значения параметра)
{
case C1: {оператор 1; break;}
case C2: {оператор 2; break;}
    ...
case Ck: {оператор k; break;}
    ...
case Cn: {оператор n; break;}
default :оператор default;

}
```

Тело оператора **switch** представляет собой блок, охватывающий все альтернативные ветви вычислительного процесса. Каждая из альтернативных ветвей тоже представляет собой блок, заканчивающийся оператором **break.**

Операторы **break** применяются для выхода из оператора **switch**. Дело в том, **case Ck** синтаксически представляют собой метки. При отсутствии оператора **break** вычислительный процесс, попав, например, на метку **case Ck**, пойдет далее по порядку следования операторов, невзирая на встречающиеся метки, и после выполнения оператора **k** произойдет выполнение всех операторов, расположенных ниже.

Константы в вариантах **case** должны быть различными, и если проверяемое значение не совпадает ни с одной из констант, выбирается вариант **default**. Можно не предусматривать вариант **default**, в этом случае вычислительный процесс покинет пределы оператора **switch** , ничего не исполнив.

Пример использования оператора **switch:**

```
switch (i)
{
case 0: {cout<<"тишина"; break;}
case 1: {cout<<"соло"; break;}
case 2: {cout<<"дуэт"; break;}
case 3: {cout<<"трио"; break;}
case 4: {cout<<"квартет"; break;}
default :cout<< "ни то, ни се";
}
```

Обратим внимание, что в ветви последней альтернативы оператор **break** можно не ставить т.к. после ее выполнения вычислительный процесс сам выйдет за пределы оператора **switch.**

2.15.3 Операторы while и do while

Операторы итерации **while** и **do while** предназначены для создания в программе циклических структур следующего вида:

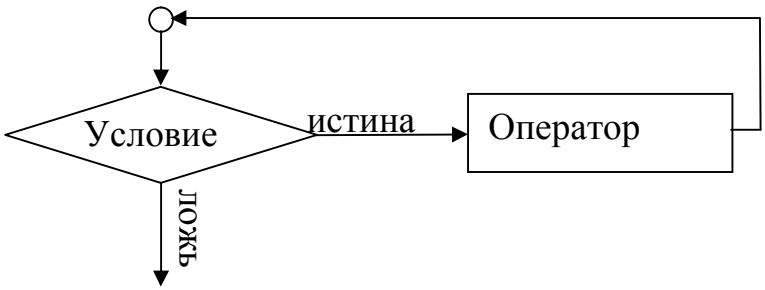


Рис. 5 — Структура **while**

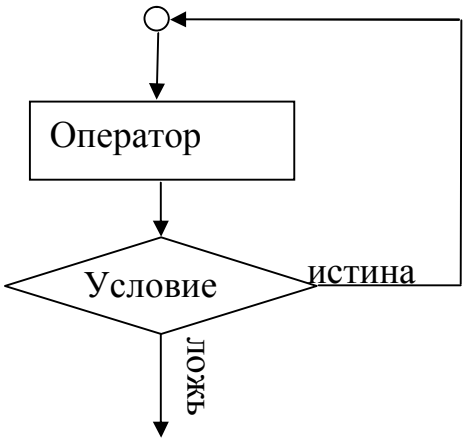


Рис. 6 — Структура **do while**

Циклические структуры **while** и **do while** обе реализуют циклы типа «*пока*». Вычислительный процесс циклит, пока условие истинно. Различие между ними лишь в точке проверки условия продолжения цикла: в структуре **while** это условие проверяется *до* исполнения оператора в теле цикла, в структуре **do while** — *после* исполнения этого оператора. Поэтому структуру **while** называют циклом *с предусловие*м, а структуру **do while** — циклом с *постусловием*.

Синтаксис записи оператора **while** :

while(*условное выражение*) *оператор*;

Синтаксис записи оператора **do while:**

do *оператор*; **while (***условное выражение***);**

В операторе **do while** зарезервированное слово **do** ставиться перед оператором, образующим тело цикла. Между **do** и **while** может располагаться только один оператор, если требуется поместить в тело цикла более одного оператора, то их следует объединить в блок.

Примеры записи операторов:

```
while(c=fgetch(F)!=EOF)C[i++]=c; // чтение в массив С
                                 // символов из файлового потока,
                                 // пока не будет достигнут конец файла

do {c='A'; cout<<c++;} while(c<256);// вывод
                                 // последовательности символов ASCII,
                                 // начиная с кода 'A' и выше.
```

Оператор выборки символов из файла из первого примера можно записать короче, исключив оператор тела цикла и заменив его пустым оператором:

```
while((C[i++]=fgetch(F))!=EOF);
```

При этом отпадает необходимость в дополнительной буферной переменной **c,** но при такой форме записи в последний элемент массива попадет символ конца файла **EOF**.

2.15.4 Оператор *for*

Оператор **for** предназначен для создания в программе следующей циклической структуры:

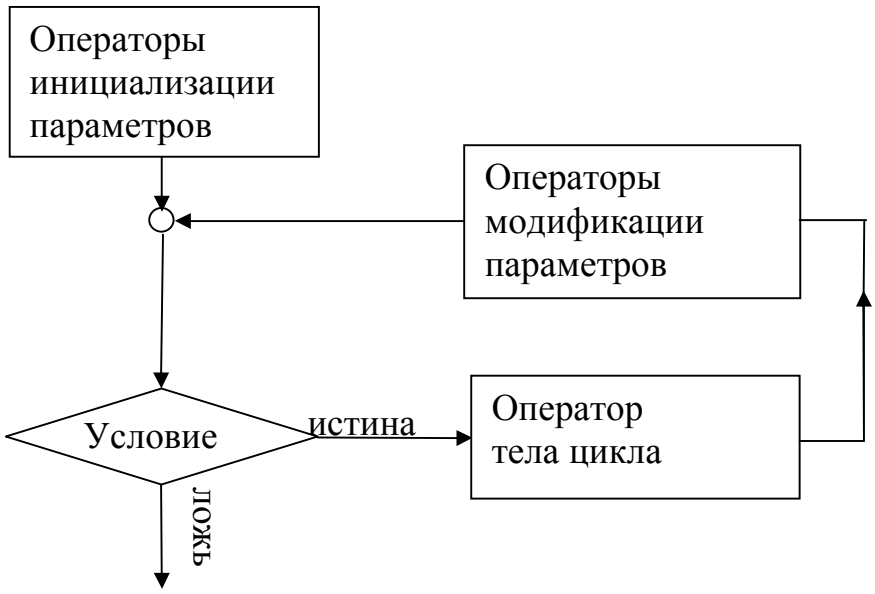


Рис. 7 — Логическая структура оператора **for**

Вычислительный процесс, соответствующий этой структуре в информационном плане может быть более насыщен, чем просто цикл со счетчиком. В его составе могут быть не только одиночные операторы инициализации счетчика, но и группы операторов инициализации для набора параметров. Такие же группы операторов могут изменять значения параметров после каждого выполнения тела цикла.

Синтаксис оператора **for**:

for(*список операторов инициализации***;** *условие продолжения цикла*; *список операторов модификации***)** *оператор тела цикла*;

Списки операторов инициализации и модификации — это перечисление их выражений через запятую.

Примеры:

```
for(i=0; i<10; i++) D[i]=i; // простейший цикл
                            // со счетчиком итераций
```

```
for(i=0,a=0.17; i<10; i++,a+=0.5) D[i]=a;// цикл с
                                  //двумя выражениями в инициализирующей
                                  // и модифицирующей частях. Переменные i
                                  // и a должны быть объявлены ранее.

for(int i=0,k=-17; i<10; i++,k+=5) D[i]=a;// цикл с
                                  //объявлением и инициализацией двух
                                  // переменных в инициализирующей
                                  // части оператора for.
```

В последнем примере в заголовке оператора **for** введены две переменные **i** и **k**. В версии Borland C++ 3.1 область видимости этих переменных распространяется до конца внешнего блока, охватывающего оператор **for**. В более старших версиях C++ область видимости переменных, введенных в инициализирующей части оператора **for** *ограничена заголовком и телом цикла*.

На основе структуры **for** можно получать многообразные программные реализации. В литературе даже высказывается мнение, что любую программу можно представить в виде логической структуры цикла **for**. На самом деле, эта логическая конструкция имеет все необходимые компоненты типичной моделирующей программы — блок формирования начальных значений, блок проверки факта достижения цели и блок смены условий моделирования перед повторным просчетом модели.

В ряде ситуаций для реализации поставленной цели может оказаться достаточным лишь тех операций, что содержатся в заголовке цикла, тогда отпадает необходимость в операторе тела цикла, и на его место можно поместить пустой оператор. Например, заполнение 10 элементов массива **s** начальным фрагментом геометрической прогрессии $s0$, $s0 \cdot q$, $s0 \cdot q^2$, … можно выполнить с помощью следующего оператора **for**:

```
for (s[0]=s0, i = 1; i<10; s[i]=s[i-1]*=q,i++);
```

В этом примере оператор тела цикла — пустой, т.к. операторы в заголовке цикла решают в полной мере решают поставленную задачу, так что в теле цикла уже ничего делать не нужно.

2.15.5 Оператор `break`

Этот оператор предназначен для использования в структурах повторения (циклах), и в структуре многоальтернативного выбора, реализуемой оператором **`switch`**. В том и другом случае назначение оператора **`break`** одинаковое — немедленное завершение выполнения оператора, в теле которого он расположен.

Действие **`break`** в операторе **`switch`** было рассмотрено выше. В циклах оператор **`break`** обычно используется совместно с условным оператором if и служит для прерывания цикла при истинности некоторого условного выражения. Оператор **`break`** удобен для прерывания по определенному условию т.н. «вечных» циклов специально зацикливающих программу в ожидании наступления какого-то внешнего события. Например:

```
while(1)
{ int c = getch();
 if(c==27) break;
cprintf("%c",c);
}
```

Этот фрагмент программы будет постоянно запрашивать ввода символа с клавиатуры и выводить полученный символ на экран, до тех пор, пока не будет введен символ с кодом 27 (код клавиши Esc), после чего цикл прервется.

2.15.6 Оператор `continue`

Оператор **`continue`** используется в циклах для возврата к началу цикла, минуя оставшиеся в теле цикла операторы. Пример:

```
for(int i=0; i<n-1; i++)
{ if(M[i]<M[i+1]) continue;
int T=M[i];
M[i]=M[i+1];
```

```
M[i+1]=T;

}
```

Этот фрагмент программы просматривает массив из n элементов и если предыдущий элемент окажется больше последующего меняет их местами. Исключение трех операторов, осуществляющих обмен, выполнено с использованием оператора **continue**.

Оператор **continue** применяется редко, т.к. можно легко заменить другими операторами, причем программа будет более структурной. Так, в приведенном примере вполне можно было обойтись и без оператора **continue**:

```
for(int i=0; i<n-1; i++)

{ if(M[i]>M[i+1])

  { int T=M[i];

   M[i]=M[i+1];

   M[i+1]=T;}

  }

}
```

Этот фрагмент кода делает то же самое, но выглядит более структурировано.

2.15.7 Оператор `return`

Оператор **return** предназначен для завершения выполнения функции и возврата в вызывающую программу. Оператор является обязательным, если функция объявлена с возвращаемым типом, отличным от **void.** В этом случае оператор **return** записывается так:

return <*выражение*>;

где < *выражение* > должно иметь тот же тип, что и тип возвращаемого значения в объявлении функции. Значение этого выра-

жения и будет возвращено в вызывающую программу в качестве результата работы функции.

Функция может содержать несколько операторов **return** , находящихся на различных ветвях ее вычислительного процесса, но в любом случае это последний исполняемый оператор в теле функции. Не будет ошибкой, если оператор **return** поместить в функцию с объявленным типом возвращаемого значения **void**. И в этом случае оператор **return** выполнит свою задачу по завершению работы функции, но при этом следует записывать этот оператор без возвращаемого значения:

```
return;
```

Поскольку на Си и C++ головная программа — тоже функция, то использовав в ней оператор **return** в соответствующей форме, можно завершить выполнение всей программы.

2.15.8 Оператор goto

Этот оператор выполняет безусловную передачу управления на метку. Синтаксис его записи:

```
goto AAA;
```

где AAA — имя метки.

Оператор **goto** относится к виду неструктурных операторов, он достался в наследство от прежних технологий программирования и в современной практике практически не применяется.

Дело в том, что с помощью этого оператора легко нарушается структурность программы: можно запросто войти внутрь цикла, минуя его заголовок, можно передать управление внутрь блока условного оператора без анализа истинности соответствующего ему условия и т.п. Даже правильно написанная программа с операторами **goto** чрезвычайно трудно читается и анализируется.

Итак, существует достаточно веских причин не использовать этот оператор, но поскольку он присутствует в синтаксисе языка, рассмотрим его работу на примере. Пусть имеется фрагмент программы:

```
    int i;
beg : i = getch();
      if(i==27) goto fin;
      cprintf("%c",i);
    goto beg;
    fin : <продолжение программы>
```

Этот фрагмент программы выполняет тот же бесконечный цикл ввода символов с клавиатуры, прерываемый нажатием клавиши Esc, что и в примере с бесконечным циклом **while**, но реализованный другим способом. Здесь использованы две метки с именами `beg` и `fin`. Первая метка отмечает точку начала цикла, вторая — первый исполняемый оператор за циклом, ту точку, куда передается управление при прерывании цикла.

Метки на C++ это особые лексемы, состоящие, подобно идентификаторам, из последовательности букв и цифр. Значение метки — это ее имя. Метка отделяется от помечаемого оператора двоеточием. Наличие метки у оператора никак не сказывается на его выполнении.

Несмотря на кажущееся удобство в приведенном простом примере, использование операторов **goto** и меток в сколько-нибудь сложных программах нежелательно, лучше воспользоваться структурными операторами.

3 ФУНКЦИИ

Языки Си и C++ иногда называют языками функций — столь велика в них их роль. Функции позволяют полной мере реализовать модульную технологию программирования. Модули — подпрограммы обычно выделяют по принципу функциональной декомпозиции, и каждый из них решает свою логически законченную задачу. Инструментом создания обособленных модулей на Си и C++ служат только функции, в отличие от PASCAL процедур здесь нет. В программах составленных на этих языках отсутствует и синтаксически выделенная головная программа, ее роль также играет функция, единственной особенностью которой является зарезервированное за ней имя **`main`**.

Ни на Си, ни на C++ нет специальных операторов ввода \ вывода, подобных паскалевским WRITE и READ, операции обмена осуществляются с помощью функций.

Более или менее сложную программу целесообразно разбить на модули, которые можно редактировать и отлаживать независимо друг от друга, так что лучший способ разработки и сопровождения большой программы — разделение ее на несколько модулей, каждый из которых более управляем, чем исходная программа. Модули пишутся на C++ в виде функций и классов. Классы мы рассмотрим позже, а сейчас займемся функциями.

3.1 Структура заголовка функции

Функция — это именованная подпрограмма, к которой можно обращаться из других частей программы столько раз, сколько потребуется. Синтаксически функция C++ представляет собой обособленный именованный блок с *интерфейсом*, задающим формат обмена информацией с вызывающей программой. Передача информации, минуя интерфейс, например, посредством «глобальных» переменных, хотя и допустима, но крайне нежелательна т.к. нарушает автономность подпрограммы.

Цель скрытия информации в функциях заключается в том, чтобы дать доступ только к той информации, которая нужна для выполнения их задач. Это средство реализации принципа наи-

меньших привилегий, одного из наиболее важных принципов разработки хорошего программного обеспечения.

Описание интерфейса содержится в заголовке функции, который имеет следующую структуру:

тип_возвращаемого_значения Имя_функции (список_параметров)

Результат работы функции обычно возвращается в вызывающую ее программу в виде значения, которое ассоциируется с именем функции точно так же, как значение переменной ассоциируется с ее именем — идентификатором. Какой тип значения будет возвращать функция, указывает первый по порядку элемент заголовка — *тип_возвращаемого_значения.*

Имя функции — обычный идентификатор **C**++, и назначение имени функции схоже с назначением идентификатора переменной — с ним связывается адрес объекта, в данном случае это адрес точки входа в подпрограмму.

Список параметров — это описание информационного входа функции. В списке через запятую указываются типы аргументов и их имена, задается количество аргументов и порядок их следования при вызове функции.

3.2 Логический механизм вызова функций

Код программы, вызывающей функцию, и код ее реализующий обычно разнесены в адресном пространстве. Поэтому при вызове функции производится скачок по адресному пространству — передача управления на точку входа в подпрограмму. Тот адрес, куда надо «прыгнуть» связывается с именем функции. До совершения скачка требуется куда-то временно поместить значения входных параметров с тем, чтобы их использовать в теле подпрограммы.

Компиляторы C++ строят исполняемую программу так, что в качестве временного хранилища используется программный стек. Перед передачей управления стек загружается значениями параметров в строгом соответствии с той последовательностью и типами, что указаны с списке параметров в заголовке функции.

.

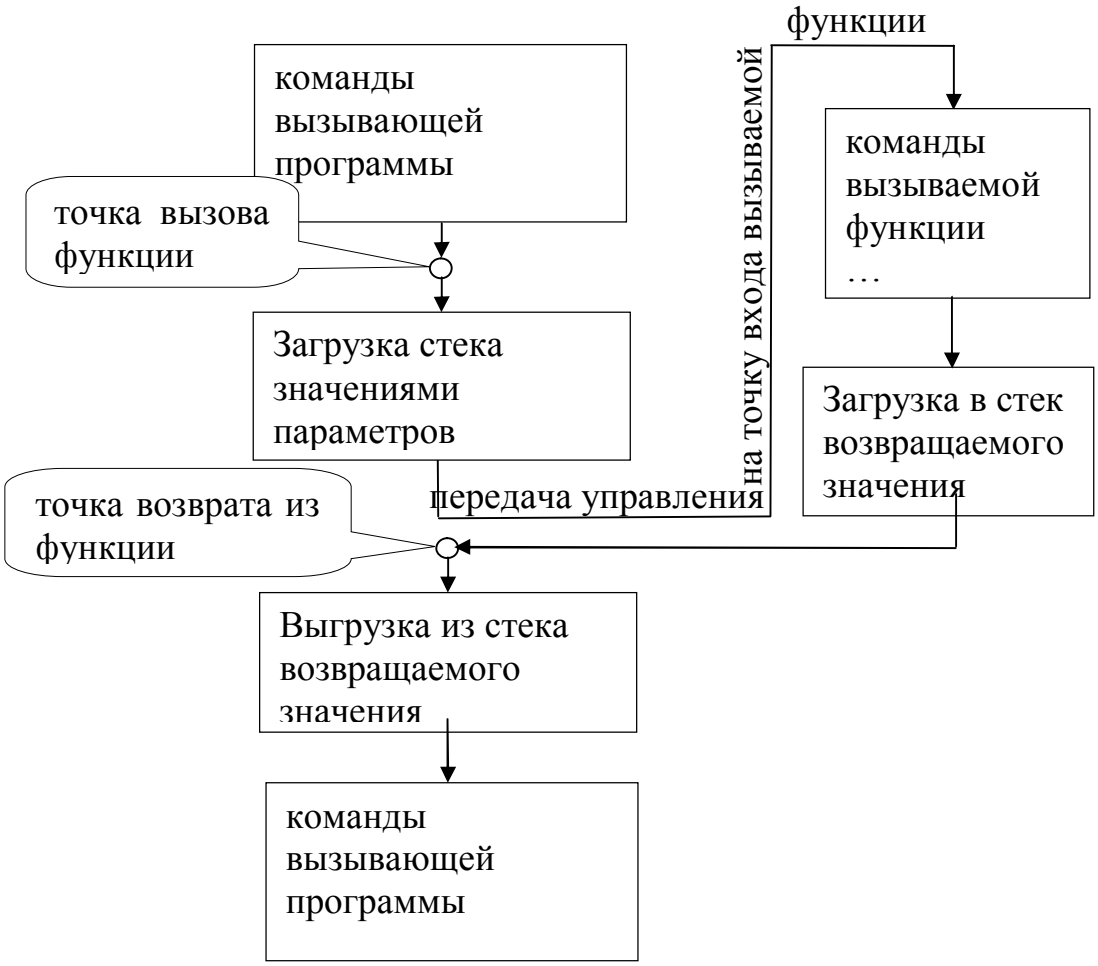


Рис. 8 — Схема вызова функции

После завершения работы подпрограммы в освободившийся к тому времени стек загружается возвращаемое значение тоже в соответствии с типом, указанным в заголовке функции, и совершается обратный скачок в вызывающую программу.

В вызывающей программе происходит выгрузка из стека возвращенного значения (в соответствии объявленным типом!) и вычислительный процесс продолжается. Цепочка действий при вызове функции показана на рисунке.

Если функция выполняет набор каких-либо простых действий, ее исполняемый код имеет небольшой размер, не содержит структур ветвления и итераций, то «накладные расходы», связанные с проходом вычислительного процесса по петле, становятся соизмеримыми с вычислительными затратами на исполнение самого тела функции, а, зачастую, даже и превосходить их. В этом случае целесообразно не вызывать функцию по обычной схеме, а встроить ее исполняемый код в вызывающую программу прямо

за точкой вызова. Функции, вызов которых организован по такому принципу, называют встраиваемыми или инлайновыми (*inline*) функциями.

Использование *inline* — функций экономит время вызова, жертвуя при этом объемом памяти, т.к. в каждой точке вызова требуется разместить исполняемый код вызываемой функции.

Компилятор предупреждается о том, что функцию следует вызывать как инлайновую зарезервированным словом **inline,** записываемым перед заголовком функции, например:

