ВВЕДЕНИЕ

Исторические замечания

Предком языка программирования C++ явился язык программирования C. Это - язык программирования общего назначения, который был создан во второй половине 70-х годов Брайаном Керниганом и Деннисом Ритчи. Первоначально он был задуман как язык системного программирования, но в дальнейшем выяснилось, что он хорошо подходит и для других задач. Основными достоинствами этого языка явились его компактность, гибкость и удобство в использовании. В настоящее время язык занимает едва ли не первое место в мире по распространенности среди инструментальных средств системного программирования. Главным достоинством является то, что в настоящее время доступно однозначное и машинно-независимое определение языка C - ANSI-стандарт этого языка. Отсюда следует, что выучив стандарт, можно писать программы на C, которые будут компилироваться любым компилятором C (поскольку все они обязаны поддерживать этот стандарт) на любой машине, где такой компилятор есть. Это свойство, которого ни в коем случае нет у Турбо Паскаля, называется *переносимостью*. Программу на C, созданную в ОС MS-DOS или Windows, можно перенести в другую ОС, например UNIX, если она использует только средства, описанные в стандарте.

Язык программирования C++ был создан Бьерном Страуструпом в начале 80-х годов. Он стал развитием языка C по трем основным направлениям :

1) поддержка абстракции данных (типов данных, определяемых пользователем);
2) поддержка объектно-ориентированного проектирования и программирования;
3) различные улучшения конструкций C.

В то же время язык сохранил все достоинства C (например, переносимость), поскольку тот вошел в него как подмножество.

Теперь язык C++ является основным языком высокого уровня для создания программного обеспечения, в первую очередь больших программных комплексов. Он реализован на самых разных ЭВМ - от персональных до суперкомпьютеров.

На компьютерах IBM PC доступно несколько реализаций C++. Мы будем опираться на 2 системы: в части начального курса на Turbo C++ (версии 3.0 и выше), а в части системного программирования на систему Microsoft Visual C++ (версия 6.0 и выше).

Система Turbo C++ выбрана не случайно. Она очень похожа на изученную ранее среду Turbo Pascal, а значит, большая часть усилий на начальном этапе будет тратиться на изучение конструкций языка, а не приемов работы со средой.

Еще одним импульсом к изучению вами C++ должно послужить то, что практически всё современное системное программное обеспечение, в том числе и операционные системы семейства Windows и различные версии UNIX, написаны на этом языке (правда, включают в себя фрагменты кода на Ассемблере).

Основные понятия

При написании программ в языках C и C++ используются следующие понятия:

- алфавит,
- константы,
- идентификаторы,
- ключевые слова,
- комментарии.

Алфавитом языка называется совокупность символов, используемых в языке. Очень важно знать и помнить, что язык C различает прописные и строчные буквы. Язык C, как говорят, является чувствительным к регистру (case sensitive).

Идентификаторы в языке программирования используются для обозначения имен переменных, функций и меток, применяемых в программе. Идентификатором может быть произвольная последовательность латинских букв (прописных и строчных), цифр и символа подчеркивания, которая начинается с буквы или символа подчеркивания. В языке С идентификатор может состоять из произвольного числа символов, однако два идентификатора считаются различными, если у них различаются первые 32 символа. В языке С++ это ограничение снято. В отличии от Паскаля, прописные и строчные буквы в идентификаторах различаются, т.е. различными считаются имена: Var1, var1 иVAR1.

В именах переменных можно использовать символ подчеркивания. Обычно с символа подчеркивания начинаются имена системных зарезервированных переменных и констант.

В библиотечных функциях также часто используются имена, начинающиеся с этого символа. Это делается в предположении, что пользователи вряд ли будут применять этот символ в качестве первого символа. Старайтесь не использовать имен, начинающихся с символа подчеркивания, и вам удастся избежать возможных конфликтов и взаимопересечений с множеством библиотечных имен.

В языках С и С++ некоторые идентификаторы употребляются как ***служебные слова*** (keywords), которые имеют специальное значение для компилятора. Их употребление строго определено, и эти слова не могут использоваться иначе. Ключевыми словами стандарта ANSI языка С являются:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Каждый компилятор может увеличивать количество ключевых слов, так как компилятор учитывает дополнительные возможности того типа компьютеров, для которых он создан. Например, компилятор Borland C++ 3.1 добавляет к ключевым словам стандарта языка С дополнительные слова, предназначенные для работы с памятью и регистрами процессоров семейства Intel, а также позволяющих использовать прерывания и фрагменты программ на другом языке. Эти дополнительные ключевые слова приведены ниже:

asm	_asm	__asm	cdecl
_cdecl	__cdecl	_cs	__cs
_ds	__ds	_es	__es
_export	__export	far	_far
__far	_fastcall	__fastcall	huge
_huge	__huge	interrupt	_interrupt
__interrupt	_loadds	__loadds	near
_near	__near	pascal	_pascal
__pascal	_saveregs	__saveregs	_seg
__seg	__ss	__ss	

Язык C++ в дополнение к ключевым словам языка С добавляет еще несколько:

asm	catch	class	friend	inline	new
operator	private	pro-tected	public	template	this
throw	try	virtual			

Часть символов язык С рассматривает как пробельные символы. Это не только символ пробела ' ', но и символы табуляции, символы перевода строки (новой строки), возврата каретки, символ перевода страницы (новой страницы).