```
inline int GetX();

inline double SQR(double x){return x*x ;}
```

Не всякую функцию компилятор может реализовать как inline. Если в теле функции есть операторы ветвления и циклы, то у компилятора «не хватает интеллекта» построить ее встраиваемый код. Поэтому директива **inline** перед заголовком функции носит рекомендательный характер, в случае, если построение функции, помеченной программистом как **inline** по встраиваемой схеме невозможно, компилятор строит ее вызов по обычной схеме, выдавая пользователю соответствующее предупреждение.

3.3 Объявление, определение и вызов функции

С функцией на языке С++ связано три понятия:
объявление функции;
определение функции;
вызов функции.

3.3.1 Объявление функции

Функции могут быть самостоятельными единицами трансляции, т.е. способны компилироваться независимо друг от друга. Такая независимость потребовала дополнительных мер по информированию компилятора о структуре интерфейсов вызываемых функций. Роль такого «информатора» отведена *объявлению (прототипу)* функции. Объявление функции должно предшествовать ее первому вызову.

Объявление (или прототип) функции на C++ выглядит следующим образом:

тип_возвращаемого_значения Имя_функции
(список_типов_параметров);

Таким образом, прототип функции — это ее заголовок, заканчивающийся точкой с запятой. Отметим, что в прототипе можно не указывать имена параметров, достаточно указать лишь их типы.

Объявления, представляющего собой описание интерфейса функции, достаточно для согласования формата обмена информацией с вызывающей программой. В самом деле, список типов входных параметров указывает, чем должен загружаться стек при вызове данной функции, и в какой последовательности это следует делать. Тип возвращаемого значения предписывает, как следует интерпретировать содержимое стека после возврата из функции.

Примеры объявлений функций:

```
int IJmaxR(int,int,double*,int*,int*);

int G_MWel(int,double*,double&,double*);

void FF (int);

int RR().
```

В приведенных примерах функции с именами **IJmaxR** и **G_MWel,** возвращают целочисленные значения. Функция **IJmaxR** принимает при вызове пять аргументов в соответствии с указанной последовательностью типов, функция **G_MWel** — четыре.

Указание специального типа **void** в качестве типа возвращаемого значения — это предупреждение компилятору о том, что по завершении работы функция не будет возвращать результат через свое имя, и выгружать что-либо из стека по ее окончании не следует. Функции, объявленные с возвращаемым типом **void** – это аналоги паскалевских процедур.

Таким образом, функция **FF** не возвращает ничего, принимает один параметр типа **int**. Функция **RR** объявлена с пустым списком параметров, это означает, что она не принимает никаких параметров.

На С++ имеется возможность задавать значения аргументов функции по умолчанию, для этого в списке параметров нужно указать какие из параметров могут быть заданы по умолчанию и какое при этом им присвоится значение. Синтаксис записи списка параметров при этом таков:

(тип_1 имя1, тип2 имя2, тип3 имя3= значение_по_умолчанию)

Здесь показан вид списка параметров функции, принимающей три параметра. Последний по порядку списка параметр может быть инициализирован значением по умолчанию, указанным справа от знака = .

Механизм присвоения по умолчанию запускается, если аргумент по умолчанию пропускается при вызове функции. Только в этом случае используется его значение по умолчанию.

Аргументы по умолчанию должны быть крайними правыми (последними) в списке параметров функции. Аргументы по умолчанию должны быть указаны при первом же упоминании имени функции. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций.

3.3.2 Определение функций

В общем виде определение функции на С++ выглядит следующим образом:

тип_возвращаемого_значения имя_функции (список_ параметров)

{

тело функции: директивы программы

}

Примеры определений функции:

```
int power(int a, int b)
{ int s=1;
  for(int i=0; i<b; i++)
```

```
  s*=a;
  return s;
}

double SUMMA(int n, double* D)
{ double s=0;
for(int i;i<n; i++)s+=D[i];
return s;
}
```

Функция первого примера возводит значение целочисленной переменной, задаваемой в качестве первого аргумента, в целочисленную степень, определяемую значением второго аргумента.

Функция второго примера вычисляет сумму **n** значений элементов массива **D**, состоящего из компонентов типа **double**. При вызове функции величина **n**, передается первым параметром типа **int**, массив передается вторым параметром как указатель на **double***.

В определении наряду с заголовком функции, определяющим ее интерфейс, содержится и текст ее программы. Как уже упоминалось выше, функция — это именованный блок. Это блок — составной оператор, со всеми присущими ему свойствами.

Локальные переменные, введенные в описании данной функции, известны только ей, прочие функции, включая функцию **main**, этих переменных не видят и, следовательно, не могут их использовать.

Блок, содержащий тело функции не может быть помещен в какой-либо иной охватывающий блок, т.е. нельзя определить одну функцию «внутри» другой, *функции не могут быть вложенными друг в друга*.

Роль охватывающего блока для функций играет область файла, где помещены их описания. В пределах этой области, вне всяких блоков, включая функции, могут находиться операторы описания объектов. Эти т.н. *глобальные* объекты могут быть видимы всеми функциями. В принципе, обмен информацией между функциями можно организовать в помощью глобальных переменных, но как уже отмечалось этот путь нарушает принцип инкапсуляции данных и чреват негативными последствиями.

В функциях, точно так же как в блоках, может происходить маскировка имен, когда имя локальной переменной совпадает с

именем глобальной переменной. В этом случае до глобальной переменной можно добраться, используя унарную операцию разрешения области действия (::), которая позволяет обеспечить доступ к глобальной переменной в случае, когда локальная переменная имеет в области действия такое же имя.

3.3.3 Вызов функций

Активизация программы функции происходит после исполнения оператора ее вызова. Оператор вызова функции — это оператор выражение. Собственно выражение вызова состоит из имени функции и следующего за ним списка аргументов в круглых скобках. Пример операторов вызова функций:

```
sqrt(x);

clrscr();

fopen("c:\\mayfile.txt","wb");
```

При вызове функции на место формальных параметров становятся фактические. Передача фактических параметров идет в строгом соответствии с порядком их следования в списке формальных параметров. При этом компилятор проверяет соответствие типов данных и их количества, тому, что было указано в списке при объявлении функции.

Пример фрагмента программы с операторами вызова функций:

```
int N=6,nn=3,II[6],JJ[6], rI, rG;// объявления данных

double A[42], detA, Xr[6];       // в вызывающей программе

//. . .

rI = IJmaxR( N, nn, A,II,JJ); // вызов функции IJmaxR

rG = G_MWel(N, nn, detA,Xr); // вызов функции G_MWel

//. . .
```

В приведенном фрагменте вызывающей программы наряду с операторами вызова функций показаны директивы объявления данных, используемых в качестве фактических параметров.

Замечание: Имена формальных параметров в заголовке определения функции никак не связаны с именами фактических параметров в операторе ее вызова.

3.3.4 Передача параметров функциям

При вызове функции в стеке выделяется память для ее формальных параметров, и каждый формальный параметр инициализируется значением соответствующего фактического параметра. Фактически, в любом случае значение фактического параметра копируется в стек, и далее функция работает только с этой копией.

Существуют три способа передачи информации внутрь функции:

1. передача параметра *по значению*;
2. передача параметра *по указателю*;
3. передача параметра *по ссылке*;

Семантика передачи параметров тождественна семантике инициализации. В частности, сверяются типы формального и соответствующего ему фактического параметра, и выполняются, если необходимо, стандартные и пользовательские преобразования типа.

Рассмотрим функцию:

```
void f(int val, int* poin, int& ref)
{val++;
ref++;
*poin = 7;
}
```

При вызове f() в выражении val++ увеличивается локальная копия первого фактического параметра, тогда как в ref++ увеличивается сам третий фактический параметр. В результате операции разыменования указателя poin произойдет изменение значения того объекта, адрес которого находится в poin. Поэтому в функции

```
void g()
{int i = 1;
 int j = 1;
 int k = 1;
f(i,&j,k);
}
```

увеличатся значения j и k, а i останется неизменным. Первый параметр i передается по значению, второй параметр j передается через указатель и третий параметр k — по ссылке. В качестве второго фактического параметра при вызове f() указан адрес переменной k, определенной в вызывающей функции g().

Функции, которые изменяют свой передаваемый по ссылке параметр, труднее понять, и что поэтому лучше их избегать. Но большие объекты, очевидно, гораздо эффективнее передавать по ссылке, чем по значению. Правда можно описать параметр со спецификацией **const**, чтобы гарантировать, что передача по ссылке используется только для эффективности, и вызываемая функция не может изменить значение объекта:

```
void f(const large& arg)
{
// значение "arg" нельзя изменить без явных операций
// преобразования типа
  }
```

Если в описании параметра ссылки **const** не указано, то это рассматривается как намерение изменять передаваемый объект:

```
void g(large& arg); // считается, что в g() arg будет меняться
```

Отсюда вывод: необходимо использовать **const** всюду, где необходимо гарантированно защитить фактический параметр от воздействия вызываемой функции.

Точно так же, описание параметра, являющегося указателем, со спецификацией **const** говорит о том, что указываемый объект не будет изменяться в вызываемой функции. Например:

```
int strlen(const char*);
```

Значение такого приема растет вместе с ростом программы.

Отметим, что семантика передачи параметров отличается от семантики присваивания. Это различие существенно для параметров, являющихся const или ссылкой, а также для параметров с типом, определенным пользователем.

Литерал, константу и параметр, требующий преобразования типа, можно передавать как параметр типа const&, но без спецификации const передавать нельзя. Допуская преобразования для параметра типа const T&, мы гарантируем, что он может принимать значения из того же множества, что и параметр типа T, значение которого передается при необходимости с помощью временной переменной.

```
float fsqrt(const float&);

  void g(double d)
  {
    float r;
    r = fsqrt(2.0);  // передача ссылки на временную
                     // переменную, содержащую 2.0
    r = fsqrt(r);     // передача ссылки на r
    r = fsqrt(d);     // передача ссылки на временную
                  // переменную, содержащую float(d)
  }
```

Запрет на преобразования типа для параметров-ссылок без спецификации **const** введен для того, чтобы избежать нелепых ошибок, связанных с использованием при передаче параметров временных переменных:

```
void update(float& i);

void g(double d)
{
  float r;

  update(2.0);    // ошибка: параметр-константа
  update(r);      // нормально: передается ссылка на r
  update(d);      // ошибка: здесь нужно преобразовывать тип

}
```

Возвращаемое значение

Если функция не описана как void, она должна возвращать значение. Возвращаемое значение указывается в операторе return в теле функции.

Например:

```
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
```

В теле функции может быть несколько операторов return:

```
int fac(int n)
{
if (n > 1) return n*fac(n-1);
else  return 1;
}
```

Подобно передаче параметров, операция возвращения значения функции эквивалентна инициализации. Считается, что оператор **`return`** инициализирует переменную, имеющую тип возвращаемого значения.

Тип выражения в операторе **`return`** сверяется с типом функции, и производятся все стандартные и пользовательские преобразования типа. Например:

```
double f()
{
 // ...
 return 1;   // неявно преобразуется в double(1)
}
```

При каждом вызове функции создается новая копия ее формальных параметров и внутренних локальных переменных. Занятая ими память после выхода из функции будет снова использоваться, поэтому неразумно возвращать указатель на локальную переменную. Содержимое памяти, на которую настроен такой указатель, может измениться непредсказуемым образом:

```
int* f()
{    int local = 1;
  return &local;    // ошибка
}
```

Эта ошибка не столь типична, как сходная ошибка, когда тип функции — ссылка:

```
int& f()
{
int local = 1;
return local;   // ошибка
}
```

К счастью, компилятор предупреждает о том, что возвращается ссылка на локальную переменную.

Вот другой пример:

```
int& f() { return 1; }  // ошибка
```

Передача параметров — массивов

Если в качестве параметра функции указан массив, то передается указатель на его первый элемент. Например:

```
int strlen(const char*);
void f()
{
char v[] = "массив";
strlen(v);
strlen("Николай");
}
```

Это означает, что фактический параметр типа T[] сначала преобразуется к типу T*, и затем передается. Поэтому присваивание элементу формального параметра-массива какого-либо значения внутри тела функции изменяет этот элемент. Иными словами, *массивы отличаются от других типов тем, что они не передаются и не могут передаваться по значению*.

В вызываемой функции размер передаваемого массива неизвестен. Это неприятно, но есть несколько способов обойти данную трудность. Прежде всего, все строки оканчиваются нулевым символом, и, значит, их размер легко вычислить. Можно передавать еще один параметр, задающий размер массива. Другой способ: определить структуру, содержащую указатель на массив и размер массива, и передавать ее как параметр. Например:

```
void compute1(int* vec_ptr, int vec_size);  // 1-ый способ

struct vec {              // 2-ой способ
  int* ptr;
  int size;
};

void compute2(vec v);
```

Сложнее с многомерными массивами, но часто вместо них можно использовать массив указателей, сведя эти случаи к одномерным массивам. Например:

```
char* day[] = {
"понедельник",
"вторник",
"среда",
"четверг",
 "пятница",
"суббота",
"воскресение" };
```

Теперь рассмотрим функцию, работающую с двумерным массивом — матрицей. Если размеры обоих индексов известны на этапе трансляции, то проблем нет:

```
void print_m34(int m[3][4])
{
for (int i = 0; i<3; i++) {
  for (int j = 0; j<4; J++)
   cout << ' ' << m[i][j];
cout << '\n';
        }
       }
```

Конечно, матрица по-прежнему передается как указатель, а размерности приведены просто для полноты описания. Первая

размерность для вычисления адреса элемента неважна, поэтому ее можно передавать как параметр:

```
void print_mi4(int m[][4], int dim1)
{
  for ( int i = 0; i<dim1; i++) {
     for ( int j = 0; j<4; j++)
        cout << ' ' << m[i][j];
     cout << '\n';
  }
}
```

Самый сложный случай — когда надо передавать обе размерности. Здесь «очевидное» решение просто непригодно:

```
void print_mij(int m[][], int dim1, int dim2)   // ошибка
{
  for ( int i = 0; i<dim1; i++) {
     for ( int j = 0; j<dim2; j++)
        cout << ' ' << m[i][j];
     cout << '\n';
  }
}
```

Во-первых, описание параметра m[][] недопустимо, поскольку для вычисления адреса элемента многомерного массива нужно знать вторую размерность. Во-вторых, выражение m[i][j] вычисляется как *(*(m+i)+j), а это, по всей видимости, не то, что имел в виду программист. Приведем правильное решение:

```
void print_mij(int** m, int dim1, int dim2)
{
for (int i = 0; i< dim1; i++) {
  for (int j = 0; j<dim2; j++)
 cout << ' ' << ((int*)m)[i*dim2+j];
  cout << '\n';
   }
}
```

Выражение, используемое для выбора элемента матрицы, эквивалентно тому, которое создает для этой же цели компилятор, когда известна последняя размерность. Можно ввести дополнительную переменную, чтобы это выражение стало понятнее:

```
int* v = (int*)m;
.....
 v[i*dim2+j]
```

Лучше такие достаточно запутанные места в программе упрятывать. Можно определить тип многомерного массива с соответствующей операцией индексирования. Тогда пользователь может и не знать, как размещаются данные в массиве

3.4 Перегрузка функций

Разные функции обычно имеют разные имена, но функциям, выполняющим сходные действия над объектами различных типов, иногда лучше дать возможность иметь одинаковые имена.

На C++ возможно определение нескольких функций с одинаковыми именами, но разными типами параметров. Компилятор C++ способен различать функции не только по ее именам, но и по составу списков параметров. Комплекс *имя функции + список ее формальных параметров* иногда называют *сигнатурой функции*. Если имена функций одинаковы, но сигнатуры различны, то компилятор всегда может различить их и выбрать для вызова нужную функцию. Функции с одинаковыми именами, но различными сигнатурами называются *перегруженными*.

При вызове перегруженной функции компилятор выбирает соответствующую функцию, анализируя количество и тип аргументов в вызове. Для этого сравниваются типы фактических параметров, указанные в вызове, с типами формальных параметров всех описаний функций с данным именем. В результате вызывается та функция, у которой формальные параметры наилучшим образом сопоставились с параметрами вызова, или выдается ошибка, если такой функции не нашлось. Например:

```
void print(double);
void print(long);
```

```
void f()
{
print(1L);   // будет вызвана print(long)
print(1.0);  // будет вызвана print(double)
print(1);    // ошибка, неоднозначность: что вызывать
          // print(long(1)) или print(double(1)) ?
}
```

Правила сопоставления типов фактических параметров в выражении вызова функций с типами формальных параметров, указанными при ее объявлении, применяются в следующем порядке по убыванию их приоритета:

1. Точное сопоставление: сопоставление произошло без всяких преобразований типа или только с неизбежными преобразованиями (например, имени массива в указатель, имени функции в указатель на функцию и типа T в const T).

2. Сопоставление с использованием стандартных целочисленных преобразований, определенных в (т.е. char в int, short в int и их беззнаковых двойников в int), а также преобразований float в double.

3. Сопоставление с использованием стандартных преобразований, определенных в C++ (например, int в double, unsigned в int).

4. Сопоставление с использованием пользовательских преобразований.

5. Сопоставление с использованием эллипсиса (многоточия ...) в описании функции.

Если найдены два сопоставления по самому приоритетному правилу, то вызов считается неоднозначным, процесс компиляции останавливается и выдается соответствующее сообщение об ошибке. Эти правила сопоставления параметров работают с учетом правил преобразований арифметических типов для C и C++.

Пусть имеются такие описания функции print:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);
```

Тогда результаты следующих вызовов print() будут такими:

```
void h(char c, int i, short s, float f)
{
print(c); // точное сопоставление: вызывается print(char)
print(i); // точное сопоставление: вызывается print(int)
print(s); // стандартное целочисленное преобразование:
          // вызывается print(int)
print(f); // стандартное преобразование:
          // вызывается print(double)
print('a'); // точное сопоставление: вызывается print(char)
print(49); // точное сопоставление: вызывается print(int)
print(0); // точное сопоставление: вызывается print(int)
print("a"); // точное сопоставление:
             // вызывается print(const char*)
  }
```

Обращение `print(0)` приводит к вызову `print(int)`, ведь `0` имеет тип **`int`**. Обращение print('a') приводит к вызову print(char), т.к. 'a' — типа char .

Отметим, что на разрешение неопределенности при перегрузке не влияет порядок описаний рассматриваемых функций, а типы возвращаемых функциями значений вообще не учитываются.

Исходя из этих правил, можно гарантировать, что если эффективность или точность вычислений значительно различаются для рассматриваемых типов, то вызывается функция, реализующая самый простой алгоритм. Например:

```
int pow(int, int);
double pow(double, double);   // из <math.h>
complex pow(double, complex);     // из <complex.h>
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);


void k(complex z)
{
int i = pow(2,2);     // вызывается pow(int,int)
double d = pow(2.0,2); // вызывается pow(double,double)
complex z2 = pow(2,z); // вызывается pow(double,complex)
complex z3 = pow(z,2); // вызывается pow(complex,int)
complex z4 = pow(z,z); // вызывается pow(complex,complex)
}
```

Таким образом, возможно определение нескольких функций с одинаковыми именами, но разными типами параметров. Эти функции называются перегруженными. При вызове перегруженной функции компилятор выбирает соответствующую функцию, анализируя количество и тип аргументов в вызове.

Перегруженные функции могут иметь разные или одинаковые типы возвращаемых значений и обязательно должны иметь разные списки параметров. Две функции, отличающиеся только типами возвращаемых значений, вызовут ошибку компиляции.

3.5 Указатели на функции

Указатель функции содержит адрес, по которому передается управление при вызове функции. В **C**++ указатель функции определяется как «*указатель функции возвращающей тип*

type_out *и воспринимающей аргументы типа* ***type1, type2, ...***». Это означает, что в указателе на функцию учитывается формат обмена информацией между функцией и вызывающей программой. Объявление указателя на функцию выглядит так:

*тип_возвращемого_значения (*имя_указателя) (список_типов_аргументов);*

Например:

float(*func)(int, char);

Здесь **func** — это имя указателя функции, возвращающей значение типа **double** и принимающей ***2*** аргумента, первый из которых типа **int**, второй — типа **char**. При формировании указателя на функцию, не возвращающую никаких значений, в качестве типа возвращаемого аргумента указывается **void**. Если функция, на которую объявляется указатель, не принимает никаких аргументов, то список типов аргументов оставляется пустым, например:

void (*UF)();

Здесь **UF** — это указатель на функцию, не возвращающую никаких значений и не принимающую аргументов.

Указатели на функцию настраиваются на конкретный интерфейс указываемой функции. Указатели на функции, объявленные для функций с различными интерфейсами несовместимы друг с другом.

Указатели на функции инициализируются именем функции. Например, известно что функция расчета синуса, объявлена в math.h как **double** `sin` (**double**). Можно построить указатель на функцию такого вида и инициализировать его именем функции `sin`:

double (*UY) (**double**)=sin;

Далее имя указателя **UY** может использоваться в программе как имя функции, например:

```
cout<< UY(M_PI/2);
```

Этот оператор выведет на экран значение $\sin(\pi/2) = 1$.

Можно объявлять массив из указателей на функцию и инициализировать его именами функций, например:

double (*Farr[3]) (**double**)={sin,cos,sqrt};

Теперь `Farr[0](x)` будет рассчитывать значение синуса `x`, `Farr[1](x)` — значение косинуса от `x`, а `Farr[2](x)` — значение корня квадратного из `x`.

3.6 Значения параметров по умолчанию

Может так случиться, что часть параметров функции в наиболее часто используемых случаях имеют фиксированное значение. Для более гибкого использования этих функций иногда применяются параметры, значение которых задается по умолчанию. Параметры, для которых значение по умолчанию задано, можно исключить из списка фактических параметров при вызове функций, тогда их значения будут инициализироваться тем, что указано по умолчанию. Таким образом, параметры, для которых предусмотрено значение по умолчанию, становятся необязательными.

Рассмотрим в качестве примера функцию печати целого числа. Вполне разумно применить в качестве необязательного параметра основание счисления печатаемого числа, хотя в большинстве случаев числа будут печататься как десятичные целые значения. Следующая функция

```
void print (int value, int base =10);

void F()
{ print(31);
   print(31,10);
   print(31,16);
   print(31,2);
}
```

напечатает такие числа: 31 31 1f 11111

Задавать значения по умолчанию можно только для подряд идущих формальных параметров в завершающей части их списка. Попытка наделить значениями по умолчанию параметры в середине списка вызывает ошибку на этапе компиляции:

```
int f(int, int =0, char* =0);  // нормально

int g(int =0, int =0, char*);  // ошибка

int h(int =0, int, char* =0);  // ошибка
```

Отметим, что в данном контексте наличие пробела между символами * и = весьма существенно, поскольку *= является операцией присваивания:

```
int nasty(char*=0);     // синтаксическая ошибка
```

Итак, C++ позволяет программисту задавать аргументы по умолчанию и их значения по умолчанию. Если аргумент по умолчанию пропускается при вызове функции, используется его значение по умолчанию. Аргументы по умолчанию должны быть крайними правыми (последними) в списке параметров функции. Аргументы по умолчанию должны быть указаны при первом же упоминании имени функции. Значения по умолчанию могут быть константами, глобальными переменными или вызовами функций.

3.7 Функции с неопределенным числом параметров

Существуют функции, в описании которых невозможно указать число и типы всех допустимых параметров. Тогда список формальных параметров завершается многоточием или эллипсисом (...), что означает: «и, возможно, еще несколько аргументов». Рассмотрим пример интерфейса функции форматированного вывода `printf`, прототип которой содержится в заголовочных файлах `stdio.h` и `conio.h`:

```
int printf(const char* ...);
```

При вызове `printf` обязательно должен быть указан параметр типа `char*`, однако могут быть (а могут и не быть) еще другие параметры. Например:

```
printf("Hello, world\n");

printf("Name is %s %s\n", frst_name, scnd_name);

printf("%d + %d = %d\n", 2,3,5);
```

В случае функции `printf` первый параметр является строкой, специфицирующей формат вывода. Она может содержать специальные символы, которые позволяют правильно воспринять

последующие параметры. Например, **%s** означает — «будет фактический параметр типа **char***», **%d** означает — «будет фактический параметр типа **int**». Но такой синтаксический разбор в компиляторе не производится, так что не известно, действительно ли ожидаемые параметры присутствуют в вызове функции и имеют ли они соответствующие типы. Например, следующий вызов

```
printf("My name is %s %s\n",2);
```

нормально транслируется, но приведет (в лучшем случае) к неожиданной выдаче.

В продуманной программе функции, в которых указаны не все типы параметров, могут потребоваться в виде исключения. Лучше использовать перегрузку функций или стандартные значения параметров, чем функцию с параметрами, типы которых не определены. Эллипсис становится необходимым только тогда, когда могут меняться не только типы, но и число параметров. Чаще всего эллипсис используется для определения интерфейса с библиотекой стандартных функций на Си, если этим функциям нет замены. В приведенных примерах функция printf — библиотечная функция Си.

3.8 Шаблоны функций

Шаблоны функций предоставляют возможность создания функций, которые выполняют одинаковые операции над разными типами данных, причем шаблон функции определяется только один раз

Часто встречаются функции, реализующие одни и те же действия для аргументов различных типов. Например, сортировка массива по возрастанию его элементов может выполняться одним и тем же методом и для данных типа **int** и для данных типа **double**. Различие состоит только в типах параметров и некоторых внутренних переменных.

В более поздние версии C++ включено специальное средство, позволяющее параметризовать определение функции, чтобы компилятор мог построить конкретную реализацию функции для

указанного типа параметров функции. Параметризованное определение функции строится по схеме:

```
template < class имя_класса >
    Заголовок функции
      { /* Тело функции */ }
```

Имя класса является параметром и задается идентификатором, локализованным в пределах определения функции. Хотя бы один из параметров функции должен иметь тип, соответствующий этому идентификатору.

Параметризованное определение функции сортировки массива методом перестановок может быть построено следующим образом:

```
template <class T >
  void sort ( T a[ ], int  n )
   {   T  temp;
      int sign;
      for ( int k = 0; k > n; k++)
       { sign = 0;
         for ( i = 0; i <n - k; i++)
          if ( a [ i ] > a [ i + 1])
             { temp = a [ i ];
               a[ i ] = a[ i + 1 ];
               a[ i + 1 ] = temp; sign++;
              }
          if ( sign == 0 ) break;
       }
       return;
  }
```

Если в программе будут объявлены массивы

```
int aint [10];
double afl [20];
```

и установлены значения элементов этих массивов, то вызов функции `sort(aint,10);` обеспечит вызов `sort` для упорядочения массива целых, а вызов функции `sort(afl,20)` обеспечит вызов sort для упорядочения массива с элементами типа **`double`**.

Если элементами массива являются объекты какого-либо определенного программистом класса, для которого определена операция отношения >, то функция sort может быть вызвана и для такого массива. Разумеется, в объектном коде программы будут присутствовать все варианты реально вызываемой функции sort. Параметризация функции сокращает объем исходного текста программы и повышает его надежность.

В описателе **template** можно указывать несколько параметров вида **class** *имя_типа*, а также параметры базовых типов. Например, функция

```
template < class T1, class T2 >
void copy ( T1 a[ ], T2 b[ ], int n)
      {  for ( int i = 0; i <n; i++)
          a[ i ] = b [ i ] ;
      }
```

копирует первые n элементов массива b типа T2 в первые n элементов массива a типа T1. Разумеется, программист несет ответственность за то, чтобы такое копирование было возможным.

3.8.1 Переопределение шаблонной функции

Пусть была определена шаблонная функция:

template <class T>

T max(T x, T y) { **return** (x > y) ? x : y; };

Можно переопределить генерацию шаблонной функции для конкретного типа с помощью обычной нешаблонной функции:

#include <string.h>

char *max(**char** *x,**char** *y){**return**(strcmp(x,y)>0) ?x:y;}

Если вызывается функция со строковыми аргументами, то она выполняется вместо автоматической шаблонной функции. В этом случае вызов функции позволяет избежать бессмысленного сравнения двух указателей.

Шаблонные функции, сгенерированные компилятором, выполняют только тривиальные преобразования аргументов.

Тип (типы) аргументов шаблонной функции должны использовать все формальные аргументы шаблона. Если это не так, то при вызове функции нельзя определить фактические значения для неиспользуемых параметров шаблона.

3.8.2 Явные и неявные шаблонные функции

При выполнении разрешения переопределения (после шагов по поиску точного совпадения) компилятор игнорирует шаблонные функции, сгенерированные неявно компилятором.

```
template<class T> T max(T a, T b)
{ return (a > b) ? a: b; }
void f(int i, char c)
{
max(i,i);    // вызывает max(int,int)
max(c,c);    // вызывает max(char, char)
max(i,c);    // вызывает max(int, char)
max(c,i);    // вызывает max(char, int)
}
```

Этот код дает следующие сообщения об ошибках:

```
Could not find a match for 'max(int,char)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(char,int)
```

"Не найдено соответствие для ... в функции ..."

Если пользователь явно определяет шаблонную функцию, то эта функция, с другой стороны, будет полностью участвовать в разрешении переопределения. Например:

```
template<class T> T max(T a, T b)
{ return (a > b) ? a: b;}
int max(int,int)          // явно объявляет max(int,int)
```

Тогда:

```
void f(int i, char c)
{
max(i,i);    // вызывает max(int,int)
max(c,c);    // вызывает max(char, char)
```

```
max(i,c);    // вызывает max(int, char)
max(c,i);    // вызывает max(char, int)
}
```

3.9 Резюме по теме фунцции

Функция активизируется посредством вызова функции. В вызове указывается имя функции и передается информация (в виде аргументов), которая нужна вызываемой функции для выполнения ее задачи. Выражение вызова функции состоит из имени функции, за которым в круглых скобках. следуют аргументы функции, список ее фактических параметров.

Каждый фактический параметр функции может быть константой, переменной или выражением.

Локальная переменная известна только в описании данной функции. Функции не знают детали реализации другой функции (включая локальные переменные). Цель скрытия информации в функциях заключается в том, чтобы дать доступ только к той информации, которая нужна для выполнения их задач. Это средство реализации принципа наименьших привилегий, одного из наиболее важных принципов разработки хорошего программного обеспечения.

Общий формат определения функции:

Тип-возвращаемого-значения имя-функции (список-параметров) { *Тело-функции* }

Тип-возвращаемого-значения устанавливает тип значения, возвращаемого в вызывающую функцию. Если функция не возвращает значение, *тип-возвращаемого-значения* объявляется как **`void`**.

Имя-функции — любой правильно написанный идентификатор.

Список-параметров — написанный через запятые список, содержащий объявления переменных, которые будут переданы функции. Если функция не принимает никаких значений, *список-параметров* оставляется пустым, либо объявляется как **`void`**.

Тело-функции — набор объявлений и операторов, которые составляют текст программы, реализующей функцию.

Аргументы, передаваемые функции, должны быть согласованы по количеству, типу и порядку следования с параметрами в определении функции.

Когда программа доходит до вызова функции, управление передается из точки активации к вызываемой функции, функция выполняется и управление возвращается оператору вызова.

Вызываемая функция может вернуть управление оператору вызова одним из трех способов. Если функция не возвращает никакого значения (объявлен тип возврата **void**), управление возвращается при достижении закрывающей операторной скобки, охватывающей тело функции или при выполнении оператора

return;

Если функция возвращает значение, то возврат производится оператором

return *выражение;*

Где тип и значение *выражения* и определяют то, что возвращается функцией в вызывающую программу.

Прототип (или объявление) функции объявляет тип возвращаемого значения функции, количество, типы и порядок следования параметров, передаваемых в функцию.

Прототипы функций дают возможность компилятору проверить, правильно ли вызывается функция.

Компилятор игнорирует имена переменных, упомянутые в прототипе функции.

Если аргумент передается в функцию по значению, создается копия значения переменной и именно она передается вызываемой функции. Изменения копии в вызываемой функции не влияют на значение исходной переменной.

Локальные переменные, объявленные в начале функции, имеют областью действия блок подобно параметрам функции, которые считаются локальными переменными функции.

Единственными идентификаторами, имеющими областью действия прототип функции, являются те, которые использованы в списке параметров прототипа функции. Идентификаторы, использованные в прототипе функции, можно повторно использовать в других местах программы без опасений возникновения неопределенности.

Функция, не возвращающая значение, объявляется с типом void. Если предпринять попытку вернуть значение функции или использовать результат активизации функции в вызывающем выражении, компилятор сообщит об ошибке.

Пустой список параметров указывается пустыми круглыми скобками или ключевым словом **void** в круглых скобках.

Встраиваемые функции исключают накладные расходы, связанные с вызовом функции. Программист может использовать ключевое слово **inline,** чтобы рекомендовать компилятору сгенерировать машинные коды функции в нужных местах программы (если это возможно), чтобы минимизировать вызовы функции. Компилятор может проигнорировать **inline.**

C++ предусматривает возможность прямой формы вызова по ссылке с помощью ссылочных параметров. Чтобы указать, что параметр функции передается по ссылке, после типа параметра в прототипе функции пишется символ &. В вызове функции переменная указывается по имени, но она будет передана по ссылке. В вызываемой функции обозначение переменной ее локальным именем на самом деле отсылает к исходной переменной в вызывающей функции. Таким образом, исходная переменная может быть изменена с помощью вызываемой функции.

Спецификация **const** может создать именованную константу. Именованная константа должна получить в качестве начального значения постоянное выражение и после этого не может изменяться. Именованные константы часто называют постоянными переменными или переменными только для чтения. Именованные константы могут быть помещены всюду, где может быть помещено постоянное выражение. Другим распространенным применением спецификации **const** является создание ссылок на константы.

Шаблоны функций предоставляют возможность создания функций, которые выполняют одинаковые операции над разными типами данных, причем шаблон функции определяется только один раз.

4 ОБЛАСТИ ВИДИМОСТИ, КЛАССЫ ПАМЯТИ И ВРЕМЯ СУЩЕСТВОВАНИЯ ОБЪЕКТОВ ПРОГРАММЫ

4.1 Область действия и область видимости

Областью видимости имени называется область исходного кода программы, из которого допустим нормальный доступ к связанному с этим именем объекту. Область действия имени объекта — участок программы, где объект существует. Обычно область действия и область видимости совпадают, однако бывают случаи, когда объект временно «скрыт» вследствие наличия идентификатора с тем же именем. Объект при этом не прекращает своего существования, но исходный идентификатор не может служить для доступа к нему до тех пор, пока не закончится область действия дублирующего идентификатора.

Область видимость не может выходить за пределы области действия, но область действия может превышать область видимости.

Таким образом, область видимости имени типа или имени объекта в программе — это та часть программы, где данное имя может быть использовано в составе операторов и выражений. Имена объектов — данных видимых в каком-либо фрагменте программы, доступны для чтения или чтения\записи своих значений. Видимые имена типов могут использоваться в описаниях переменных, в выражениях преобразования типов, в операторах, new и **`sizeof`**. Видимые имена функций могут использоваться в выражениях вызова.

В пределах *одной* области видимости имена *должны быть уникальны*. Если здесь ввести объекты с одинаковыми именами, возникнет неопределенность и компиляция прервется.

Существует четыре области действия: *блок, функция, файл и класс*. Здесь мы рассмотрим только первые три, область действия *класс*, рассмотрим далее.

Область действия блок. Имя, описанное в блоке, является локальным в этом блоке и может использоваться только в нем и во вложенных в него блоках, расположенных после точки описания. Имена формальных параметров в определениях функций

рассматриваются, как если бы они были описаны в самом объемлющем блоке этой функции.

Область действия функция. Метки можно использовать повсюду в функции, в которой они описаны. Только метки имеют область видимости, совпадающую с функцией.

Область действия файл Имя описанное вне всех блоков и классов имеет область видимости файл и может быть использовано в единице трансляции, в которой оно появляется после момента описания. Имена, описанные с файловой областью видимости, называются глобальными.

Имя может быть скрыто явным описанием того же имени в объемлющем блоке или классе это явление маскировки имен было рассмотрено выше.

Скрытое имя объекта с файловой областью видимости можно использовать, если оно предваряется унарной операцией ::

Области видимости любого объекта начинается в точке его описания. Точкой описания имени считается точка завершения описателя имени, перед инициализатором, если таковой имеется.

Например,

```
int x = 12;
{ int x = x; }
```

Здесь второе ***x*** инициализируется своим собственным (неопределенным) значением.

4.2 Время существования

Время существования, близко связанное с классом памяти, определяет продолжительность периода, в течение которого объявленным идентификаторам соответствуют распределенные в памяти реальные физические объекты. Также делается различие между объектами этапа компиляции и этапа выполнения. Например, переменным, в отличие от определяемых типов (typedefs) и типов, память непосредственно во время выполнения не распределяется. Существует три вида продолжительности времени существования: *статическое, локальное и динамическое.*

4.2.1 Статическое время существования (static)

Объекты со статическим временем существования получают распределение памяти сразу же при начале выполнения программы. Такое распределение памяти сохраняется до выхода из программы. Объекты со статической продолжительностью обычно размещаются в фиксированных сегментах данных, распределенных в соответствии с используемой моделью памяти.

Все функции, независимо от того, где они определены, являются объектами со статическим временем существования.

Также статическое время существования имеют все переменные с файловым контекстом. Прочим переменным может быть задано статическое время существования, если использовать явные спецификаторы класса памяти **`static`** или **`extern`**.

Например:

```
static int N;
extern float S;
```

При отсутствии явного инициализатора, либо (в C++) конструктора, объекты со статическим временем существования инициализируются нулем (или пустым значением).

Статическую продолжительность не следует путать с файловой или глобальной областью действия. Объект может иметь статическую продолжительность и при этом локальную область действия.

4.2.2 Локальное время существования

Объект с локальным временем существования всегда имеет локальную область действия, поскольку он не существует вне своего охватывающего блока. Обратное неверно: объект с локальной областью действия может иметь статическое время существования.

Объекты с локальным временем существования, называемые также динамическими локальными переменными, менее надежны. Они создаются в стеке (или в регистре) при входе в охватывающий их блок или функцию. При выходе программы из такого

блока или функции они уничтожаются. Объекты с локальной продолжительностью должны инициализироваться явно, в противном случае их исходное содержимое непредсказуемо. Объекты с локальной продолжительностью всегда должны иметь локальную область действия или область действия в границах функции. При объявлении переменных с локальным временем существования можно использовать спецификатор класса памяти **auto**, однако он является избыточным, поскольку **auto** для переменных, объявленных в блоке, всегда используется по умолчанию.

При объявлении переменных (например, **int, char, float**) спецификатор класса памяти **register** также подразумевает **auto**, однако компилятору при этом передается запрос (или рекомендация) о том, что при возможности данный объект желательно разместить в регистре. Компилятор можно настроить таким образом, чтобы он распределял регистр локальной интегральной переменной или переменной типа указатель, если какой-либо регистр свободен. Если свободных регистров нет, то переменная распределяется как **auto** (динамический локальный объект) без выдачи предупреждения или генерации ошибки.

4.2.3 Динамическое время существования

Объекты с динамическим временем существования создаются и разрушаются конкретными вызовами функций при выполнении программы. Им распределяется память из специального резерва памяти, называемого динамически распределяемой областью, при помощи либо стандартных библиотечных функций, как например `malloc`, либо при помощи операции C++ **new**. Соответствующая отмена распределения выполняется при помощи вызовов `free` или **delete**.