Комментарий - это часть программы, которая игнорируется компилятором и служит для удобочитаемости исходного текста программы. В процессе компиляции комментарий заменяется пробелом, следовательно, комментарий может располагаться в любом месте программы, где допустимо использование пробела. Комментарием в языке С является любая последовательность символов, заключенная между парами символов /* и */. В стандарте языка С запрещены вложенные комментарии, хотя во многих реализациях компиляторов, в частности в Borland C++, вложенные комментарии разрешены. В языке С++ появился еще один вид комментариев: так называемый однострочный комментарий. Все символы, располагающиеся за парой символов // и до конца строки, рассматриваются как комментарий. Компилятор языка С, встроенный в систему Borland C++, позволяет использовать комментарий в стиле С++ в программах на языке С.

Рассмотрим несколько простейших примеров программ на языке С, а затем вернемся к систематическому изучению языка.

Две простые программы

Вызовем систему Turbo C++ и введем в режиме редактирования следующую программу:

Пример 1.

```
# include <stdio.h>
 /* Пример 1.*/
main()
{
int year, month;
year=2004;
printf("Сейчас %d год\n", year);
}
```

Детально рассмотрим нашу первую программу.

Первая строка:

#include <stdio.h>

Она сообщает компилятору о необходимости подключить файл **stdio.h**. Этот файл содержит информацию, необходимую для правильного выполнения функций библиотеки стандартного ввода/вывода языка С. Язык С предусматривает использование некоторого числа файлов такого типа, которые называются заголовочными файлами (header files). В файле stdio.h находится информация о стандартной функции вывода **printf()**, которую мы использовали.

Вторая строка:

/* Пример 1.*/

является комментарием.

При внимательном рассмотрении программы заметим, что между 5-й и 6-й строками находится пустая строка. Пустые строки в языке С не оказывают никакого влияния и могут быть вставлены для удобочитаемости программы.

Строка

main()

определяет имя функции. Любая программа на языке С состоит из одной или нескольких функций. Выполнение программы начинается с вызова функции **main()**. Поэтому каждая программа на языке С должна содержать функцию **main().** Эта конструкция может выглядеть по-другому:

void main (void)

или

void main()

Эти 3 конструкции взаимозаменяемы. Для простоты пока поясним , что void означает нечто не существующее и его можно явно не указывать. Все 3 заголовка говорят о том, что наша главная функция не имеет параметров и не возвращает никакого значения.

Следующая строка

{

содержит открывающую фигурную скобку (brace), обозначающую начало тела функции **main().** Фигурные скобки в языке С всегда используются парами (откры-

вающая и закрывающая). Закрывающую скобку мы еще встретим в нашей программе.

Строка

int year, month ;

объявляет (declare) переменную, называемую **year**, и сообщает компилятору, что эта переменная целая. ***В языке С все переменные должны быть объявлены прежде, чем они будут использованы***. Процесс объявления переменных включает в себя определение имени (идентификатора) переменных **(year, month)** и указание типа переменных **(int)**.

Строка

year=2004;

является оператором присваивания. В этой строке переменой с именем year присваивается значение 2004. Заметим, что в языке С используется просто знак равенства в операторе присваивания. Все операторы в языке С заканчиваются символом «точка с запятой».

Строка

printf ("Сейчас %d год\n", year);

является вызовом стандартной функции **printf()**, которая выводит на экран некоторую информацию. Эта строка состоит из двух частей: имени функции **printf()** и двух ее аргументов **"Сейчас %d год\n"** и **year**, разделенных запятой. В языке С нет встроенных функций ввода/вывода. Но библиотеки языка С и С++ содержат много полезных и удобных функций ввода/вывода. Функция **printf()**, которую мы использовали, является универсальной функцией форматного вывода.

Для вызова функции нужно написать имя функции и в скобках указать необходимые фактические аргументы. Первый аргумент функции **printf()** - это строка в кавычках **"Сейчас %d год\n"**, которую иногда называют управляющей строкой (control string). Эта строка может содержать любые символы или спецификации формата, начинающиеся с символа **'%'**. Обычные символы просто отображаются на экран в том порядке, в котором они следуют.

Спецификация формата, начинающаяся с символа **'%**', указывает формат, в котором будет выводиться значение переменной **year**, являющейся вторым аргументом функции **printf().** Спецификация **%d** указывает, что будет выводиться целое число в десятичной записи. Комбинация символов **' \n'** сообщает функции **printf()** о необходимости перехода на новую строку. Этот символ называется символом новой строки (newline).