5 СТРУКТУРА ПРОГРАММЫ

Программа на C++ может состоять из нескольких исходных файлов, каждый из которых содержит описания типов, функций, переменных и констант. Чтобы имя можно было использовать в разных исходных файлах для ссылки на один и тот же объект, оно должно быть описано как внешнее.

Например:

```
extern double sqrt(double);
extern instream cin;
```

Практически в любой программе возникает необходимость в обращении к некоторому «стандартному» набору функций. Имеется обширные библиотеки функций, разработанные в разное время для Си и C++. Функции, вошедшие в некоторые из таких библиотек стали, по сути дела, стандартными компонентами языка.

Уже обсуждался тот факт, что для компиляции программы, использующей вызовы функций, требуется предварительное объявление (прототип) этих функций. Это описание прототипа должно быть сделано до того, как в тексте программы появится оператор вызова функции. Программа на C++ не компилируется, если какая-то функция не имеет прототипа и не описана перед ее использованием.

Очевидный способ обеспечить исходные тексты пользовательских программ описаниями прототипов библиотечных функций — это поместить такие описания в отдельные текстовые файлы, называемые заголовочными файлами (или h — файлами), а затем скопировать содержимое этих h-файлов во все файлы, где нужны эти описания.

Например, если прототип функции **`sqrt`** хранится в заголовочном файле для стандартных математических функций **`math.h`**, и необходимо эту функцию использовать, чтобы извлечь квадратный корень из 4, можно написать:

```
#include <math.h>
//...
x = sqrt(4);
```

5.1 Заголовочные файлы и препроцессор

Поскольку обычные заголовочные файлы включаются во многие исходные файлы, они не должны содержать повторяющихся описаний. Например, тела функций даются только для **`inline`** – подставляемых функций и инициализаторы даются только для констант. В заголовочный файл помещаются описания пользовательских типов, на C++ там размещаются протоколы классов. Таким образом, заголовочный файл обеспечивает интерфейс между отдельно компилируемыми частями программы.

Заголовочные файлы подключаются до работы компилятора и делает это специальная программа *препроцессор*.

В пользовательской программе указываются только команды препроцессору — лексемы начинающиеся символом #. Препроцессор не производит какого бы то ни было лексического разбора текста и, естественно, «не понимает», что такое идентификатор, константа, оператор и т.п., он делает преобразование одного текста в другой.

5.1.1 Директива `#include`

По всей вероятности, ни одна, сколько ни будь осмысленная программа на Си или C++ не может обойтись без использования библиотечных функций, хотя бы для вывода информации, поэтому любая программа начинается с директив подключения заголовочных файлов

`#include` < *имя_h-файла*>
или
`#include` “ *имя_h-файла”*

Обратим внимание на синтаксис: никаких разделителей типа точки с запятой здесь ставить не нужно.

В команде включения **`include`** имя файла, заключенное в угловые скобки, например, относится к файлу с этим именем в стандартном каталоге прописанном в параметрах среды разработки. На файлы, находящиеся в каких-либо других местах ссы-

лаются с помощью имен, заключенных в двойные кавычки. Например:

```
#include "math1.h"

#include "/usr/bs/math2.h"
```

включит ***math1.h*** из текущего пользовательского каталога, а math2.h из каталога /usr/bs.

Исполняя директиву **include,** препроцессор возьмет содержимое текстового файла — аргумента и вставит его содержимое на место директивы.

Заголовочные файлы с прототипами библиотечных функции могут использоваться как на Си, так и на C++. Естественно, и тексты описаний прототипов могут различаться. Могут быть различия, зависимые от настроек компилятора, редактора связей и т.п. Поэтому препроцессор может выполнять условные включения тех или иных фрагментов текста заголовочного файла. Препроцессор способен определять, было ли произведено подключение того или иного h — файла, и в соответствии с этим строить дальнейший процесс подключений. Это гарантирует пользователя от введения дублирующих описаний объектов. Процесс препроцессорной обработки управляется содержащимися в h — файлах директивами препроцессору **#if , #else, #endif** и т.п. Здесь не будем подробно освещать все директивы препроцессора, рассмотрим лишь еще одну популярную директиву.

5.1.2 Директива #define

Директива **#define** называется директивой *макроподстановки*. Смысл ее работы тот же, что и у команды «Найти \ Заменить» в редакторах текста. Синтаксис записи директивы:

#define Что_найти На_что_заменить

Например:

```
#define Esc 27
#define
```

Директива **#define** служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют именованными константами. Идентификаторы, заменяющие фрагменты программ, называют макроопределениями, причем макроопределения могут иметь аргументы.

```
#define MAX(x,y)  ((x)>(y))?(x):(y)
```

Эта директива заменит фрагмент

```
t=MAX(i,s[i]);
```

на фрагмент

```
t=((i)>(s[i])?(i):(s[i]);
```

Как и в предыдущем примере, круглые скобки, в которые заключены формальные параметры макроопределения, позволяют избежать ошибок связанных с неправильным порядком выполнения операций, если фактические аргументы являются выражениями. Например, при наличии скобок фрагмент

```
t=MAX(i&j,s[i]||j);
```

будет заменен на фрагмент

```
t=((i&j)>(s[i]||j)?(i&j):(s[i]||j);
```

а при отсутствии скобок — на фрагмент

```
t=(i&j>s[i]||j)?i&j:s[i]||j;
```

в котором условное выражение вычисляется совершенно в другом порядке.

5.2 Программа и связывание

Программа может состоять из одного или нескольких файлов, связываемых вместе. Рассмотрим, как взаимодействуют имена объектов программы, определяемые в разных файлах одной и той же программы.

Имя с файловой областью видимости, которое явно описано как **static**, является локальным в своей единице трансляции и может использоваться для именования других объектов, функций и т.п. в других единицах трансляции. Говорят, что такие имена имеют внутреннее связывание.

Имя с файловой областью видимости, которое явно описано со спецификацией **inline**, является локальным в своей единице трансляции.

Имя с файловой областью видимости, которое явно описано со спецификацией **const** и не описано явно как **extern**, считается локальным в своей единице трансляции.

Всякое описание некоторого имени с файловой областью видимости, которое не описано одним из перечисленных способов так, чтобы иметь внутреннее связывание, в многофайловой программе обозначает один и тот же объект.

Такие имена называются внешними или говорят, что они имеют внешнее связывание. В частности, поскольку нельзя описать имя класса как **static**, всякое употребление имени некоторого класса с файловой областью видимости, который использовался для описания объекта с внешним связыванием, или же который имеет статический член или функцию-член, не являющуюся подстановкой, будет обозначать один и тот же класс.

Имена определяемых типов (**typedef**), элементы перечисления или имена шаблонов типа имеют внутреннее связывание.

Статические члены класса допускают внешнее связывание.

Функции-члены, не являющиеся подстановкой, допускают внешнее связывание. Функции-члены, являющиеся подстановкой, должны иметь в точности одно определение в программе.

Локальные имена, явно описанные со спецификацией extern, имеют внешнее связывание, если только они уже не были описаны как **static** .

Типы, используемые во всех описаниях некоторого внешнего имени, должны совпадать, за исключением использования имен определяемых типов и указания границ массивов.

Должно быть *только одно определение* для каждой функции, объекта, класса и элемента перечисления, используемых в программе. Только если функция никогда не вызывается и ее адрес никогда не используется, ее не нужно определять.

Аналогично с именем класса, если его имя используется только таким образом, что не требуется знать определение класса, то класс не нужно определять.

Областью видимости функции может быть только файл или класс (см. ниже).

5.3 Начало и окончание программы

Программа должна содержать функцию с именем main(). Ей приписывается роль начала программы. Эта функция не является предопределенной для транслятора, она не может быть перегружена, а ее тип зависит от реализации. Предполагается, что любая реализация должна допускать два приведенных ниже определения и что можно добавлять после argv любые параметры. Функция main может определяться так

```
int main() { /* ... */ }
```

или

```
int main(int argc, char* argv[])
 { /* ... */ }
```

В последнем определении `argc` задает число параметров, передаваемых программе окружением, в котором она выполняется. Если argc не равно нулю, параметры должны передаваться как строки, завершающиеся символом '\0', с помощью argv[0] до argv[argc-1], причем argv[0] должно быть именем, под которым программа была запущена, или " ". Должно гарантироваться, что argv[argc]==0.

Функция main() не должна вызываться в программе. Связывание main() зависит от реализации. Нельзя получать адрес main() и не следует описывать ее как inline или static.

Вызов функции

```
void exit(int);
```

объявленной в <stdlib.h>, завершает программу. Значение параметра передается окружению программы в качестве результата программы.

Инициализация нелокальных статических объектов единицы трансляции происходит прежде первого обращения к функции или объекту, определенному в этой единице трансляции. Эта инициализация может быть проведена перед выполнением первого оператора **`main()`** или отложена до любого момента, предше-

ствующего первому использованию функции или объекта, определенных в данной единице трансляции.

Все статические объекты по умолчанию инициализируются нулем прежде любой динамической (во времени выполнения программы) инициализации. Больше никаких требований на порядок инициализации объектов из различных единиц трансляции не налагается.

Если в программе есть глобальные объекты, для которых существуют деструкторы, то они вызываются при возврате из `main()` или при вызове `exit()`.

Уничтожение происходит в обратном порядке по сравнению с инициализацией.

С помощью функции `atexit()` из <stdlib.h> можно указать функцию, которую нужно вызывать при выходе из программы.

Если было обращение к функции `atexit()`, объекты, инициализированные до вызова `atexit()`, не должны уничтожаться до тех пор, пока не произойдет вызов функции, указанной в atexit(). Если реализация C++ сосуществует с реализацией C, все действия, которые должны были произойти после вызова функции, заданной в atexit(), происходят только после вызова всех деструкторов.

Вызов функции

```
void abort();
```

описанной в <stdlib.h>, завершает программу без выполнения деструкторов статических объектов и без вызова функций, заданных в `atexit()`.

6 КЛАССЫ C++

Классы C++ предусматривают создание расширений системы преопределенных типов. Каждый тип класса представляет собой уникальное множество объектов и операций (правил), а также преобразований, используемых для создания, манипулирования и уничтожения таких объектов. Могут объявляться производные классы, наследующие элементы одного или более базовых (порождающих) классов.

В C++ структуры **`struct`** и объединения **`union`** рассматриваются как классы с определенными по умолчанию правилами доступа.

Упрощенный, в «первом приближении», синтаксис объявления класса следующий:

ключ_класса имя_класса <:*базовый_список*>[<*список_элементов*>]

Здесь «***ключ_класса***» — это одно из служебных слов: `class, struct или union.`

Необязательный «*базовый_список*» перечисляет базовый класс или классы, из которого «имя_класса» берет (или наследует) объекты и правила. Если объявлены некоторые базовые классы, то класс «имя_класса» называется производным классом. Базовый список содержит спецификаторы доступа по умолчанию и необязательные их переопределения, которые могут модифицировать права доступа производного класса к элементам базовых классов.

Необязательный «*список_элементов*» объявляет элементы класса (данные и функции) для «имени_класса» с используемыми по умолчанию и переопределяемыми спецификаторами доступа, которые могут влиять на то, какие функции к каким элементам класса могут иметь доступ.

Примечание: «*список_элементов*» класса заключается в фигурные скобки {}. Определение класса заканчивается точкой с запятой.

6.1 Имена классов

«Имя_класса» — это любой идентификатор, уникальный в пределах своего контекста. В классах структур и в объединениях «имя_класса» может опускаться.

6.2 Типы классов

Объявление создает уникальный тип, тип класса «имя_класса». Это позволяет объявлять последующие объекты класса (или вхождения класса) данного типа, а также объекты, являющиеся производными от этого типа, такие как указатели, ссылки, массивы объектов и т.д.

Пусть классом X определен тип объектов, тогда в программе возможно использование следующих его производных типов:

X x;	x — объект типа X
X& xr	xr — ссылка на тип X
X* xptr	xptr — указатель на тип X
X arr[10]	arr — массив из 10 элементов типа X
X func()	функция, возвращающая тип X

Примечание: Необходимо четко различать понятия: «тип объекта» и «экземпляр объекта». Например:

```
class X
{…};
main()
{ …
X x1,x2,x3;        // X-тип объекта; x1,x2,x3 – экземпляры объекта.
…}
```

6.3 Область действия имени класса

Область действия имени класса является локальной с некоторыми особенностями, характерными для классов. Область действия имени класса начинается с точки его объявления и закан-

чивается вместе с объемлющим блоком. Имя класса скрывает любой класс, объект, нумератор или функцию с тем же именем в объемлющей области действия. Если имя класса объявлено в области действия, содержащей описание объекта, функции или нумератора с тем же именем, обращение к классу возможно только при помощи уточненного спецификатора типа. Это означает, что с именем класса нужно использовать ключ класса, **`class`**, **`struct`** или **`union`**. Например:

```
class S { ... };
int S(class S *Sptr);
void func(void)
void main{
S t;          // (ОШИБКА!!!) недопустимое объявление: нет ключа класса и функции //S в области действия
class S s;    // так можно: есть уточнение ключом
S(&s);              // так можно: это вызов функции
}
```

C++ допускает также неполное объявление класса без элементов (например, просто **class X**;).Такие неполные объявления позволяют использовать некоторые ссылки на имя класса **X**, (обычно ссылки на указатели объектов классов) до того, как класс будет полностью определен. Разумеется, прежде чем объявить и использовать объекты классов, необходимо выполнить полное объявление класса, добавив соответствующий список элементов.

6.4 Объекты классов

Объекты классов могут присваиваться (если не было запрещено копирование), передаваться как аргументы функции, возвращаться функцией (за некоторыми исключениями) и т.д. Прочие операции с объектами и элементами классов могут различными способами определяться пользователем, включая функции элементы и «дружественные» функции, а также переопределение стандартных функций и операций при работе с объектами кон-

кретного класса. Переопределение функций и операций называется иногда совмещением. Операции и функции, ограниченные объектами конкретного класса (или взаимосвязанной группы классов) называются функциям элементами данного класса.

Язык C++ имеет механизм, позволяющий вызвать то же имя функции или операции для выполнения другой задачи, в зависимости от типа или числа аргументов или операндов.

6.5 Список элементов класса

Необязательный «список_элементов» представляет собой последовательность объявлений данных любого типа, включая другие классы, объявлений и определений функций, каждое из которых может иметь необязательные спецификаторы класса памяти и модификаторы доступа. Определенные таким образом объекты называются элементами класса. Элементы класса могут объявляться со спецификаторами класса памяти **`static.`**

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

6.6 Функции элементы

Функция, объявленная в пределах определения класса без спецификатора **`friend`**, называется функцией элементом класса. Функция, объявленные с модификатором **`friend`**, называется «дружественной» функцией класса.

Одно и то же имя может использоваться для обозначения более чем одной функции, при условии, что они отличны по типам или числу аргументов.

Примечание: Объявление функций элементов внутри определения класса и описание функций элементов вне этого определения отделяет интерфейс класса от его реализации. Это способствует высокому качеству разработки программного обеспечения.

6.7 Ключевое слово this

Нестатические функции элементы работают с объектом типа класса, с которым они вызываются. Например, если **x** — это объект класса **X**, а функция **f** — элемент **X**, то вызов функции **x.f()** работает с **x**. Аналогичным образом, если **xptr** представляет собой указатель объекта **X**, то вызов функции **xptr-> f()** работает с ***xptr**. Откуда **f** может знать, с каким **x** работать? C++ передает **f** указатель на **x**, называемый **this**. Указатель **this** передается как скрытый аргумент при вызове нестатических функций элементов.

Ключевое слово **this** представляет собой локальную переменную, доступную в теле нестатической функции элемента. **this** не требует объявлений, и на него редко встречаются явные ссылки в определении функции. Однако, это ключевое слово неявно используется в функции для ссылки к элементам. Если, например, вызывается **x.f(y)**, где **y** есть элемент **X**, то **this** устанавливается на **&x**, а **y** устанавливается на **this->y**, что эквивалентно **x.y**.

6.8 Статические элементы

Спецификатор класса памяти **static** может использоваться в объявлениях элементов данных и функций элементов класса. Такие элементы называются статическими и имеют свойства, отличные от свойств нестатических элементов. В случае нестатических элементов для каждого объекта класса «существует» отдельная копия. В случае же статических элементов существует только одна копия, а доступ к ней выполняется без ссылки на какой-либо конкретный объект класса. Если **x** — это статический элемент класса **X**, то к нему можно обратиться как **X::x** (даже если объекты класса **X** еще не созданы). Однако, можно выполнить доступ к **x** и при помощи обычных операций доступа к элементам. Например, в виде **y.x** и **yptr->x**, где **y** — это объект класса **X**, а **yptr** — это указатель объекта класса **X**, хотя выражения **y** и **yptr** еще не вычислены. В частности, статическая функция элемент может вызываться как с использованием специального синтаксиса вызова функций элементов, так и без него:

```
class X {
int member_int;
public:
static void func(int i, X* ptr);
};

void main(void)
{
X obj;
func(1, &obj);    // ОШИБКА!!!, если где-нибудь еще не определена глобальная func()
X::func(1, &obj);       // вызов static func() в X допустим только для статических                                    //функций
obj.func(1, &obj);      // то же самое (допустимо как для статических, так и //нестатичских функций)
}
```

Поскольку статическая функция элемент может вызываться без учета какого-либо конкретного объекта, она не имеет указателя **this**. В результате статическая функция элемент не имеет доступа к нестатическим элементам без явного задания объекта при помощи операции **.** или **->**. Например, с учетом объявлений, сделанных в предыдущем примере, func можно определить следующим образом:

```
void X::func(int i, X* ptr)
{
member_int = i; // на какой объект ссылается member_int? Ошибка !
ptr->member_int = 1;  // допустимо: теперь мы знаем!
}
```

Статические функции элементы не могут быть виртуальными функциями. Недопустимо иметь статическую и нестатическую функции элементы с одинаковыми именами и типами аргументов.

Объявление статического элемента данных в объявлении класса не является определением, поэтому где-нибудь еще долж-

но присутствовать определение, отвечающее за распределение памяти и инициализацию.

Статические элементы подчиняются обычным правилам доступа к элементам класса, за исключением того, что они могут инициализироваться.

```
class X {...
static int x;
...};
int X::x = 1;
```

Основное назначение статических элементов состоит в том, чтобы хранить данные общие для всех объектов класса. Например, число созданных объектов, либо использованный последним ресурс из пула, разделяемого всеми подобными объектами. Статические элементы используются также для:

- уменьшения числа видимых глобальных имен;
- для того, чтобы сделать очевидным, какие именно статические объекты какому классу принадлежат;
- разрешения управления доступам к их именам.

6.9 Область действия элемента

Выражение **X::func()** в примере, приведенном выше, использует имя класса **X** с модификатором области действия доступа, обозначающим, что функция **func**, хотя и определена «вне» класса, в действительности является функцией элементом **X** и существует в области действия **X**. Влияние **X::** распространяется на тело определения этой функции. Это объясняет, почему возвращаемое функцией значение **i** относится к **X::i**, **char* i** из **X**, а не к глобальной переменной **int i**. Без модификатора **X::** функция **func** представляла бы обычную, не относящуюся к классу функцию, возвращающую глобальную переменную **int i.**

Следовательно, все функции — элементы находятся в области действия своего класса, даже если они определены вне этого класса.

К элементам-данным класса **X** можно обращаться при помощи операций выбора **.** и -> (как и в структурах Си). Функции элементы можно вызывать также при помощи операций выбора. Например:

```
class X {
public:
int i;
char name[20];
X *ptr1;
X *ptr2;
void Xfunc(char *data, X* left, X* right);    // определение вынесено за
// пределы протокола класса
};
void f(void);
{
X x1, x2, *xptr=&x1;
x1.i = 0;
x2.i = x1.i;
xptr->i = 1;
x1.Xfunc("stan", &x2, xptr);
}
```

Если **m** является элементом класса **X**, то выражение **X::m** называется уточненным именем. Оно имеет тот же тип, что и **m**, и является именующим выражением только в том случае, если именующим выражением является **m**. Ключевой момент здесь состоит в том, что даже если имя класса **X** скрыто другим именем, уточненное имя **X::m** тем не менее обеспечит доступ к нужному имени класса, **m**.