Последняя строка программы:

}

содержит закрывающую фигурную скобку. Она обозначает конец функции **main().**

Если при наборе программы вы не допустили опечаток, то на экране увидите строку:

Сейчас 2004 год

Попробуем намеренно ввести ошибку в нашу программу. Например, не поставив одну из точек с запятой. Попробуем еще раз выполнить уже неправильную программу. Нажмем комбинацию клавиш **Ctrl-F9**. При компиляции программы будет обнаружена ошибка, которая будет подсвечена в окне сообщений (message window). Причем в окне сообщений ошибка будет подсвечена более яркой строкой, а в окне редактирования курсор устанавливается в том месте программы, где компилятор системы Turbo C++ обнаружил ошибку. На самом деле курсор будет находиться в следующей строке, что бывает всегда, когда компилятор не находит точки с запятой.

Одним из наибольших удобств интегрированной среды является то, что в интерактивном режиме пользователь может обнаружить и исправить ошибку. Для перехода к следующей ошибке нужно набрать комбинацию клавиш **ALT-F8**.

Нажав комбинацию клавиш **ALT-F7**, можно перейти к предыдущей ошибке. Нажатием клавиши **Enter** можно активизировать окно редактирования и исправить выделенную курсором ошибку.

Наличие ошибок (errors) не позволяет выполнить программу. Необходимо исправить найденные ошибки и снова компилировать программу. Однако даже если в программе нет синтаксических ошибок, некоторые ситуации могут вызвать подоз-

рение у компилятора. Когда Turbo C++ встречается с одной из таких ситуаций, то печатается предупреждение (warning). Пользователь должен проанализировать указанную ситуацию и принять соответствующее решение.

Например, была объявлена переменная **month**, но она не была использована в программе. Предупреждение об этом должно быть сделано системой Turbo C++.

Эта программа была написана более в стиле языка С, а не С++. Приведем пример программы с использованием конструкций ввода/вывода языка С++.

Пример 2.

```
//Вычисление длины окружности
#include <iostream.h>
void main()
{
int radius:
float length:
cout<<"Введите значение радиуса:\n";
cin>>radius;
length = 3.1415 * 2 * radius;
cout<<" Радиус- “<< radius<<”длина – “<<length;
}
```

В этой программе по сравнению с предыдущей использовано несколько важных новшеств.

Во-первых, объявлены две переменные двух разных типов: **radius** - типа целое (**int**); **length** - типа с плавающей запятой (**float**), содержащую дробную часть.

Во-вторых, используются другие операторы ввода/вывода даже на первый взгляд значительно отличающиеся от функции printf.

Стадии прохождения программы

Объединенная единым алгоритмом совокупность описаний и операторов образует *программу на алгоритмическом языке.* При работе с языком С++ текст программы должен быть подготовлен в файле с расширением **.CPP**.

Для того чтобы выполнить программу, необходимо ее перевести на язык, понятный процессору - в машинные коды. Этот процесс состоит из нескольких этапов. Рис. 1 иллюстрирует эти этапы для языка С++.

Сначала программа передается *препроцессору*, который выполняет *директивы*, содержащиеся в ее тексте (например, #include - включение файла в текст программы, см. примеры 1, 2, 3).

Получившийся текст передается на вход *компилятора*, который выделяет лексемы (отдельные слова), а затем на основе грамматики языка распознает выражения и операторы, построенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и в случае их отсутствия строит *объектный модуль.*

Компоновщик, или *редактор связей*, формирует *исполняемый модуль* программы, подключая к объектному модулю другие объектные модули, в том числе содержащие функции библиотек, обращение к которым содержится в любой программе. Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. Исполняемый модуль имеет расширение **.exe** и запускается на выполнение обычным путем.

Немного о функциях языка С (С++)

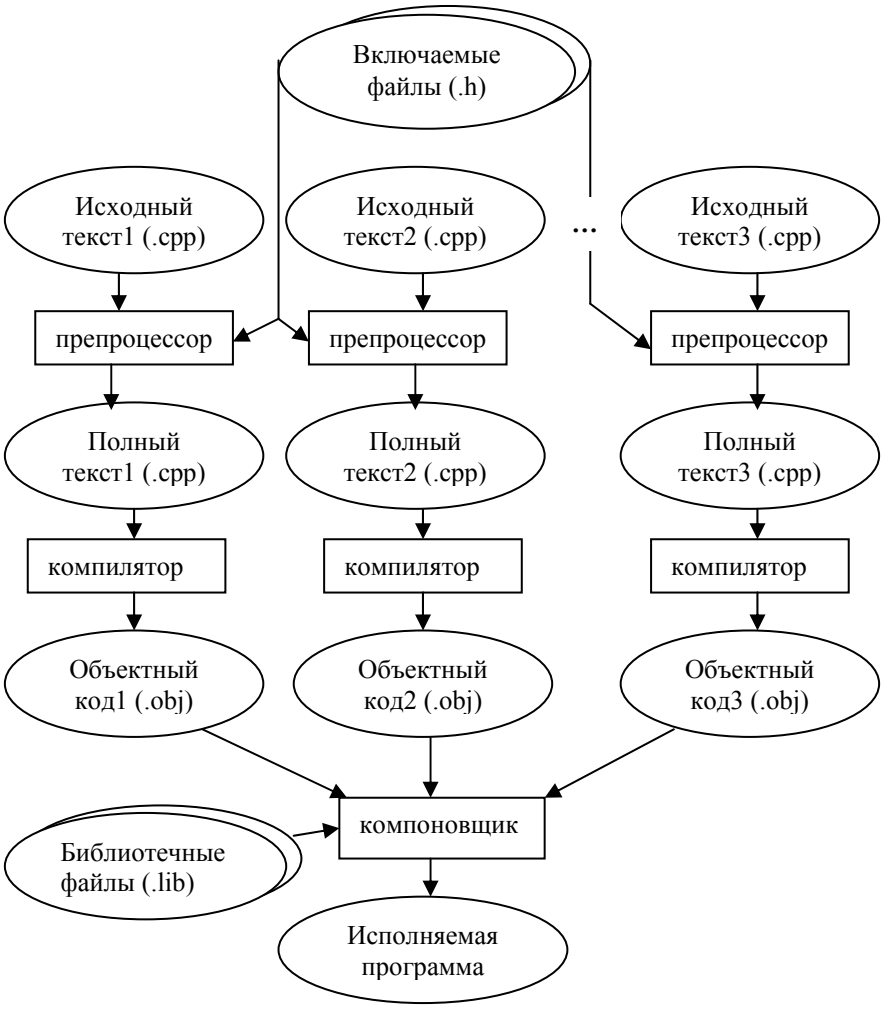


Рис. 1. Этапы создания исполняемой программы

Принципы программирования на языке С основаны на понятии функции. ***Функция*** - это самостоятельная единица программы, созданная для решения конкретной задачи. Функция в языке С играет ту же роль, что и подпрограммы или процедуры в других языках.

Программы на языке С обычно состоят из большого числа функций. Как правило, эти функции имеют небольшие размеры и могут находиться как в одном, так и в нескольких файлах. Все функции являются глобальными. В языке С запрещено объявлять одну функцию внутри другой. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние (глобальные) переменные.

Каждая функция языка С имеет имя и список аргументов. По соглашению, принятому в языке С, при записи имени функции после него ставят круглые скобки. Это соглашение позволяет легко отличать имена переменных от имен функций.

Рассмотрим пример программы, в которой кроме функции main() содержатся еще две функции.

Пример 4.

```
#include <stdio.h>
/* Пример программы с функциями */
function1(); /* прототип первой функции     */
function2(); /* прототип второй функции     */
main()       /* Главная функция           */
{       /* Начало тела главной функции    */
function1(); /* вызов первой функции      */
function2(); /* вызов второй функции      */
}       /* Конец тела функции main()      */
function1() /* Начало определения первой функции */
{       /* Начало тела первой функции     */
    printf("вызвали первую функцию\n");
}     /* Конец тела первой функции       */
function2() /* Начало определения второй функции */
{       /* Начало тела второй функции      */
    printf("вызвали вторую функию\n");
}       /* Конец тела второй функции       */
```

Здесь 3 функции: **main()**, **function1()**, **function2()**. Эти функции не имеют аргументов.

Общий вид программы

В общем виде программа на С есть, как мы видели, набор функций, предваряемый инструкциями препроцессору. Выглядит программа в общем случае следующим образом:

```
#инструкция препроцессору 1
#инструкция препроцессору 2
. . . . .
описание функции 1;
описание функции 2;
. . . .
описания глобальных переменных
функция n(…)
{
описания локальных переменных и констант;
оператор 1;
оператор 2;
…
}
функция m(…)
{
описания локальных переменных и констант;
оператор 1;
оператор 2;
…

}
. . . .
```

Некоторые из приведенных разделов могут быть опущены или переставлены местами, язык не регламентирует порядок разделов.

Примечания.

В языке С точка с запятой обозначает конец оператора. Таким образом, каждый отдельный оператор должен заканчиваться точкой с запятой.

Составной оператор **(блок)** - это набор логически связанных операторов, находящихся между открывающей и закрывающей фигурными скобками (оператор-

ными скобками). Если вы рассматриваете блок как набор операторов, то за блоком точка с запятой не ставится.

Конец строки в С не является окончанием оператора. Это означает, что нет ограничений на расположение операторов в программе и их можно располагать так, чтобы программу было удобно читать. Нельзя разрывать идентификаторы. Зато в операторах там, где можно поставить пробел, можно и перенести оператор на другую строку. Два фрагмента программ, представленные ниже, эквивалентны:

```
а) x = y;   y = y + 1;   mul      б)   x = y; y = y + 1;
(x, y);                           mul (x, y );
```

ОПИСАНИЕ ЯЗЫКА

Операторы языка С манипулируют переменными и константами в виде выражений. Имена, которые используются для обозначения переменных, функций, меток и других определенных пользователем объектов, называются ***идентификаторами*** (identifiers). Идентификатор может содержать латинские буквы, цифры и символ подчеркивания, и начинаться обязан с буквы или символа подчеркивания. Идентификатор не должен совпадать с ключевыми словами.