Элементы класса не могут добавляться к нему в другом разделе Вашей программы. Класс **X** не может содержать объекты класса **X**, но может содержать указатели или ссылки на объекты класса **X** (отметим аналогию с типами структур и объединений Си).

Если функция элемент описана в определении класса, она автоматически встраивается **inline**. Функция элемент, описанная

вне определения класса, может быть сделана встраиваемой посредством явного использования ключевого слова **inline**.

Примечание: Использование принципов объектно-ориентированного программирования часто может упростить вызовы функций за счет уменьшения числа передаваемых параметров. Это достоинство объектно-ориентированного программирования проистекает из того факта, что инкапсуляция данных элементов и функций элементов внутри объекта дает функциям элементам право прямого доступа к данным элементам.

6.10 Вложенные типы

Класс, объявленный внутри другого класса, называется вложенным классом. Его имя является локальным для охватывающего класса; вложенный класс имеет область действия, лежащую внутри области действия охватывающего класса. Эта вложенность является чисто лексической. Вложенный класс не имеет никаких дополнительных привилегий в доступе к элементам охватывающего класса (и наоборот).

7 УПРАВЛЕНИЕ ДОСТУПОМ К ЭЛЕМЕНТАМ

Элементы класса получают атрибуты доступа либо по умолчанию (в зависимости от ключа класса и местоположения объявления), либо при использовании какого-либо из спецификаторов доступа: **public**, **private** или **protected**. Значение этих атрибутов следующие:

Спецификатор доступа	Характер доступа
Private	Элемент доступен только для функций — элементов данного класса и функций — «друзей» (**friend**) этого класса
Protected	То же самое, что для **private**, но кроме того, элемент доступен для функций элементов и «дружественных» функций в производных классах
Public	Элемент может использоваться любой функцией в области видимости класса

Примечание: спецификаторы доступа не действуют на функции, объявленные как дружественные (**friend**).

Элементы класса (**class**) по умолчанию имеют атрибут **private**, поэтому для переопределения данного объявления спецификаторы доступа **public** или **protected** должны задаваться явно.

Элементы структуры (**struct**) по умолчанию имеют атрибут **public**, но можно его переопределить при помощи явного задания спецификаторов доступа **public** или **protected**.

Элементы объединения (**union**) по умолчанию имеют атрибут **public**, не переопределяемый в дальнейшем, поэтому само задание спецификаторов доступа для элементов объединения недопустимо.

Модификатор доступа, принимаемый по умолчанию или заданный для переопределения атрибута доступа, остается действительным для всех последующих объявлений элементов, пока не встретится другой модификатор доступа. Например:

```
class X {
int i;   // X::i по умолчанию private
char ch;  // так же, как и X::ch
public:
int j;   // следующие два элемента - public
int k;
protected:
int l;   // X::l - protected
};

struct Y {
int i;   // Y::i по умолчанию public
private:
int j;   // Y::j - private
public:
int k;   // Y::k - public
};

union Z {
int i;   // public по умолчанию, других вариантов нет
double d;
};
```

Спецификаторы доступа могут перечисляться и группироваться в любой удобной последовательности. Например, можно сэкономить место при вводе программы, объявив все элементы **private** сразу, и т.д.

Примечание: Предпочтительнее делать все данные элементы класса закрытыми и использовать открытые функции элементы для задания и получения значений закрытых данных элементов. Такая архитектура помогает скрыть реализацию класса от его клиентов, что снижает число ошибок и улучшает модифицируемость программ.

Никогда не возвращайте из открытой функции элемента неконстантную ссылку (или указатель) на закрытый элемент данных. Возвращение такой ссылки нарушает инкапсуляцию класса.

7.1 Доступ к базовым и производным классам

Одна из наиболее важных особенностей объектно-ориентированного программирования это наследование. Наследование — способ повторного использования программного обеспечения, при котором новые классы создаются из уже существующих классов путем заимствования их атрибутов и функций и обогащения этими возможностями новых классов. Повторное использование кодов экономит время при разработке программ. Это способствует повторному использованию проверенного и отлаженного программного обеспечения и таким образом уменьшает число проблем, возникающих после того как система начнет функционировать.

При создании нового класса вместо написания полностью новых данных элементов и функций элементов Вы можете указать, что новый класс является *наследником* данных элементов и функций элементов ранее определенного базового класса. Этот новый класс называется *производным классом*.

Примечание: Создание производного класса не влияет на исходный или объектный код базового класса, сохранность базового класса оберегается наследованием.

При объявлении производного класса **D** базовые классы **B1, B2 ...** перечисляются в разделяемом запятой «базовом_списке»:

ключ_класса D **:** базовый_список {<список_элементов>}

Примечание: Поскольку сам базовый класс может являться производным классом, то вопрос об атрибуте доступа решается рекурсивно: отслеживанием цепочки наследования до тех пор, пока не будет достигнут «самый базовый» класс, породивший все остальные.

Класс **D** наследует все элементы базовых классов. (Переопределенные элементы базовых классов наследуются, и при необходимости доступ к ним возможен при помощи переопределений области действия). Производный класс **D** может использовать только элементы базовых классов с атрибутами **public** и **protected**. Однако, что будут представлять собой атрибуты доступа унаследованных элементов с точки зрения класс **D**?

Допустим, производному классу **D** может понадобиться общедоступный (**public**) элемент базового класса, но при этом надо сделать его недоступным для внешних функций. Для этого можно использовать спецификаторы доступа в «базовом_списке».

При объявлении производного класса **D** перед классами в базовом списке могут быть заданы спецификаторы доступа **public, protected** или **private**:

```
class D : public B1, private B2, ... { ... };
```

Эти спецификаторы не изменяют атрибутов доступа к элементам в базовых классах, но могут изменить атрибуты доступа к унаследованным элементам в производном классе. Спецификатор в базовом списке может быть опущен, тогда для классов по умолчанию принимается спецификатор **private**.

Производный класс наследует атрибуты доступа к элементам базового класса следующим образом:

Тип доступа в базовом классе	Тип доступа к наследованным элементам в производном классе при различных спецификаторах наследования		
	:public	:protected	:private
public:	public	protected	private
protected:	protected	protected	private
private:	недоступны	недоступны	недоступны

Во всех случаях элементы **private** базового класса были и остаются недоступными для функций элементов производного класса, пока в описании доступа базового класса не будут явно заданы объявления **friend**.

Действие спецификаторов доступа в базовом списке можно скорректировать при помощи «уточненного_имени» в объявлениях **public** или **protected** для производного класса. Например:

```
class B {
int a;       // по умолчанию private
public:
int b, c;
int Bfunc(void);
};

class X : private B {  // теперь в X a, b, c и Bfunc - private
int d;            // по умолчанию private.
public:
B::c;            // c была private; теперь она public
int e;
int Xfunc(void);
};

int Efunc(X& x);    // внешняя по отношению к B и X
```

Функция **Efunc** может использовать только имена с атрибутом **public**, например **c**, **e** и **Xfunc**. Функция **Xfunc** в **X** является производной от **private B**, поэтому она имеет доступ к:

«скорректированной_к_типу_public» **c**;

«private_относительно_X» элементам **B:b** и **Bfunc**;

собственным **private** и **public** элементам: **d**, **e** и **Xfunc**.

Однако, **Xfunc** не имеет доступа к элементу **a,** который является «private_относительно_B».

Каждый производный класс является кандидатом на роль базового класса для будущих производных классов. При *простом наследовании* класс порождается одним базовым классом. При *множественном наследовании* производный класс наследует несколькими базовыми классами (возможно не родственными).

Производный класс обычно добавляет свои собственные данные элементы и функции элементы, так что производный класс в общем случае больше своего базового класса. Производный класс более специфичен по своему функциональному назначению, более конкретен, чем его базовый класс, и представляет меньшую группу объектов. При простом наследовании предполагается, что производный класс будет выполнять примерно те же функции, что и базовый класс.

Примечание. Каждый объект производного класса является также объектом соответствующего базового класса. Обратное утверждение не верно: объект базового класса не является объектом порожденного им производного класса.

Некоторые замечания по повышению эффективности:

Если классы, полученные путем наследования, более широкие, чем необходимо, это ведет к потере ресурсов памяти и производительности. Наследуйте из класса только самое необходимое.

В объектно-ориентированных системах классы часто тесно связаны. «Факторизуйте» общие атрибуты и функции и помещайте их в базовом классе. Затем используйте наследование для формирования производных классов.

Производный класс содержит атрибуты и функции своего базового класса. Производный класс может также содержать дополнительные атрибуты и функции. При наследовании базовый класс может быть скомпилирован независимо от производного класса. При формировании производного класса нужно компилировать только дополнительные атрибуты и функции; их объединение с базовым классом сформирует производный класс.

Изменения в базовом классе не требуют изменений в производных классах до тех пор, пока открытый интерфейс базового класса остается неизменным. Однако, производные классы могут нуждаться при этом в перекомпиляции.

7.2 Прямые и косвенные базовые классы

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

7.3 Виртуальные базовые классы

При множественном наследовании базовый класс не может задаваться в производном классе более одного раза:

```
class B { ... };
class D : B, B { ... }:    // недопустимо
```

Однако, базовый класс можно передавать производному классу более одного раза косвенно, например, с помощью следующей цепочки наследования:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }
```

В данном случае в каждом объекте класса **Z** будут элементы двух подобъектов класса **B**, наследованные с **X** и **Y** соответственно. Если это вызывает проблемы, то к спецификатору базового класса должно быть добавлено ключевое слово **virtual**. Например:

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

Теперь **B** является виртуальным базовым классом, а класс **Z** имеет только один подобъект класса **B**.

7.4 Пример построения системы классов

Известно, что при объявлении массивов в Си/Си++ количество элементов массива задается константой и в дальнейшем не может быть изменено. При обращении к элементам массив отсутствует контроль выхода за пределы индексов массива, что приводит к трудно обнаруживаемым ошибкам в программах. Построим систему классов для обработки динамических массивов, в которые можно добавлять новые элементы и исключить возможность выхода за пределы текущего размера массива. Общие свойства массивов с такими свойствами, не зависящие от типа элементов массива, объединим в классе **TBase**, а для массивов с различными типами элементов образуем свои классы. Описания классов объединим в файле заголовков **TBASEARR.H**, а определения методов приведем в файле **TBASEARR.CPP**.

```
// файл TBASEARR.H
#include <string.h>
#include <iostream.h>
class Tbase        //базовый класс для массивов всех типов
{int size,         //размер элемента
count,             //текущее число элементов
maxCount,          //размер выделенной памяти в байтах
delta;             //приращение памяти в байтах
char *pmem;        //указатель на выделенную память
int changeSize();  //перераспределение памяти

protected:
void* getAddr( ){return (void*) pmem;};
void addNewItem(void*);   //добавление в конец массива
void error(const char* msg){cout <<msg<<endl;};

public:
int getCount() {return count;};
TBase(int s,int m,int d);
TBase();
TBase(TBase&);
~TBase();
};
/* Массив с элементами типа int */
class TIntArray: public TBase
{public:
int getElem(int index);       //  Значение элемента по индексу
void putElem(int index,int &pe);// Замена значения элемента по индексу
void addElem(int& el);        // Добавление элемента в конец массива
TIntArray& add(TIntArray&);   // Сложение двух массивов поэлементно
TIntArray& subtract(TIntArray&);      // Вычитание массивов
void printElem(int index);  // Вывод значения элемента на экран
void print();               // Вывод на экран всего массива
```

```
        TIntArray(int      m,int      d):TBase((int)sizeof(int),m,d){      };
/*Конструктор */
        TIntArray(TBase& a):TBase( a ){};    /*Конструктор */
        ~TIntArray();
        };
```

Определения методов приведены в файле TBASEARR.CPP:

```
        #include <iostream.h>
        #include <stdlib.h>
        #include <constrea.h>
        #include <tbasearr.h>
        /* Методы класса TBase  */
        TBase::TBase(int s,int m,int d):size(s),maxCount(m),delta(d)
        {char* p;
        int k;
        count = 0; p = pmem = new char [size * maxCount];
        for (k=0; k < maxCount; k++)
        { *p = '\0'; p++;}
        }
        TBase::TBase():size(1),maxCount(10),delta(1)
        {char* p;
        int k;
        count = 0; p = pmem = new char [size *maxCount];
        for (k=0; k < maxCount; k++)
        { *p = '\0'; p++;}
        }
        TBase::TBase(TBase&
b):size(b.size),maxCount(b.maxCount),delta(b.delta)
        {int k;
        count = b.count;  pmem = new char [size * maxCount];
        for (k=0; k < maxCount * size; k++)
        { pmem[k] = b.pmem[k];}
        }
        TBase::~TBase ()
        { delete [ ] pmem; }
```

8 «ДРУЗЬЯ» КЛАССОВ (FRIEND)

«Друг» **F** класса **X** — это функция или класс, которые, не являясь функцией элементом **X**, имеют тем не менее полные права доступа к элементам **X** private и protected. Во всех прочих отношениях **F** — это обычная с точки зрения области действия, объявлений и определений функция.

Поскольку **F** не является элементом **X**, она не лежит в области действия **X** и поэтому не может вызываться операциями выбора **x.F** и **xptr->F** (где **x** — это объект **X**, а **xptr** — это указатель объекта **X**).

Если в объявлении или определении функции в пределах класса **X** используется спецификатор **friend**, то такая функция становится «другом» класса **X**. Дружественные функции не зависят от их позиции в классе или спецификаторов доступа.

Например:

```
class X {
int i;          // private относительно X
friend void friend_func(X*, int);
/* friend_func не является private, хотя она и объявлена в разделе private */
public:
void member_func(int);
};      /* определения; для обеих функций отметим доступ к private int i*/
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }

X xobj;
/* отметим различие в вызовах функций */
friend_func(&xobj, 6);
xobj.member_func(6);
```

Можно сделать все функции класса **Y** дружественными для класса **X** в одном объявлении, объявив класс **Y** дружественным классу **X**:

```
class Y;          // неполное объявление
class X {
friend Y;
int i;
void member_funcX();
};

class Y; {
void friend_X1(X&);
void friend_X2(X*);
...
};
```

Функции, объявленные в **Y**, являются дружественными для **X**, хотя они и не имеют спецификаторов **friend**. Они имеют доступ к частным элементам **X** (**private**), таким как **i** и **member_funcX**.

Кроме того, отдельные функции элементы класса **X** также могут быть дружественными для класса **Y**:

```
class X {
...
void member_funcX();
}

class Y {
int i;
friend void X::member_funcX();
...
};
```

«Дружественность» классов не транзитивна: если **X** является дружественным для **Y**, а **Y** — дружественный для **Z**, это не означает, что **X** — дружественный **Z**. Однако, «дружественность» наследуется.

9 КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

Существует несколько специальных функций элементов, определяющих, каким образом объекты класса создаются, инициализируются, копируются и разрушается. Конструкторы и деструкторы являются наиболее важными из них. Они обладают большинством характеристик обычных функций элементов — Вы должны объявить и определить их в пределах класса, либо объявить их в классе, но определить вне его. Однако, они обладают и некоторыми уникальными свойствами.

Они не имеют объявлений значений возврата (даже **void**).

Они не могут быть унаследованы, хотя производный класс может вызывать конструкторы и деструкторы базового класса.

Конструкторы, как и большинство функций языка C++, могут иметь аргументы по умолчанию или использовать списки инициализации элементов.

Деструкторы могут иметь атрибут **virtual**, но конструкторы такового иметь не могут.

Нельзя работать с их адресами:

Если конструкторы и деструкторы не были заданы явно, то они могут генерироваться компилятором, могут запускаться при отсутствии явного вызова в программе. Любой конструктор или деструктор, создаваемый компилятором, должен иметь атрибут **public.**

Вызвать конструктор тем же образом, что и обычную функцию, нельзя. Вызов деструктора допустим только с полностью уточненным именем. Например, пусть — **X** тип, определяемый классом, **X *p** — указатель на этот тип, тогда **p->X::~X();** — допустимый вызов деструктора, но **X::X();** — недопустимый вызов конструктора в исполняемой программе. При определении и уничтожении объектов компилятор выполняет вызов конструкторов и деструкторов автоматически.

Конструкторы и деструкторы при необходимости распределения объекту памяти могут выполнять неявные вызовы операций **new** и **delete**.

Объект с конструктором или деструктором не может использоваться в качестве элемента объединения.

Если класс **X** имеет один или более конструкторов, то один из них запускается всякий раз при определении объекта **x** класса **X**. Конструктор создает объект **x** и инициализирует его. Деструкторы выполняют обратный процесс, разрушая объекты класса, созданные конструкторами.

Конструкторы активизируются также при создании локальных или временных объектов данного класса. Деструкторы активизируются всякий раз, когда эти объекты выходят из области действия.

9.1 Конструкторы

Конструкторы отличаются от прочих функций элементов тем, что имеют то же самое имя, что и класс, к которому они относятся. При создании или копировании объекта данного класса происходит неявный вызов соответствующего конструктора.

Объекты создаются, как только становится активной область действия переменной. Конструктор также запускается при создании временного объекта данного класса.

```
class x
{
public:
X();    // конструктор класса X
};
```

Конструктор класса **X** не может воспринимать **X** как аргумент:

```
class X {
...
public:
X(X);   // недопустимо
}
```

Параметры конструктора могут быть любого типа, за исключением класса, элементом которого является данный конструктор. Конструктор может принимать в качестве параметра

ссылку на свой собственный класс. В таком случае он называется конструктором копирования. Конструктор, не воспринимающий параметров вообще, называется конструктором, используемым по умолчанию. Далее мы рассмотрим эти конструкторы, используемые по умолчанию. Описание же конструктора копирования приведено ниже.

9.1.1 Конструктор, используемый по умолчанию

Конструктором, используемым по умолчанию для класса **X** называется такой конструктор, который не воспринимает никаких аргументов: **X::X()**. Если для класса не существует конструкторов, определяемых пользователем, то компилятор генерирует конструктор по умолчанию. При таких объявлениях, как **X x**, конструктор по умолчанию создает объект **x**.

Как и все функции, конструкторы могут иметь аргументы, используемые по умолчанию. Например, конструктор: **X::X(int, int = 0)** может воспринимать один или два аргумента. Если данный конструктор будет представлен только с одним аргументом, недостающий второй аргумент будет принят как **int 0**. Аналогичным образом, конструктор: **X::X(int = 5, int = 6)** может воспринимать два аргумента, один аргумент, либо не воспринимать аргументов вообще, причем в каждом случае используются соответствующие умолчания. Однако, конструктор по умолчанию **X::X()** не воспринимает аргументов вообще, и его не следует путать с конструктором, например, **X::X(int = 0),** который может либо воспринимать один аргумент, либо не воспринимать аргументов.

При вызове конструкторов следует избегать неоднозначностей. В следующем примере возможно неоднозначное восприятие компилятором конструктора, используемого по умолчанию и конструктора, воспринимающего целочисленный параметр:

```
class X
{
public:
X();
X(int i = 0);
```

```
};

main()
{
X one(10);// допустимо: используется  X::X(int)
X two;     // ОШИБКА! Неоднозначность: используется ли X::X()
           // или X::X(int = 0)
}
```

Примечание: Для каждого класса может существовать только один конструктор, используемый по умолчанию.

9.1.2 Конструктор копирования

Конструктор копирования для класса **X** это такой конструктор, который может вызываться с одним единственным аргументом типа **X: X::X(const X&)** или **X::(const X&, int = 0)**. В конструкторе копирования допустимыми также являются аргументы по умолчанию. Конструкторы копирования запускаются при копировании объекта данного класса, обычно в случае объявления с инициализацией объектом другого класса: **X x = y**. Если такой конструктор необходим, но в классе **X** не определен, компилятор генерирует конструктор копирования для класса **X** автоматически.

9.1.3 Переопределение конструкторов

Конструкторы можно переопределить, что позволяет создавать объекты в зависимости от значений, использованных при инициализации.