Базовые типы данных

В языке С все переменные должны быть объявлены до их использования. В нем определены 5 типов данных, которые можно назвать базовыми:

char - символьные,

int - целые,

float - с плавающей точкой,

double - с плавающей точкой двойной длины,

void - пустой, не имеющий значения.

Типы **char** и **int** являются целыми типами и предназначены для хранения целых чисел, хотя название типа **char** - символьная переменная. Любой символ в компьютере связан с целым числом – кодом этого символа, например в таблице ASCII. Сам же символ необходим, когда информация выводится на экран или на принтер или, наоборот, вводится с клавиатуры. Подобные преобразования символа в код и наоборот производятся автоматически. Обычно тип **char** представлен байтом (8 разрядами). Размеры всех типов в языке кратны размеру **char**. Тип **char** по умолчанию является знаковым типом, однако настройкой опций интегрированной среды можно установить по умолчанию беззнаковый тип **char**. Тип **int** всегда знаковый, так же как и типы **float** и **double**.

Размер типа **int** не определяется стандартом, а зависит от компьютера и компилятора. Для компилятора системы Turbo C++ под величины этого типа отводится 2 байта, в системе Visual C++ - 4 байта.

Переменные типа **double** и **float** являются числами с плавающей точкой.

Ключевое слово **void** было привнесено в стандарт ANSI C из языка C++. Нельзя создать переменную типа **void**. Однако введение этого типа оказалось весьма удачным.

На основе этих пяти типов строятся дальнейшие типы данных.

Простейшим приемом является использование модификаторов (modifiers) типа, которые ставятся перед соответствующим типом.

В стандарте ANSI языка C такими модификаторами являются следующие зарезервированные слова:

signed - знаковый,

unsigned - беззнаковый,

long - длинный,

short - короткий.

Модификаторы **signed** и **unsigned** могут применяться к типам **char** и **int**. Модификаторы **short** и **long** могут применяться к типу **int**. Модификатор **long** может применяться также к типу **double**. Модификаторы **signed** и **unsigned** могут комбинироваться с модификаторами **short** и **long** в применении к типу **int**.

В следующей таблице приведены все возможные типы с различными комбинациями модификаторов, использующиеся в языке C. Размер в байтах и интервал изменения могут варьироваться в зависимости от компилятора, процессора и операционной системы (среды). В таблице приведены значения для 2-байтового типа **int**

Основные типы данных

Таблица 1

Тип	*Размер в байтах (битах)*	*Интервал изменения*	
char	1 (8)	от -128	до 127
unsigned char	1 (8)	от 0	до 255
signed char	1 (8)	от -128	до 127
int	2 (16) или	от -32768 (-	до 32767

	4(32)	2^{15})	(2^{15}-1)
unsigned int	2 (16)	от 0	до 65535
signed int	2 (16)	от -32768	до 32767
short int	2 (16)	от -32768	до 32767
unsigned short int	2 (16)	от 0	до 65535
signed short int	2 (16)	от -32768	до 32767
long int	4 (32)	от - 2147483648	до 2147483647
signed long int	4 (32)	от - 2147483648	до 2147483647
unsigned long int	4 (32)	от 0	до 4294967295
float	4 (32)	от 3.4E-38	до 3.4E+38
double	8 (64)	от 1.7E-308	до 1.7E+308
long double	10 (64)	от 3.4E-4932	до 3.4E+4932

Запись 3.4E-38 соответствует числу $3.4 \cdot 10^{-38}$, это так называемый научный формат записи числа с плавающей запятой.

Различие между целыми числами со знаком и целыми числами без знака состоит в том, как интерпретируется старший бит целого числа. Старший бит для целого числа со знаком определяет знак числа. Если старший бит равен нулю, - число положительное, если же старший бит равен единице, то число отрицательное. Целое число +3 типа int будет храниться в памяти компьютера в виде

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Если объявлено отрицательное целое число, то компилятор генерирует так называемый дополнительный код. Чтобы получить число -3 надо поменять значения

всех битов на обратные, т. е. 0 заменить на 1, 1 заменить на 0 и прибавить к младшему биту 1. Число -3 в двоичной записи в дополнительном коде будет иметь вид:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1

При внимательном рассмотрении табл. 1 можно заметить, что типы **int**, **short int**, **signed int** и **signed short int** имеют одни и те же пределы изменения. Эти типы, хотя и имеют разные названия, являются совершенно одинаковыми в данной реализации компилятора языка. Типы **int** и **signed int** всегда обозначают один и тот же тип, так как тип **int** всегда знаковый и без модификатора **signed**. Модификатор **short** в данной реализации не оказывает никакого воздействия на тип, поэтому в программах, написанных для компиляторов фирмы Borland, вы не встретите этого ключевого слова.

В старом стандарте Керниган&Ритчи тип **int** являлся настолько важным, что его можно было иногда опускать. В частности, в типах **signed int**, **unsigned int, long int, unsigned long int** ключевое слово **int** можно опустить и писать просто **signed, unsigned, long, unsigned long**.

Для того чтобы понять различие в интерпретации компилятором целых чисел со знаком и целых чисел без знака, рассмотрим следующий пример:

Пример 6.

```
#include <stdio.h>
main()
{
  int i;
  unsigned int j;
  j = 60000;
  i =j;
  printf("i=%d j=%u\n", i, j);
}
```

Результатом работы программы будет строка

i = -5536 j= 60000

Спецификация **%d** сообщает функции **printf()**, что используется целое число со знаком; спецификация **%u** - другая спецификация формата, которая соответствует беззнаковому целому.

Дополнительные типы данных

Стандарт ANSI C кроме уже перечисленных типов данных поддерживает еще несколько дополнительных типов (но их нет в среде Turbo C++).

Расширенный символьный тип (wchar_t)

Предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например, Unicode. Размер этого типа зависит от реализации, как правило, он соответствует типу **short**. Строковые константы типа **wchar_t** записываются с префиксом **L**, например, L"C++".

Логический тип (bool)

Величины логического типа могут принимать только значения **true** и **false**, являющиеся зарезервированными словами. Внутренняя форма представления значения **false** - 0 (нуль). Любое другое значение интерпретируется как **true**. При преобразовании к целому типу **true** имеет значение 1.

Примечание

В таких средах, как Turbo C++, где нет типа **bool** именно так и трактуется истинность или ложь некоторого логического выражения: за ложь принимается нулевое значение, а любое другое значение считается истинным.

Объявление переменных

Одним из основных понятий языка C++ является унаследованное из языка C и предшествующих языков понятие объекта как некоторой области памяти. Отличительной чертой переменной является возможность связывать с ее именем различные значения, совокупность которых определяется типом переменной. При определении значения переменной в соответствующую ей область памяти помещается некоторый код. Это может происходить либо во время компиляции, либо во время исполнения

программы. В первом случае говорят об *инициализации,* во втором - о присваивании. Операция присваивания **E=B** содержит имя переменной (**E**) и некоторое выражение (**B**). Имя переменной есть частный случай более общего понятия - «леводопустимое выражение» (l-value). Это название произошло как раз от изображения операции присваивания, т.к. только леводопустимое выражение может быть использовано в качестве ее левого операнда. L-значение определяет в общем случае ссылку на некоторый объект. Частным случаем такой ссылки является переменная.

Итак, объект определяется как некоторая область памяти. Это понятие вводится как понятие времени исполнения программы, а не понятие языка C++. В языке C++ термин объект зарезервирован как термин объектно-ориентированного подхода к программированию. При этом объект всегда принадлежит некоторому классу, и такие объекты рассматривает область объектно-ориентированного программирования.

Имя объекта как частный случай леводопустимого выражения обеспечивает как получение значения объекта, так и изменение этого значения. Не любые выражения являются леводопустимыми. К леводопустимым относятся:

- имена скалярных арифметических и символьных переменных;
- имена переменных, принадлежащих массивам (индексированные переменные);
- имена указателей;
- уточненные имена компонентов структурированных данных (структур, объединений, классов):
- леводопустимые выражения, заключенные в круглые скобки;
- ссылки на объекты;
- выражения с операцией разыменования « *»'.

Кроме леводопустимых выражений, определены праводопустимые выражения, которые невозможно использовать в левой части оператора присваивания. Например:

- имя функции;
- имя массива;
- имя константы;
- вызов функции.

Так как с объектом (переменной) связано некоторое значение, то для переменной необходимо задать тип, который:

- определяет требуемое для объекта количество памяти при ее начальном распределении;
- задает совокупность операций, допустимых для объекта;
- интерпретирует двоичные коды значений при последующих обращениях к объекту;
- используется для контроля типов с целью обнаружения возможных случаев недопустимого присваивания.

Кроме типов, для переменных явно либо по умолчанию определяются:

- класс памяти;
- область действия;
- область видимости;
- продолжительность существования;

Основная форма объявления переменных имеет вид:

[класс памяти] тип <список переменных> ;

Примечание.

Наличие квадратных скобок указывает на необязательность заключенной в них информации.

Здесь:

тип - один из существующих в С типов переменных;

<***список переменных***>- может состоять из одной или нескольких переменных, разделенных запятыми, которым могут быть присвоены некоторе значения, имеет вид:

имя_перем1[=значение1], имя_перем2[=значение2], имя_перем3,…;

класс памяти (необязателен) - определяет размещение объекта в памяти и продолжительность его существования.