```
class X
{
int  integer_part;
double double_part;
public:
X(int i) { integer_part = i; }
X(double d)  { double_part = d }
};

main()
```

```
{
X one(10);// вызывает X::X(int) и устанавливает integer_part в значение
10
X one(3.14);// вызывает X::X(double) и устанавливает double_part
...
return 0;
}
```

9.1.4 Порядок вызова конструкторов

В случае, когда класс использует один или более базовых классов, конструкторы базовых классов запускаются до того, как будут вызваны конструкторы производного класса. Конструкторы базового класса вызываются в последовательности их объявления. Например, в следующей программе:

```
class Y {...}
class X : public Y {...}
X one;
```

сначала вызывается конструктор базового класса **Y()** , а затем — конструктор производного класса **X().**

В случае множественных базовых классов:

```
class X : public Y, public Z
X one;
```

первыми вызываются конструкторы базовых классов в последовательности их объявления **Y()** , **Z()** и лишь затем вызывается конструктор **X().**

Конструкторы виртуальных базовых классов запускаются до каких-либо невиртуальных базовых классов. Если иерархия содержит множественные виртуальные базовые классы, то конструкторы виртуальных базовых классов запускаются в последовательности их объявления. Затем, перед вызовом конструкторов производного класса, конструируются невиртуальные базовые классы.

Если виртуальный класс является производным от невиртуального базового класса, то для того, чтобы виртуальный базовый класс был сконструирован правильно, эта невиртуальная база должна являться первой. Код:

```
class X : public Y, virtual public Z
X one;
```

дает следующую последовательность:

```
Z();   // инициализация виртуального базового класса
Y();   // невиртуальный базовый класс
X();   // производный класс
```

Либо, в более сложном случае,

```
class base;
class base2;
class level1:public base2,virtual public base;
class level2:public base2,virtual public base;
class toplevel:public level1,virtual public level1;
toplevel view;
```

Порядок конструирования `view` будет принят следующим:

```
base();       // старший в иерархии виртуальный базовый класс
              // конструируется только однажды

base2();      // невиртуальная база виртуальной базы level2
              // вызывается для конструирования level2

level2();     // виртуальный базовый класс

base2();      // невирттуальная база для level1

level1();     // другая невиртуальная база
toplevel();
```

В случае, когда иерархия класса содержит множественные вхождения виртуального базового класса, данный базовый класс конструируется только один раз. Однако, если существуют и виртуальные, и невиртуальные вхождения базового класса, то конструктор класса запускается один раз для всех виртуальных вхождений и один раз для всех невиртуальных вхождений базового класса.

Конструкторы элементов массива запускаются в порядке возрастания индексов массива.

9.1.5 Инициализация класса

Объект класса только с общедоступными элементами (**public**) и без конструкторов или базовых классов (обычно это структура) может инициализироваться при помощи списка инициализаторов. При наличии конструктора класса объекты либо инициализируются, либо имеют конструктор по умолчанию. Последний используется в случае объектов без явной инициализации.

Объекты классов с конструкторами могут инициализироваться при помощи задаваемых в круглых скобках списков инициализаторов. Этот список используется как список передаваемых конструктору аргументов. Альтернативным способом инициализации является использование знака равенства, за которым следует отдельное значение.

Это отдельное значение может иметь тип первого аргумента, воспринимаемого конструктором данного класса. В этом случае дополнительных аргументов либо не существует, либо они имеют значения по умолчанию. Значение может также являться объектом данного типа класса. В первом случае для создания объекта вызывается соответствующий конструктор. В последнем случае для инициализации объекта вызывается конструктор копирования.

```
class X
{
int i;
public:
X(); // тела функций для ясности опущены
X(int x);
X(const X&);
};

main()
{
X one;       // запуск конструктора по умолчанию
X two(1);    // используется конструктор X::X(int)
X three = 1; // вызывает X::X(int)
```

```
    X four = one; // запускает X::X(const X&) для копиро-
вания
    X five(two); // вызывает X::X(cont X&)
```

Конструктор может присваивать значения своим элементам следующим образом. Он может воспринимать значения в качестве параметров и выполнять присваивание элементам переменным собственно в теле функции конструктора:

```
class X
{
int a, b;
public:
X(int i, int j) { a = i; b = j }
};
```

Либо он может использовать находящийся до тела функции список инициализаторов:

```
class X
{
int a, b;
public:
X(int i, int j) : a(i), b(j) {}
};
```

В обоих случаях инициализация **X x(1, 2)** присваивает значение **1** элементу `x::a` и значение **2** элементу `x::b`. Второй способ, а именно список инициализаторов, обеспечивает механизм передачи значений конструкторам базового класса.

Чтобы обеспечить вызов конструкторов базового класса из производного, конструкторы в базовом классе должны иметь спецификатор доступа **`public`** или **`protected`**.

```
class base1
{
int x;
public:
base1(int i) { x = i; }
};
```

```
class base2
{
int x;
public:
base2(int i) : x(i) {}
};

class top:public base1,public base2
{
int a, b;
public:
top(int i,int j):base(i*5),base2(j+i),a(i)
{b=j;}
};
```

В случае такой иерархии класса объявление **top one(1, 2)** приведет к инициализации **base1** значением **5**, а **base2** значением **3**. Методы инициализации могут комбинироваться друг с другом (чередоваться).

Как было описано выше, базовые классы инициализируются в последовательности описания. Затем происходит инициализация элементов, также в последовательности их объявления, независимо от их взаимного расположения в списке инициализации.

```
class X
{
int a,b;
public:
X(int i,j):a(i),b(a+j){}
};
```

В пределах класса объявление **X x(1,1)** приведет к присваиванию числа **1** элементу `x::a` и числа **2** элементу `x::b`.

Конструкторы базовых классов вызываются перед конструированием любых элементов производных классов. Значения производного класса не могут изменяться и затем влиять на создание базового класса.

```
class base
{
int x;
public:
base(int i) : x(i) {}
};

class derived : base
{
int a;
public:
derived(int i) : a(i*10), base(a) {}
//Обратите внимание! base будет передано неинициализированное а
}
```

В данном примере вызов производного **d(1)** не приведет к присваиванию элементу базового класса **x** значения **10**. Значение, переданное конструктору базового класса, будет неопределенным.

Если нужно иметь список инициализаторов в невстроенном конструкторе, не следует помещать этот список в протоколе класса. Вместо этого надо поместить его в точку определения функции:

```
derived::derived(int i):a(i){...}
```

9.2 Деструкторы

Деструктор класса вызывается для освобождения элементов объекта до уничтожения самого объекта. Деструктор представляет собой функцию элемент, имя которой совпадает с именем класса, перед которым стоит символ тильды (~). Деструктор не может воспринимать каких-либо параметров, а также не объявляет возвращаемого типа или значения.

```
class X
{
public:
```

```
~X(); // деструктор класса X
};
```

Если деструктор не объявлен для класса явно, компилятор генерирует его автоматически. Класс может иметь только один деструктор.

9.2.1 Вызов деструкторов

Вызов деструктора выполняется неявно, когда переменная выходит из своей объявленной области действия. Для локальных переменных деструкторы вызываются, когда перестает быть активным блок, в котором они объявлены. В случае глобальных переменных деструкторы вызываются как часть процедуры выхода после функции **main**.

Когда указатели объектов выходят за пределы области действия, неявного вызов деструктора не происходит. Это значит, что для уничтожения такого объекта операция **delete** должна задаваться явно.

Деструкторы вызываются строго в обратной последовательности относительно последовательности вызова соответствующих им конструкторов (см. выше).

Все глобальные объекты остаются активными до тех пор, пока не будут выполнены коды во всех процедурах выхода. Локальные переменные, включая те, что объявлены в функции **main**, уничтожаются при их выходе из области действия.

Деструктор может также вызваться явно, одним из двух следующих способов: косвенно, через вызов **delete**, или непосредственно, путем задания полностью уточненного имени деструктора. **Delete** можно использовать для уничтожения тех объектов, для которых память распределялась при помощи **new**. Явный вызов деструктора необходим только в случае объектов, которым при помощи вызова **new** распределялся конкретный адрес памяти.

```
class X
{...
~X();
...};
```

```
  void* operator new(size_t size, void *ptr)
  {
  return ptr;
  }
  char buffer[sizeof(x)];

  main()
  {
  X* pointer = new X;
  X* exact_pointer;
exect_pointer=new(&buffer)X;//указатель инициализируется адресом буфера
  ...
  delete pointer; // delete служит для разрушения указателя
exect_pointer->X::~X();//прямой вызов для отмены распределения памяти
  }
```

9.2.2 Виртуальные деструкторы

Деструктор может быть объявлен как виртуальный (**virtual**). Это позволяет указателю объекта базового класса вызывать необходимый деструктор в случае, когда указатель фактически ссылается на объект производного класса. Деструктор класса, производного от класса с виртуальным деструктором, сам является виртуальным.

```
class color
{
public:
virtual ~color(); // виртуальный деструктор для color
};

class red : public color
{
public:
~red();     // деструктор для red также является виртуальным
};

class brightred : public red
{
```

```
public:
~brightred();  // деструктор для brightred также виртуальный
};
```

Ранее перечисленные в следующих объявлениях классы:

```
color *palette[3];
palette[0] = new red;
palette[1] = new brightred;
palette[2] = new color;
```

даст следующие результаты:

delete `palette[0];` // Деструктор для red вызывается после деструктора для color

delete `palette[1];` // Деструктор для brightred вызывается после ~red и ~color

delete `palette[2];` // Запуск деструктора для color

Однако, если ни один из деструкторов не был объявлен виртуальным, **delete** `palette[0],` **delete** `palette[1]` и **delete** `palette[2]` вызывают только деструктор для класса `color`. Это приведет к неправильному уничтожению первых двух элементов, которые фактически имели тип `red` и `brightred`.

10 ПЕРЕОПРЕДЕЛЕННЫЕ ОПЕРАЦИИ

C++ позволяет переопределить действие большинства операций, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Как и в случае переопределяемых функций C++ в целом, компилятор определяет различия в функциях по контексту вызова, по числу и типам аргументов операндов.

Переопределение можно выполнить для всех операций используемых в C++, за исключением следующих операций:

Операция	Название	Пример
.	Обращение к элементу через имя объекта	Ob.field =2;
.*	Разыменование элемента-указателя через имя объекта	Ob.*Pfield =2:
::	Разрешение области видимости	X :: x = 7
?:	Условное выражение	a>b? a : b;

Также невозможно переопределение символов препроцессора # и **##.**

Ключевое слово **operator**, за которым следует символ операции, называется именем функции-операции. При определении нового (переопределенного) действия операции оно используется как обычное имя функции.

Операция-функция, вызываемая с аргументами, ведет себя как операция, выполняющая определенные действия с операндами в выражении. *Операция-функция не может изменять число аргументов или правила приоритета и ассоциативности операций, сравнительно с ее «нормальным» использованием*. Рассмотрим класс `complex`:

```
class complex {
double real, imag;      // по умолчанию private
public:
...
complex() { real = imag = 0; } // встроенный конструктор
complex(double r, double i=0) //еще один конструктор
{real = r; imag = i;}
...}
```

Примечание. Данный класс был создан только для задач иллюстрации. Он отличен от класса `complex` из библиотеки поддержки.

Мы можем легко разработать функцию для сложения комплексных чисел, например:

```
complex AddComplex(complex c1, complex c2);
```

однако будет более естественным иметь возможность записать:

```
complex c1(0,1), c2(1,0), c3
c3 = c1 + c2;
```

вместо:

```
c3 = AddComplex(c1, c2);
```

Операцию + можно легко переопределить, если включить в класс `complex` следующее объявление:

```
friend complex operator +(complex c1,complex c2};
```

и определить его (возможно, **`inline`**) следующим образом:

```
complex operator +(complex c1, complex c2)
{return complex(c1.real+c2.real,c1.imag+c2.imag);}
```

10.1 Операции-функции

Операции-функции можно вызывать непосредственно, хотя обычно они вызываются косвенно, при использовании переопределенных:

```
c3 = c1.operator +(c2);
```

Это то же, что и

```
c3 = c1 + c2;
```

В отличие от **`new`** и **`delete`**, которые имеют свои собственные правила, операция-функция должна быть либо нестатической функцией элементом, либо иметь как минимум один аргумент типа класса. Операции-функции =, **(), []** и **->** должны представлять собой нестатическими функции элементыПреопределенные операции и наследование

За исключением операции-функции присваивания **=()** все операции-функции для класса **X** наследуются классом, производным от **X**, согласно стандартным правилам разрешения для пере-

определенных функций. Если **X** является базовым классом для **Y**, переопределенную операцию-функцию для класса **X** можно далее переопределить для класса **Y**.

10.2 Переопределение *new* и *delete*

Операции **new** и **delete** могут переопределяться таким образом, чтобы давать альтернативные варианты подпрограммы управления динамически распределяемой памятью.

Определяемая пользователем операция `new` должна возвращать **void*** и иметь в качестве первого аргумента **size_t**. Определяемая пользователем операция `delete` должна иметь тип **void** и первый аргумент **void*** .

Второй аргумент, типа **size_t**, не является обязательным. Тип **size_t** определяется в файле **stdlib.h**. Например:

```
#include <stdlib.h>
class X {...
public:
void* operator new(size_t size)
{return newalloc(size);}
void operator delete(void* p} {newfree(p); }
X() { /* инициализация */}
X(char ch) { /* здесь тоже */ }
~X() { /* очистка */ }
...};
```

Аргумент `size` задает размер создаваемого объекта, а `newalloc` и `newfree` — это определяемые пользователем функции распределения и отмены распределения памяти. Вызовы конструктора и деструктора для объектов класса **X** (или объектов, производных от **X**, для которых не существует собственных переопределенных операций **new** и **delete**) приведет к запуску соответствующих определяемых пользователем операций **X::operator new()** и **X::operator delete()**, соответственно.

Операции-функции **X::operator new** и **X::operator delete** являются статическими элементами **X**, как при явном объявлении их **static**, так и без него, поэтому они не могут быть виртуальными функциями.

Стандартные, предопределенные (глобальные) операции **new** и **delete** могут при этом также использоваться в области действия **X**, как явно с операциями глобальной области действия (**::operator new** и **::operator delete**), так и неявно, при создании и разрушении объектов классов, отличных от класса **X** и не являющихся производными от класса **X**. Например, можно использовать стандартные операции **new** и **delete** при определении совмещенных версий:

```
void* X::operator new(size_t)
{
void* ptr = new char[s]; // вызов стандартной new
...
return ptr;
}
void X::operator delete(void* ptr)
{
delete (void*) ptr; // вызов стандартной delete
}
```

Причиной того, что размер аргумента определяется **size**, является то, что классы, производные от **X**, наследуют **X::operator new.**

Размер объекта производного класса может существенно отличаться от размера, определяемого базовым классом.

10.3 Переопределение унарных операций

Перегрузка префиксной или постфиксной унарной операции выполняется при помощи объявления нестатической функции элемента, не воспринимающей никаких аргументов, либо при помощи объявления функции, не являющейся функцией элементом, воспринимающей один аргумент. Если **@** представляет собой унарную операцию, то **@x** и **x@** можно интерпретировать как

x.operator@() и **operator@**(x), соответственно, в зависимости от объявления. Если объявление было сделано в обеих формах, то разрешение неоднозначности зависит от переданных при вызове операции стандартных аргументов.

10.4 Переопределение бинарных операций

Переопределение бинарной операции выполняется при помощи объявления нестатической функции элемента, воспринимающей *один* аргумент, либо при помощи объявления не являющейся элементом функции (обычно **friend**), воспринимающей *два* аргумента. Если @ представляет собой бинарную операцию, то **x@y** можно интерпретировать либо как **x operator@(y)**, либо как **operator@(x,y)**, в зависимости от выполненных объявлений. Если объявлены обе формы, то разрешение неоднозначности зависит от переданных при вызове операции стандартных аргументов.

10.4.1 Переопределение операции присваивания =

Операцию присваивания = можно переопределить только при помощи объявления нестатической функции элемента. Например:

```
class String {
...
String& operator = (String& str);
...
String (String&)
~String();
}
```

Данный код, совместно с соответствующим определением функции String::**operator**=(), позволяет выполнять необходимые действия при присваивания содержимого одного объекта другому. Например, выполнить операцию присваивания строк str1 = str2, как это делается в других языках программирования.

В отличие от прочих функций — операций, *функция-операция присваивания не может наследоваться производными классами.* Если для какого-либо класса **X** не существует определяемой операции =, то операция = определяется по умолчанию, как поэлементное присваивание элементов класса **X.** Напомним, *побитовое присваивание не допустимо для объектов, использующих динамическую память.*

10.4.2 Переопределение операции вызова функции ()

Вызов функции:

первичное_выражение(<список_выражений>)

рассматривается в качестве двоичной операции с операндами «*первичное_выражение*» и «*список_выражений*» (возможно, пустой). Соответствующая функция-операция — это **`operator().`** Данная функция может являться определяемой пользователем для класса **X** (и любых производных классов) только в качестве нестатической функции элемента. Вызов `x(arg1, arg2),` где **x** представляет собой объект класса **X**, интерпретируется в таком случае как **`x.operator()(`**`arg1,arg2`**`).`**

10.4.3 Переопределение операции индексирования []

Аналогичным образом, операция индексирования:

первичное_выражение [выражение]

рассматривается как бинарная операция с операндами «первичное_выражение» и «выражение». Соответствующая операция-функция — это **`operator[].`** Она может определяться пользователем для класса **X** (и любых производных классов) только посредством нестатической функции элемента. Выражение `x[y],` где x это объект класса **X**, интерпретируется в данном случае как x.**operator**[](y).

10.5 Переопределение операции доступа к элементу класса ->

Доступ к элементу класса при помощи:

первичное_выражение -> выражение

рассматривается как унарная операция. Функция **operator->** должна рассматриваться как нестатическая функция элемента. Выражение **x-> m**, где **x** — это объект класса **{,** интерпретируется как **(x.operator -> ())->m**, таким образом, что **operator->()** должен либо возвращать указатель на объект данного класса, либо возвращать объект класса, для которого определяется **`operator->.`**

11 ВИРТУАЛЬНЫЕ ФУНКЦИИ

При описании объектных типов функции, имеющие сходное назначение в разных классах, могут иметь одинаковые имена, типы параметров и возвращаемого значения. При обращении к такой функции с указанием имени объекта компилятору известно, какая из одноименных функций требуется. В то же время к объектам производного типа можно обращаться по указателю на базовый тип и тогда на этапе компиляции нельзя установить, функция какого из производных типов должна быть вызвана. В ходе выполнения программы требуется проверять, на объект какого типа ссылается указатель и после такой проверки вызывать требуемую функцию. Эти действия называют *«поздним»* связыванием, в отличие от *«раннего»* связывания, при котором уже на этапах компиляции или редактирования связей можно установить адрес точки входа вызываемой функции. В объектно-ориентированных языках программирования для решения этой проблемы применяются виртуальные методы.

Виртуальные функции позволяют производным классам обеспечивать разные версии функции базового класса. Вы можете объявить виртуальную функцию в базовом классе и затем переопределить ее в любом производном классе, даже если число и тип ее аргументов остается без изменений. Можно объявить функции `int Base:: Fun (int)` и `int Derived::Fun (int)`, даже если они не являются виртуальными. Версия базового класса доступна объектам производного класса через переопределение области действия. Если фунции являются виртуальными, то доступна только функция, связанная с фактическим типом объекта.

Примечание. Виртуальные функции могут быть только функциями элементами.

В виртуальных функциях нельзя изменять интерфейс, недопустимым является переопределение виртуальной функции таким образом, чтобы она отличалась только типом возврата. Если две функции с одним и тем же именем имеют разные аргументы, компилятор C++ рассматривает их как разные функции, механизм виртуальных функций игнорируется и работает механизм перегрузки функций.

Если базовый класс **B** содержит виртуальную функцию **vf**, а класс **D**, являющийся производным от класса **B**, содержит функцию **vf** того же типа, то если функция **vf** вызывается для объекта **d** или **D**, выполняется вызов **D::vf**, даже если доступ определен через указатель или ссылку на **B**. Например:

```
struct B {
virtual void vf1();
virtual void vf2();
virtual void vf3();
void f();
}
class D : public B {
virtual void vf1();//спецификатор virtual допустим, но избыточен
void vf2(int);  // не virtual, поскольку здесь используется другой список аргументов
char f3();  // так нельзя: изменяется только тип возврата
void f();
};

void extf()
{
D d;              // объявление объекта D
B* bp = &d;       // стандартное преобразование из D* в B*
bp->vf1();        // вызов D::vf1
bp->vf2();        // вызов B::vf2, так как vf2 из D имеет отличные аргументы
bp->f();    // вызов B::f (не виртуальной)
}
```

Переопределяющая функция **vf1** в классе **D** автоматически становится виртуальной. Спецификатор **virtual** может использоваться в определении переопределяющей функции в производном классе, но на самом деле он является в данном случае избыточным.

Интерпретация вызова виртуальной функции *зависит от типа того объекта,* для которого она вызывается. В случае вызова не виртуальных функций интерпретация зависит только *от типа указателя или ссылки*, указывающих объект, для которого она вызывается.

Виртуальные функции должны представлять собой элементы некоторого класса, но они не могут быть статическими элементами. Виртуальная функция может являться дружественной (**friend**) для другого класса.

Виртуальная функция в базовом классе, как и все функции элементы базового класса, должна определяться или объявляться как функция без побочного эффекта.

В классе, производном от такого базового класса, каждая чистая» функция должна быть определена или переобъявлена в качестве таковой. См. следующий раздел «Абстрактные классы».

Если виртуальная функция определена в базовом классе, то нет необходимости ее переопределения в производном классе. При вызовах будет просто вызвана соответствующая базовая функция.

Виртуальные функции заставляют определенным образом расплачиваться за свою универсальность: каждый объект производного класса должен содержать указатель на таблицу функций с тем, чтобы во время выполнения программы вызвать нужную (позднее связывание).

12 АБСТРАКТНЫЕ КЛАССЫ

Имеются случаи, в которых полезно определять классы, для которых программист не намерен создавать какое-либо объекты. Такие классы называются абстрактными классами. Классы, объекты которых могут быть реализованы, называются конкретными классами.

Абстрактным называется класс с как минимум одной чистой виртуальной функцией. Чистой виртуальной функцией является такая функция, у которой в ее объявлении тело определено как 0 (инициализатор равен 0), например:

```
class B {
virtual void vf(int)=0;//=0 означает "чистую" функцию
```

Абстрактный класс может использоваться только в качестве базового класса для других классов. Объекты абстрактного класса создаваться не могут. Абстрактный класс не может использоваться как тип аргумента или как тип возврата функции. Однако, допускается объявлять указатели на абстрактный класс. Допустимы ссылки на абстрактный класс при условии, что при инициализации не требуется создание временного объекта. Например:

```
class shape {// абстрактный класс
point center;
…
public:
point where() { return center; }
move(point p) { center = p; draw(); }
virtual void rotate(int)=0;// "чистая" виртуальная функция
virtual void draw()=0;      // "чистая" виртуальная функция
virtual void hilite() = 0;// "чистая" виртуальная функция
…
}

shape x;  // ошибка: попытка создания объекта абстрактного класса
shape* sptr;  // указатель на абстрактный класс допустим
shape f();//ошибка: абстрактный класс не может являться типом
                        // возврата
```

```
int g(shape s);    // ошибка: абстрактный класс не может являться
                   //типом аргумента функции
shape& h(shape&); //ссылка на абстрактный класс в качестве типа //возврата или аргумента функции допустим
```

Предположим, что класс **D** является производным непосредственно от абстрактного базового класса **B**. Тогда для каждой чистой виртуальной функции **pvf** в **B**, если **D** не обеспечивает определение для **pvf**, то **pvf** становится «чистой» функцией элементом **D**, а сам **D** будет абстрактным классом. Например, с использованием показанного выше класса **shape**:

```
class circle : public shape { // circle является производным от абстрактного класса
    int radius;               // private
    public:
    void rotate(int) { }      // определяется виртуальная функция: действия по вращению
    // окружности отсутствуют
    void draw();              // circle::draw должна быть где-либо определена
```

Функции — элементы могут объявляться из конструктора абстрактного класса, но вызов чистой виртуальной функции непосредственно или косвенно из такого конструктора приводит к ошибке времени выполнения.

12.1 Полиморфизм

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

Иногда функция элемент определена в базовом классе не как виртуальная, но переопределена в производном классе. Если такая функция элемент вызывается через указатель базового класса,

то используется версия базового класса. Если же эта функция элемент вызывается через указатель производного класса, то используется версия производного класса. Это не полиморфное поведение.

Благодаря применению виртуальных функций и полиморфизму вызов функции элемента может привести к различным действиям, которые зависят от типа вызываемого объекта.

Полиморфизм способствует расширяемости: программное обеспечение, использующее полиморфный механизм, пишется независимо от типов объектов, которым отправляются сообщения. Таким образом, новые типы объектов, которые должны реагировать на соответствующие сообщения, могут включаться в такую систему без модификации ее основы. За исключением кода пользователя, который создает новые объекты, программа не требует перекомпиляции.

Примечание: абстрактный класс определяет интерфейс для разных типов иерархии классов. Абстрактный класс включает чистые виртуальные функции, которые будут определены в производных классах. Все функции в иерархии могут применять один и тот же интерфейс, используя полиморфизм.

осторожность при интерпретации таких фраз, как «охватывающая область действия» и «точка объявления».

13 ОБЛАСТЬ ДЕЙСТВИЯ КЛАССА

Имя **M** элемента класса **X** имеет область действия класса «локальную по отношению **X**». Оно может использоваться в следующих ситуациях:

- в функциях элементах **X**;
- в выражениях типа **x.M**, где **x** есть объект **X**;
- в выражениях типа **`xptr->M`**, где **`xptr`** — указатель объекта **X**;
- в выражениях типа **X::M;**
- в выражениях типа **D::M**, где **D** — производный класс от **X**;
- в ссылках вперед в пределах класса, которому принадлежит элемент.

Классы, перечисления или имена **`typedef`**, объявленные в пределах класса X, либо имена функций, объявленные как «дружественные» для X, не являются элементами X. Их имена просто имеют охватывающую область действия.

13.1 Скрытые имена

Имя может быть скрыто явным объявлением того же имени в охватывающем (внешнем) блоке или в классе. Скрытый элемент класса, тем не менее, остается доступным при помощи модификатора области действия, заданного с именем класса **X:M**. Скрытое имя с областью действия файла (глобальное) допускает ссылку на него при помощи унарной операции **::,** например **::g**. Имя класса **X** может быть скрыто именем объекта, функции, либо нумератора, заданного в области действия класса **X**, независимо от последовательности объявления имен. Однако, имя скрытого класса **X** доступно при использовании префикса **X** с соответствующим ключевым словом **`class, struct`** или **`union`**.

Точка объявления имени **x** находится непосредственно после его полного объявления, но до инициализатора, если таковой существует.

13.2 Краткое изложение правил определения области действия в C++

Следующие правила применимы ко всем именам, включая имена **typedef** и имена классов, при условии, что то или иное имя допустимо в C++ в конкретной обсуждаемой области действия:

1. Сначала само имя проверяется на наличие неоднозначностей. Если в пределах области действия неоднозначности отсутствуют, то инициируется последовательность доступа.

2. Если ошибок управления доступа не обнаружено, то проверяется тип объекта, функции, класса, **typedef**, и т.д.

3. Если имя используется вне какой-либо функции или класса, либо имеет префикс унарной операции доступа к области действия **::,** и если имя не уточнено бинарной операцией **::** или операциями выбора элемента **.** или **->,** то это имя должно быть именем глобального объекта, функции или перечислимого типа.

4. Если имя **n** появляется в одном из видов: **X::n**, **x.n** (где **x** — это объект класса **X** или ссылка на **X**), либо **ptr->n** (где **ptr** — это указатель на **X**), то **n** является именем элемента **X** или элементом класса, производным от которого является **X**.

5. Любое до сих пор не рассмотренное имя, используемое в качестве статической функции элемента, должно быть объявлено в блоке, в котором оно встречается, либо в объемлющем блоке, либо являться глобальным именем. Объявление локального имени **n** скрывает объявления **n** в объемлющих блоках и глобальные объявления **n**. Имена в различных областях действия не могут переопределяться.

6. Любое не рассмотренное до сих пор имя, используемое в качестве имени нестатической функции элемента класса **X**, должно быть объявлено в блоке, в котором оно встречается, либо в объемлющем блоке, быть элементом класса **X** или базового относительно **X** класса, либо быть глобальным именем. Объявление локального имени **n** скрывает объявления **n** в охватывающих блоках, функциях элементах функциях и глобальных объявлениях **n**. Объявление имени элемента скрывает объявления этого имени в базовых классах.

7. Имя аргумента функции в определении функции находится в контексте самого внешнего блока данной функции. Имя ар-

гумента функции в неопределяющем объявлении функции вообще не имеет области действия. Область действия аргумента по умолчанию объявляется точкой описания данного аргумента, но не позволяет доступ к локальным переменным или нестатическим элементам класса. Аргументы по умолчанию вычисляются в каждой точке вызова.

8. Инициализатор конструктора вычисляется в области действия самого внешнего блока конструктора, и поэтому разрешает ссылки на имена аргументов конструктора.

14 ШАБЛОНЫ КЛАССОВ

Шаблон класса (также называемый родовым классом или генератором классов) позволяет вам задавать образец для определений классов. Хорошими примерами в этом смысле являются родовые классы **container**. Рассмотрим следующий пример класса векторов (одномерный массив). Когда Вы имеете вектор из целых чисел или данных любого другого типа, базовые операции, выполняемые с типом, являются одними и теми же (вставка, удаление, индексирование и т.д.).

В случае типа элемента, рассматриваемого как параметр `type` для класса, система будет на ходу генерировать определения класса, надежного по типам:

```
#include <iostream.h>
template <class T>      // Определение шаблона класса
class Vector
{
T *data;
int size;
public:
Vector(int);
~Vector() {delete[] data;}
T& operator[](int i) {return data[i];}
};    // Обратите внимание на синтаксис для внешних определений

template <class T>
Vector<T>::Vector(int n)
{
data = new T[n];
size = n;
}

main()
{
Vector<int> x(5);  // Сгенерировать вектор из целых
for (int i = 0; i < 5; ++i)
x[i] = i;
for (i = 0; i < 5; ++i)
cout << x[i] << ' ';
cout << '\n';
```

```
return 0;
}          //  на выходе будет 0 1 2 3 4
```

Как и в случае шаблонов функций, для переопределения автоматического определения для данного типа может быть реализовано явное определение шаблонного класса:

```
class Vector<char *> {...};
```

Символ `Vector` всегда должен сопровождаться типом данных в угловых скобках. Он не может появляться отдельно, за исключением некоторых случаев в определении оригинального шаблона.

14.1 Аргументы

Хотя в этих примерах используется только один шаблонный аргумент, допускается и несколько аргументов. Кроме типов данных, шаблонные аргументы могут также представлять значения:

```
template<class T,int size=64> class Buffer{...};
```

Шаблонные аргументы, не являющиеся типами, такие как **size**, могут иметь аргументы, принимаемые по умолчанию. Значением для таких аргументов может быть выражение-константа:

```
const int N = 128;
int i = 256;
Buffer<int, 2*N> b1;     // правильно
Buffer<float,i>b2;// ошибка: i не является константой
```

Так как каждое предписание значения для шаблонного класса на самом деле является классом, то он принимает свою собственную копию статических элементов. Аналогичным образом, шаблонные функции получают свои собственные копии статических локальных переменных.

14.2 Угловые скобки

Следует соблюдать аккуратность при использовании правой угловой скобки при предписании значения:

Buffer<**char**, (x > 100 ? 1024 : 64) > buf;

В этом примере, если опустить круглые скобки вокруг второго аргумента, то **>** между **x** и **100** будет досрочно закрывать список шаблонных аргументов.

14.3 Родовые списки, надежные по типам

В общем случае, когда нужно запрограммировать множество очень похожих вещей, вспомните о шаблонах. Проблемы с определением следующего класса, класса родовых списков,

```
class GList
{
public:
void insert( void * );
void *peek();
};
```

заключаются в том, что он не является надежным по типам и общие решения требуют повторяющихся определений класса. Так как проверка типов при вставке отсутствует, то у вас нет способа узнать, что же Вы получите обратно. Проблему надежности по типам можно решить, создав класс возврата

```
class FooList: public GList
{
public:
void insert( Foo *f ) { GList::insert( f ); }
Foo *peek() { return (Foo *)GList::peek(); }
};
```

Этот класс является надежным по типам. **insert** будет лишь брать аргументы типа **указатель_на_Foo** или **объект_полученный_из_Foo**, поэтому внутренний контейнер будет только содержать указатели, которые на самом указывают на что-то типа **Foo**. Это означает, что приведение типа **FooList::peek** всегда безопасно и создает правильный **FooList**. Теперь чтобы

проделать то же самое для **BarList**, **BazList** и т.д., нужны повторные определения отдельных классов. Для решения проблемы повторных определений классов с точки зрения безопасности типов, вновь воспользуемся шаблонами.

```
template <class T> class List : public GList
{
public:
void insert( T *t ) { GList::insert( t); }
T *peek() {return (T *)GList::peek(); }
}
List<Foo> fList; // создать класс FooList и экземпляр с именем fList
List<Bar> bList; // создать класс BarList и экземпляр с именем bList
List<Baz> zList; // создать класс BazList и экземпляр с именем zList
```

С помощью шаблонов Вы можете создавать какие угодно надежные по типам списки, используя простые объявления. При этом никакие коды, связанные с преобразованием типов из каждого класса возврата, не генерируются, то есть, накладные расходы, связанные с надежностью по типам, не возникают.

14.4 Исключение указателей

Другим методом является включение фактических объектов, что делает ненужными указатели, а также уменьшает число требуемых вызовов виртуальных функций, так как компилятор знает фактические типы объектов. Этот метод дает большие выгоды, если виртуальные функции достаточно малы для того, чтобы быть эффективно встроенными. При вызове через указатели встраивание функций достаточно сложно, так как компилятор не знает фактические типы указываемых объектов.

Ниже приводится определение шаблона, исключающее указатели:

```
template <class T> aBase
{
```

```
private
T buffer;
};

class anObject : public aSubject, public aBase<aFilebuf>
{
//...
};
```

Все функции в **aBase** могут вызывать функции, определенные в **aFilebuf**, непосредственно, без необходимости прибегать к указателю. И если какая-либо из функций в **aFilebuf** может быть встроена, то Вы получите большой выигрыш по скорости, так как шаблоны допускают, чтобы они были встроенными.

15 ПОТОКИ

Язык C++ не обеспечивает средств для ввода/вывода. Ему это и не нужно; такие средства легко и элегантно можно создать с помощью самого языка Разработка и реализация стандартных средств ввода/вывода для языка программирования зарекомендовала себя как заведомо трудная работа. Традиционно средства ввода/вывода разрабатывались исключительно для небольшого числа встроенных типов данных. Однако в C++ программах обычно используется много типов, определенных пользователем, и нужно обрабатывать ввод и вывод также и значений этих типов. Очевидно, средство ввода/вывода должно быть простым, удобным, надежным в употреблении, эффективным и гибким, и ко всему прочему полным. Ничье решение еще не смогло угодить всем, поэтому у пользователя должна быть возможность задавать альтернативные средства ввода/вывода и расширять стандартные средства ввода/вывода применительно к требованиям приложения.

C++ разработан так, чтобы у пользователя была возможность определять новые типы столь же эффективные и удобные, сколь и встроенные типы. Поэтому обоснованным является требование того, что средства ввода/вывода для C++ должны обеспечиваться в C++ с применением только тех средств, которые доступны каждому программисту. Описываемые здесь средства ввода/вывода представляют собой попытку ответить на этот вызов.

Средства ввода/вывода связаны исключительно с обработкой преобразования типизированных объектов в последовательности символов и обратно. Есть и другие схемы ввода/вывода, но эта является основополагающей в системе UNIX, и большая часть видов бинарного ввода/вывода обрабатывается через рассмотрение символа просто как набор бит, при этом его общепринятая связь с алфавитом игнорируется. Тогда для программиста ключевая проблема заключается в задании соответствия между типизированным объектом и принципиально не типизированной строкой.

Обработка и встроенных и определенных пользователем типов однородным образом и с гарантией типа достигается с помощью одного перегруженного имени функции для набора функций вывода. Например:

```
put(cerr,"x = "); // cerr - поток вывода ошибок
put(cerr,x);
put(cerr,"\n");
```

Тип параметра определяет то, какая из функций put будет вызываться для каждого параметра. Это решение применялось в нескольких языках. Однако ему недостает лаконичности. Перегрузка операции << значением «поместить в» дает более хорошую запись и позволяет программисту выводить ряд объектов одним оператором. Например:

```
cerr << "x = " << x << "\n";
```

где cerr — стандартный поток вывода ошибок. Поэтому, если x является int со значением 123, то этот оператор напечатает в стандартный поток вывода ошибок

```
x = 123
```

и символ новой строки. Аналогично, если X принадлежит определенному пользователем типу complex и имеет значение (1,2.4), то приведенный выше оператор напечатает в cerr

```
x = 1,2.4)
```

Этот метод можно применять всегда, когда для x определена операция <<, и пользователь может определять операцию << для нового типа.

15.1 Вывод

В этом разделе сначала обсуждаются средства форматного и бесформатного вывода встроенных типов, потом приводится стандартный способ спецификации действий вывода для определяемых пользователем типов.

15.1.1 Вывод Встроенных Типов

Класс ostream определяется вместе с операцией << («поместить в») для обработки вывода встроенных типов:

```
class ostream {
    // ...
public:
    ostream& operator<<(char*);
    ostream& operator<<(int i) { return
*this<
```

15.1.2 О выборе знака для переопределяемой операции ввода/вывода

Операция вывода используется, чтобы избежать той многословности, которую дало бы использование функции вывода. Но почему был выбран оператор << ?

Возможности изобрести новый лексический символ, как известно, нет. Операция присваивания была кандидатом одновременно и на ввод, и на вывод, но оказывается, большинство людей предпочитают, чтобы операция ввода отличалась от операции вывода. Кроме того, = не в ту сторону связывается (ассоциируется), то есть cout=a=b означает cout=(a=b).

Делались попытки использовать операции < и >, но значения «меньше» и «больше» настолько прочно вросли в сознание людей, что новые операции ввода/вывода во всех реальных случаях оказались нечитаемыми. Помимо этого, "<" находится на большинстве клавиатур как раз на ",", и у людей получаются операторы вроде такого:

```
cout < x , y , z;
```

Для таких операторов непросто выдать хорошие сообщения об ошибках.

Операции << и >> к такого рода проблемам не приводят, они асимметричны в том смысле, что их можно проассоциировать с «в» и «из», а приоритет << достаточно низок, чтобы можно

было не использовать скобки для арифметических выражений в роли операндов. Например:

```
cout << "a*b+c=" << a*b+c << "\n";
```

Естественно, при написании выражений, которые содержат операции с более низкими приоритетами, скобки использовать надо. Например:

```
cout << "a^b|c=" << (a^b|c) << "\n";
```

Операцию левого сдвига тоже можно применять в операторе вывода:

```
cout << (a<<2);
```

15.1.3 Форматированный вывод

Пока << применялась только для неформатированного вывода, и на самом деле в реальных программах она именно для этого главным образом и применяется. Помимо этого существует также несколько форматирующих функций, создающих представление своего параметра в виде строки, которая используется для вывода. Их второй (необязательный) параметр указывает, сколько символьных позиций должно использоваться.