Примеры объявлений (описаний) переменных:

```
int x=0, y, z=1;
float radius;
unsigned char ch='q';
long double integral;
static long LL;
```

Классы памяти

Класс *памяти* определяет размещение объекта в памяти и продолжительность его существования. Для явного задания класса памяти при определении (описании) объекта используются или подразумеваются по умолчанию следующие спецификаторы:

auto (автоматически выделяемая, локальная память)

Спецификатор **auto** может быть задан только при определении объектов блока, например, в теле функции. Этим объектам память выделяется при входе в блок и освобождается при выходе из него. Вне блока объекты класса **auto** не существуют. для глобальных переменных этот спецификатор не используется, а для локальных применяется по умолчанию, поэтому задавать его явным образом смысла нет.

register (автоматически выделяемая, по возможности регистровая память)

Спецификатор **register** аналогичен **auto,** но для размещения значений объектов используются регистры, а не участки основной памяти. Такая возможность имеется не всегда, и в случае отсутствия регистровой памяти (если регистры заняты

другими данными) объекты класса **register** компилятор обрабатывает как объекты автоматической памяти.

static (внутренний тип компоновки и статическая продолжительность существования). Объект, описанный со спецификатором **static,** будет существовать только в пределах того файла с исходным текстом программы (модуля), где он определен. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. Класс памяти **static** может приписываться переменным и функциям.

extern (внешний тип компоновки и статическая продолжительность существования)

Этот спецификатор означает, что переменная определена в другом месте программы (в другом файле или дальше по тексту). Объект класса **extern** глобален, т.е. доступен во всех модулях (файлах) программы. Класс **extern** может быть приписан переменной или функции.

Кроме явных спецификаторов, на выбор класса памяти существенное влияние оказывают размещение определения и описаний объекта в тексте программы. Такими определяющими частями программы являются блок, функция, файл с текстом кода программы (модуль) и т.д. Таким образом, класс памяти, т.е. размещение объекта (в регистре, стеке, в динамически распределяемой памяти, в сегменте) зависит как от синтаксиса определения, так и от размещения определения в программе.

В языке С могут быть три места, где переменная может быть объявлена:

- ***во-первых***, вне каких-либо функций, в том числе и main(). Такая переменная называется ***глобальной*** (global) и может использоваться в любом месте программы (за исключением глобальных статических переменных, о которых мы поговорим чуть позже);
- ***во-вторых***, переменная может быть объявлена внутри блока, в том числе внутри тела функции. Объявленная таким образом переменная называется ***локальной*** (local) и может использоваться только внутри этого блока. Такая переменная неизвестна вне этого блока;

- ***в-третьих***, переменная может быть объявлена как ***формальный параметр*** функции. Кроме специального назначения этой переменной для передачи информации в эту функцию и места ее объявления, переменная может рассматриваться как локальная переменная для данной функции.

Рассмотрим пример объявления переменных в разных местах программы:

Пример 7.

```
# include <conio.h>
# include <stdio.h>
/* пример объявления переменных */
char ch;     /* глобальная переменная ch */
void print_str (int m);
main()
{
  int n;     /* локальная переменная n */
  printf ("Введите символ:");
  ch=getche();    /* использование глобальной переменной */
  printf ("Введите количество символов в строке:");
  scanf ("%d", &n);
     print_str(n);
  }
print_str (int m) /* формальный параметр m */
{
   for ( int j=0; j<m ; j++)
     printf ("%c\n", ch); /*использование глобальной переменной ch */
}
```

Эта программа иллюстрирует использование глобальных и локальных переменных, а также формальных параметров.

Область (сфера) действия идентификатора (имени) - это часть программы, в которой идентификатор может быть использован для доступа к связанному с ним объекту. Область действия зависит от того, где и как определены объекты и описаны идентификаторы. Здесь имеются следующие возможности: блок, функция, прототип функции, файл (модуль) и класс.

Если идентификатор описан (определен) в блоке, то область его действия - от точки описания до конца блока, включая все вложенные блоки, такая переменная называется *локальной* (как уже говорилось). Когда блок является телом функции, то в нем определены не только описанные в нем объекты, но и указанные в заголовке функции формальные параметры. Таким образом, сфера действия формальных параметров в определении функции есть тело функции. Если переменная объявлена вне любого блока, она называется *глобальной* и областью ее действия считается файл, в котором она определена, от точки описания до конца.

Например, областью действия переменной ch является весь файл с программой, а переменной j - только строка печати printf ("%c\n", ch) (пример 7).

Имя переменной должно быть уникально в своей области действия.

Понятие *видимость объекта* понадобилось в связи с возможностью повторных определений идентификатора внутри вложенных блоков (или функций). В этом случае разрывается исходная связь имени с объектом, который становится «невидимым» из блока, хотя сфера действия имени сохраняется. Достаточно часто сфера (область) действия идентификатора и видимость связанного с ним объекта совпадают. Область действия может превышать видимость, но обратное невозможно. К глобальной переменной из вложенного блока, имеющего переменную с тем же именем, можно обратиться, используя операцию доступа к области видимости ::.