```
  char* oct(long, int =0);     // восьмеричное
представление
  char* dec(long, int =0);     // десятичное
представление
  char* hex(long, int =0);     // шестнадцати-
ричное представление
  char* chr(int, int =0);      // символ
  char* str(char*, int =0);    // строка
```

Если не задано поле нулевой длины, то будет производиться усечение или дополнение; иначе будет использоваться столько символов (ровно), сколько нужно. Например:

```
cout << "dec(" << x
     << ") = oct(" << oct(x,6)
     << ") = hex(" << hex(x,4)
     << ")";
```

Если x==15, то в результате получится:

```
dec(15) = oct(    17) = hex(   f);
```

Можно также использовать строку в общем формате:

```
char* form(char* format ...);
cout<
```

15.1.4 Виртуальная функция вывода

Иногда функция вывода должна быть virtual. Рассмотрим пример класса shape, который дает понятие фигуры:

```
class shape {
   // ...
public:
   // ...
   virtual void draw(ostream& s); // рисует "this" на "s"
};

class circle : public shape {
   int radius;
public:
   // ...
   void draw(ostream&);
};
```

То есть, круг имеет все признаки фигуры и может обрабатываться как фигура, но имеет также и некоторые специальные свойства, которые должны учитываться при его обработке.
Чтобы поддерживать для таких классов стандартную парадигму вывода, операция << определяется так:

```
ostream& operator<<(ostream& s, shape* p)
{
    p->draw(s);
    return s;
}
```

Если next — итератор, то список фигур распечатывается например так:

```
while ( p = next() ) cout << p;
```

15.2 Файлы и Потоки

Потоки обычно связаны с файлами. Библиотека потоков создает стандартный поток ввода cin, стандартный поток вывода cout и стандартный поток ошибок cerr. Программист может открывать другие файлы и создавать для них потоки.

15.2.1 Инициализация потоков вывода

Класс ostream имеет конструкторы:

```
class ostream {
  // ...
   ostream(streambuf* s);  // связывает с буфером потока
   ostream(int fd);        // связывание для файла
  ostream(int size, char* p); // связывет с вектором
};
```

Главная работа этих конструкторов — связывать с потоком буфер. streambuf — класс, управляющий буферами. Класс filebuf является производным от класса streambuf.

Описание стандартных потоков вывода cout и cerr, которое находится в исходных кодах библиотеки потоков ввода/вывода, выглядит так:

```
    // описать подходящее пространство буфера
char cout_buf[BUFSIZE]

    // сделать "filebuf" для управления этим пространством
    // связать его с UNIX'овским потоком вывода 1 (уже открытым)
   filebuf cout_file(1,cout_buf,BUFSIZE);

    // сделать ostream, обеспечивая пользовательский интерфейс
   ostream cout(&cout_file);
   char cerr_buf[1];

    // длина 0, то есть, небуферизованный
    // UNIX'овский поток вывода 2 (уже открытый)
   filebuf cerr_file()2,cerr_buf,0;
   ostream cerr(&cerr_file);
```

15.2.2 Закрытие Потоков Вывода

Деструктор для ostream сбрасывает буфер с помощью открытого члена функции ostream::flush():

```
   ostream::~ostream()
   {
   flush(); // сброс
   }
```

Сбросить буфер можно также и явно. Например:

```
   cout.flush();
```

15.2.3 Открытие Файлов

Точные детали того, как открываются и закрываются файлы, различаются в разных операционных системах и здесь подробно не описываются. Поскольку после включения становятся доступны cin, cout и cerr, во многих (если не во всех) программах не нужно держать код для открытия файлов. Вот, однако, програм-

ма, которая открывает два файла, заданные как параметры командной строки, и копирует первый во второй:

```
   #include

  void error(char* s, char* s2)
  {
      cerr << s << " " << s2 << "\n";
      exit(1);
  }

  main(int argc, char* argv[])
  {
   if (argc != 3) error("неверное число парамет-
ров","");

      filebuf f1;
      if (f1.open(argv[1],input) == 0)
          error("не могу открыть входной
файл",argv[1]);
      istream from(&f1);

      filebuf f2;
      if (f2.open(argv[2],output) == 0)
         error("не могу создать выходной
файл",argv[2]);
      ostream to(&f2);

      char ch;
      while (from.get(ch)) to.put(ch);

      if (!from.eof() !! to.bad())
          error("случилось нечто странное","");
  }
```

Последовательность действий при создании `ostream` для именованного файла та же, что используется для стандартных потоков: (1) сначала создается буфер (здесь это делается посредством описания `filebuf`); (2) затем к нему подсоединяется файл (здесь это делается посредством открытия файла с помощью функции `filebuf::open()`); и, наконец, (3) создается сам

ostream с filebuf в качестве параметра. Потоки ввода обрабатываются аналогично.

Файл может открываться в одной из двух мод:

```
enum open_mode { input, output };
```

Действие filebuf::open() возвращает 0, если не может открыть файл в соответствие с требованием. Если пользователь пытается открыть файл, которого не существует для output, он будет создан.

Перед завершением программа проверяет, находятся ли потоки в приемлемом состоянии. При завершении программы открытые файлы неявно закрываются.

Файл можно также открыть одновременно для чтения и записи, но в тех случаях, когда это оказывается необходимо, парадигма потоков редко оказывается идеальной. Часто лучше рассматривать такой файл как вектор (гигантских размеров). Можно определить тип, который позволяет программе обрабатывать файл как вектор.

15.2.4 Копирование потоков

Есть возможность копировать потоки. Например:

```
cout = cerr;
```

В результате этого получаются две переменные, ссылающиеся на один и тот же поток. Главным образом это бывает полезно для того, чтобы сделать стандартное имя вроде cin ссылающимся на что-то другое.

15.3 Ввод

Ввод аналогичен выводу. Имеется класс istream, который предоставляет операцию >> («взять из») для небольшого множества стандартных типов. Функция operator>> может определяться для типа, определяемого пользователем.

15.3.1 Ввод встроенных типов

Класс istream определяется так:

```
class istream {
      // ...
  public:
      istream& operator>>(char*);       // строка
      istream& operator>>(char&);       // символ
      istream& operator>>(short&);
      istream& operator>>(int&);
      istream& operator>>(long&);
      istream& operator>>(float&);
      istream& operator>>(double&);
      // ...
  };
```

Функции ввода определяются так:

```
  istream& istream::operator>>(char& c);
  {
    // игнорирует пропуски
      int a;
    // неким образом читает символ в "a"
      c = a;
  }
```

Пропуск определяется как стандартный пропуск в C, через вызов isspase() в том виде, как она определена в (пробел, табуляция, символ новой строки, перевод формата и возврат каретки).
В качестве альтернативы можно использовать функции get():

```
  class istream {
      // ...
istream& get(char& c);              // char
istream& get(char* p, int n, int ='\n');  // строка
  };
```

Они обрабатывают символы пропуска так же, как остальные символы. Функция istream::get(char) читает один и тот же символ в свой параметр; другая istream::get читает не более n символов в

вектор символов, начинающийся в p. Необязательный третий параметр используется для задания символа остановки (иначе, терминатора или ограничителя), то есть этот символ читаться не будет. Если будет встречен символ ограничитель, он останется как первый символ потока. По умолчанию вторая функция get будет читать самое большее n символов, но не больше чем одну строку, '\n' является ограничителем по умолчанию. Необязательный третий параметр задает символ, который читаться не будет. Например:

```
cin.get(buf,256,'\t');
```

будет читать в buf не более 256 символов, а если встретится табуляция ('\t'), то это приведет к возврату из get. В этом случае следующим символом, который будет считан из cin, будет '\t'.
Стандартный заголовочный файл определяет несколько функций, которые могут оказаться полезными при осуществлении ввода:

```
int isalpha(char)    // 'a'..'z' 'A'..'Z'
int isupper(char)    // 'A'..'Z'
int islower(char)    // 'a'..'z'
int isdigit(char)    // '0'..'9'
int isxdigit(char)   // '0'..'9' 'a'..'f' 'A'..'F'
int isspase(char)// ' ' '\t' возврат новая строка
                        // перевод формата
int iscntrl(char)    // управляющий символ
                     // (ASCII 0..31 и 127)
int ispunct(char)    // пунктуация: ни один из вышеперечисленных
int isalnum(char)    // isalpha() | isdigit()
int isprint(char)    // печатаемый: ascii ' '..'-'
int isgraph(char) //isalpha()|isdigit()|ispunct()
int isascii(char c)  { return 0<=c &&c<=127; }
```

Все кроме isascii() реализуются внешне одинаково, с применением символа в качестве индекса в таблице атрибутов символов. Поэтому такие выражения, как

```
(('a'<=c&&c<='z')||('A'<=c&&c<='Z')) // алфавитный
```

не только утомительно пишутся и чреваты ошибками (на машине с набором символов EBCDIC оно будет принимать неалфавитные символы), они также и менее эффективны, чем применение стандартной функции:

```
isalpha(c)
```

15.3.2 Состояния потока

Каждый поток (istream или ostream) имеет ассоциированное с ним состояние, и обработка ошибок и нестандартных условий осуществляется с помощью соответствующей установки и проверки этого состояния.

Поток может находиться в одном из следующих состояний:

```
enum stream_state{_good,_eof,_fail,_bad};
```

Если состояние _good или _eof, значит последняя операция ввода прошла успешно. Если состояние _good, то следующая операция ввода может пройти успешно, в противном случае она закончится неудачей. Другими словами, применение операции ввода к потоку, который не находится в состоянии _good, является пустой операцией. Если делается попытка читать в переменную v, и операция оканчивается неудачей, значение v должно остаться неизменным (оно будет неизменным, если v имеет один из тех типов, которые обрабатываются функциями членами istream или ostream). Отличия между состояниями _fail и _bad очень незначительно и представляет интерес только для разработчиков операций ввода. В состоянии _fail предполагается, что поток не испорчен и никакие символы не потеряны. В состоянии _bad может быть все что угодно.

Состояние потока можно проверять например так:

```
switch (cin.rdstate()) {
  case _good:
     // последняя операция над cin прошла успешно
     break;
  case _eof:
```

```
    // конец файла
    break;
  case _fail:
    // некоего рода ошибка форматирования
    // возможно, не слишком плохая
    break;
  case _bad:
    // возможно, символы cin потеряны
    break;
  }
```

Для любой переменной z типа, для которого определены операции << и >>, копирующий цикл можно написать так:

```
while (cin>>z) cout << z << "\n";
```

Например, если z — вектор символов, этот цикл будет брать стандартный ввод и помещать его в стандартный вывод по одному слову (то есть, последовательности символов без пробела) на строку.

Когда в качестве условия используется поток, происходит проверка состояния потока и эта проверка проходит успешно (то есть, значение условия не ноль) только если состояние _good. В частности, в предыдущем цикле проверялось состояние istream, которое возвращает cin>>z. Чтобы обнаружить, почему цикл или проверка закончились неудачно, можно исследовать состояние. Такая проверка потока реализуется операцией преобразования.
Делать проверку на наличие ошибок каждого ввода или вывода действительно не очень удобно, и обычно источником ошибок служит программист, не сделавший этого в том месте, где это существенно. Например, операции вывода обычно не проверяются, но они могут случайно не сработать. Парадигма потока ввода/вывода построена так, чтобы когда в C++ появится (если это произойдет) механизм обработки исключительных ситуаций (как средство языка или как стандартная библиотека) его будет легко применить для упрощения и стандартизации обработки ошибок в потоках ввода/вывода.

15.3.3 Ввод типов, определяемых пользователем

Ввод для пользовательского типа может определяться точно так же, как вывод, за тем исключением, что для операции ввода важно, чтобы второй параметр был ссылочного типа. Например:

```
istream& operator>>(istream& s, complex& a)
/*
    форматы ввода для complex; "f" обозначает float:
    f
    ( f )
    ( f , f )
*/
{
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(_bad);    // установить state
    }
    else {
        s.putback(c);
        s >> re;
    }

    if (s) a = complex(re,im);
    return s;
}
```

Несмотря на то, что не хватает кода обработки ошибок, большую часть видов ошибок это на самом деле обрабатывать будет.

Локальная переменная с инициализируется, чтобы ее значение не оказалось случайно '(' после того, как операция окончится неудачно. Завершающая проверка состояния потока гарантирует,

что значение параметра a будет изменяться только в том случае, если все идет хорошо.
Операция установки состояния названа clear() (очистить), потому что она чаще всего используется для установки состояния потока заново как _good. _good является значением параметра по умолчанию и для istream::clear(), и для ostream::clear().

15.3.4 Инициализация потоков ввода

Естественно, тип istream, так же как и ostream, снабжен конструктором:

```
class istream {
    // ...
    istream(streambuf* s, int sk =1, ostream* t =0);
    istream(int size, char* p, int sk =1);
    istream(int fd, int sk =1, ostream* t =0);
};
```

Параметр sk задает, должны пропускаться пропуски или нет. Параметр t (необязательный) задает указатель на ostream, к которому прикреплен istream. Например, cin прикреплен к cout; это значит, что перед тем, как попытаться читать символы из своего файла, cin выполняет

```
cout.flush(); // пишет буфер вывода
```

С помощью функции istream::tie() можно прикрепить (или открепить, с помощью tie(0)) любой ostream к любому istream. Например:

```
int y_or_n(ostream& to, istream& from)
/*
    "to", получает отклик из "from"
*/
{
    ostream* old = from.tie(&to);
    for (;;) {
    cout << "наберите Y или N: ";
    char ch = 0;
```

```
 f (!cin.get(ch)) return 0;

if (ch != '\n') { // пропускает остаток строки
char ch2 = 0;
while (cin.get(ch2) && ch2 != '\n') ;
}
    switch (ch) {
    case 'Y':
    case 'y':
    case '\n':
  from.tie(old); // восстанавливает старый tie
   return 1;
   case 'N':
     case 'n':
  from.tie(old); // восстанавливает старый tie
  return 0;
     default:
    cout << "извините, попробуйте еще раз: ";
     }
  }
}
```

Когда используется буферизованный ввод (как это происходит по умолчанию), пользователь не может набрав только одну букву ожидать отклика. Система ждет появления символа новой строки. y_or_n() смотрит на первый символ строки, а остальные игнорирует.

Символ можно вернуть в поток с помощью функции istream::putback(char). Это позволяет программе «заглядывать вперед» в поток ввода.

15.4 Работа со строками

Можно осуществлять действия, подобные вводу/выводу, над символьным вектором, прикрепляя к нему istream или ostream. Например, если вектор содержит обычную строку, завершающуюся нулем, для печати слов из этого вектора можно использовать приведенный выше копирующий цикл:

```
void word_per_line(char v[], int sz)
/*
   печатет "v" размера "sz" по одному слову на строке
*/
{
   istream ist(sz,v); // сделать istream для v
   char b2[MAX];      // больше наибольшего слова
   while (ist>>b2) cout << b2 << "\n";
}
```

Завершающий нулевой символ в этом случае интерпретируется как символ конца файла. С помощью ostream можно отформатировать сообщения, которые не нужно печатать тотчас же:

```
char* p = new char[message_size];
ostream ost(message_size,p);
do_something(arguments,ost);
display(p);
```

Такая операция, как do_something, может писать в поток ost, передавать ost своим подоперациям и т.д. с помощью стандартных операций вывода. Нет необходимости делать проверку не переполнение, поскольку ost знает свою длину и когда он будет переполняться, он будет переходить в состояние _fail. И, наконец, display может писать сообщения в «настоящий» поток вывода. Этот метод может оказаться наиболее полезным, чтобы справляться с ситуациями, в которых окончательное отображение данных включает в себя нечто более сложное, чем работу с традиционным построчным устройством вывода. Например, текст из ost мог бы помещаться в располагающуюся где-то на экране область фиксированного размера.

15.5 Буферизация

При задании операций ввода/вывода мы никак не касались типов файлов, но ведь не все устройства можно рассматривать одинаково с точки зрения стратегии буферизации. Например, для ostream, подключенного к символьной строке, требуется буфери-

зация другого вида, нежели для ostream, подключенного к файлу. С этими проблемами можно справиться, задавая различные буферные типы для разных потоков в момент инициализации (обратите внимание на три конструктора класса ostream). Есть только один набор операций над этими буферными типами, поэтому в функциях ostream нет кода, их различающего. Однако функции, которые обрабатывают переполнение сверху и снизу, виртуальные. Этого достаточно, чтобы справляться с необходимой в данное время стратегией буферизации. Это также служит хорошим примером применения виртуальных функций для того, чтобы сделать возможной однородную обработку логически эквивалентных средств с различной реализацией. Описание буфера потока в выглядит так:

```
struct streambuf {      // управление буфером потока

    char* base;         // начало буфера
    char* pptr;         // следующий свободный char
    char* qptr;         // следующий заполненный char
    char* eptr;         // один из концов буфера
    char  alloc;        // буфер, выделенный с помощью new

       // Опустошает буфер:
       // Возвращает EOF при ошибке и 0 в случае успеха
    virtual int overflow(int c =EOF);

     // Заполняет буфер
    // Возвращет EOF при ошибке или конце ввода,
    // иначе следующий char
    virtual int underflow();

  int snextc()     // берет следующий char
 {
 return (++qptr==pptr)?underflow():*qptr&0377;
    }

    //
```

```
int allocate()//выделяет некоторое пространство буфера

    streambuf() { /* ... */}
    streambuf(char* p, int l) { /* ... */}
    ~streambuf() { /* ... */}
};
```

Обратите внимание, что здесь определяются указатели, необходимые для работы с буфером, поэтому обычные посимвольные действия можно определить (только один раз) в виде максимально эффективных inline-функций. Для каждой конкретной стратегии буферизации необходимо определять только функции переполнения overflow() и underflow(). Например:

```
struct filebuf : public streambuf {

    int fd;             // дескриптор файла
    char opened;        // файл открыт

    int overflow(int c =EOF);
    int underflow();

    // ...

        // Открывает файл:
        // если не срабатывает, то возвращает 0,
        // в случае успеха возвращает "this"
    filebuf* open(char *name, open_mode om);
    int close() { /* ... */ }

    filebuf() { opened = 0; }
    filebuf(int nfd) { /* ... */ }
    filebuf(int nfd, char* p, int l) : (p,l) { /* ... */ }
    ~filebuf() { close(); }
};

int filebuf::underflow()        // заполняет буфер из fd
{
```

```
    if (!opened || allocate()==EOF) return EOF;

    int count = read(fd, base, eptr-base);
    if (count < 1) return EOF;

    qptr = base;
    pptr = base + count;
    return *qptr & 0377;
}
```

15.6 Эффективность

Можно было бы ожидать, что раз ввод/вывод определен с помощью общедоступных средств языка, он будет менее эффективен, чем встроенное средство. На самом деле это не так. Для действий вроде «поместить символ в поток» используются inline-функции, единственные необходимые на этом уровне вызовы функций возникают из-за переполнения сверху и снизу. Для простых объектов (целое, строка и т.п.) требуется по одному вызову на каждый. Как выясняется, это не отличается от прочих средств ввода/вывода, работающих с объектами на этом уровне.

16 ЗАКЛЮЧЕНИЕ

Итак, нами рассмотрены основные средства языка C++, позволяющие реализовать объектно – ориентированную технологию программирования. Объектно ориентированный подход в программировании – это отнюдь не дань очередной моде. Эта технология является естественным развитием модульного прицип программирования, когда свойства и методы объекта инкапсулированы в единый агрегрегат, отраженный в типе данных.

Разумеется, составить небольшую программу – однодневку можно и без объектно – ориентированного подхода, но создать серьезный программый продукт без него, весьма трудно, если вообще возможно, по крайней мере, за приемлемые сроки.

17 ЛИТЕРАТУРА

Основная литература

1. Франка П. C++: Учебный курс: Пер. с англ. П. Бибикова. — СПб.: Питер, 2005. — 521 с

2. Павловская Т.А., Щупак Ю.А. C++. Объектно-ориентированное программирование: практикум: Учебное пособие для вузов. — СПб.: Питер, 2005. — 464 с

3. Павловская Т.А. C/C++: Программирование на языке высокого уровня: Учебник для вузов — СПб.: Питер, 2002. — 460 с

4. Боровский А.Н. Самоучитель C++ и Borland C++ Builder: самоучитель. — СПб.: Питер, 2005. — 255 с.

Дополнительная литература

1. Ричард Вайнер, Льюис Пинсон. C++ изнутри: Пер. с англ. — Киев: «ДиаСофт», 1993. — 304 с.

2. Стефан Дьюхарст, Кэти Старк. Программирование на C++: Пер. с англ. — Киев: «ДиаСофт», 1993. — 272 с.

3. Ирэ Пол. Объектно-ориентированное программирование с использованием C++: Пер. с англ. — Киев: «ДиаСофт», 1995. — 480 с.

4. Фейсон Т. Объектно-ориентированное программирование на Borland C++ 4.: Пер. с англ. — Киев: Диалектика, 1996. — 544 с.

5. Скляров В.А. Язык C++ и объектно-ориентированное программирование: Справочное издание. — Минск: Вышэйшая школа, 1997. — 480 с.

6*. Бьярн Страустрап Введение в C++. 1995
http://www.umx.org.ua/cpp/aglav.htm

7*. Патрикеев Ю.Н. Объектно-ориентированное программирование на BORLAND C++.Лекции: 1998
http://www.object.newmail.ru/obj0.html

8*. Вахтеров М., Орлов С. Четвертый BORLAND C++ и его окружение, http://www.unix.org.ua/bccp/index.htm

Примечание: символом * помечены электронные учебники.