Пример.

```
/*часть программы, демонстрирующая область видимости некоторых переменных */
int x;              // глобальное x

void f()
{
```

```
    int x;          // локальное x скрывает глобальное x
    x = 1;          // присвоить локальному x
    {
        int x;      // скрывает первое локальное x
        x = 2;      // присвоить второму локальному x
    }
    x = 3;          // присвоить первому локальному x
}

int* p = &x;        // взять адрес глобального x
```

Продолжительность существования объектов определяет период, в течение которого идентификаторам в программе соответствуют конкретные объекты в памяти. Определены 3 вида продолжительности: статическая, локальная и динамическая.

Объектам со статической продолжительностью существования память выделяется в начале выполнения программы и сохраняется до конца обработки. Статическую продолжительность имеют все функции и файлы. Остальным объектам статическая продолжительность существования может быть задана с помощью явных спецификаторов класса памяти **static** и **extern**. при статической продолжительности существования объект не обязан быть глобальным. При отсутствии явной инициализации объекты со статической продолжительностью существования по умолчанию инициализируются нулевыми, или пустыми значениями.

Автоматические переменные имеют локальную продолжительность существования: они создаются при каждом входе в блок (или функцию), где описаны, и уничтожаются при выходе.

Объекты с динамической продолжительностью существования создаются (получают память) и уничтожаются с помощью явных операторов в процессе выполнения программы (например, операторов **new** и **delete**).

Примечание.

Объявление переменной может быть выполнено в форме *описания* (иногда употребляют термин «объявление») и определения. Описание информирует компилятор о типе переменной и классе памяти, а *определение* содержит кроме этого указание компилятору выделить память в соответствии с типом переменной.

Константы в языке С

Константы представляют фиксированную величину, которая не может быть изменена в программе. Константы могут быть любого базового типа данных. Примеры констант:

Тип данных	*Константа*
char	'a', '\n', '9'
int	123, -346
unsigned int	60000
long int	75000, -27, 5L
short int	10, 12, -128
float	123.23, 4.34E-3, 4E+5
double	123.23, 12312311, -0.987

К какому типу относится константа 13 - к типу char, int, unsigned или к другому? Для языка С это почти не играет никакой роли. В то же время для языка С++ с его жесткой проверкой типов параметров функций это может сыграть очень большую роль.

Правила определения типа констант следующие.

Целая константа (т. е. константа, не имеющая десятичной точки или порядка) относится к типу int, если эта константа входит в интервал значений типа int.

Если эта константа не входит в интервал значений типа int, например 37000, то она считается константой типа unsigned. Если же константа не входит в интервал изменения unsigned, она считается константой типа long.

Константа с десятичной точкой считается константой типа double, если она помещается в соответствующий интервал изменения.

В языке С имеется механизм явного задания типов констант с помощью суффиксов. В качестве суффиксов целочисленных констант могут использоваться буквы u, l, h, U, L, H. Для чисел с плавающей запятой - l, L, f и F.

Например:

l2h 34H	short int
23L -237l	long int
891u 89Lu 89ul 7UL	unsigned long
45uh	unsigned short
23.4f 67.7E-24F	float
1.39*l* 12.0L2e+10	long double

Так как в программировании важную роль играют восьмеричные и шестнадцатеричные системы счисления, важно уметь использовать восьмеричные (octal) и шестнадцатеричные (hexadecimal) константы. Для того чтобы отличать шестнадцатеричные константы, перед ними ставится пара символов 0x. Восьмеричные константы всегда начинаются с нуля. Шестнадцатеричные и восьмеричные константы могут быть только беззнаковыми.

Например:

Шестнадцатеричные константы	*Восьмеричные константы*
0xFFFF	01
0x10	055
0x1F1A	07777

В качестве цифр шестнадцатеричной константы нам надо использовать 16 символов - 0. 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

В дальнейшем нам понадобится двоичная запись шестнадцатеричных чисел, поэтому мы приведем двоичную запись шестнадцатеричных цифр, из которых легко сложить шестнадцатеричные числа.

Таблица 2.2

Десятичное число	*Шестнадцатеричное число*	*Двоичная запись числа*	*Восьмеричная константа*	*Шестнадцатеричная константа*
0	0	0000	0	0x0
1	1	0001	01	0x1
2	2	0010	02	0x2
3	3	0011	03	0x3
4	4	0100	04	0x4
5	5	0101	05	0x5
6	6	0110	06	0x6
7	7	0111	07	0x7
8	8	1000	10	0x8
9	9	1001	11	0x9
10	A	1010	12	OxA
11	B	1011	13	OxB
12	C	1100	14	OxC
13	D	1101	15	OxD
14	E	1110	16	OxE
15	F	1111	17	OxF

Для представления в двоичном виде шестнадцатеричного числа надо просто заменить двоичной записью каждую цифру этого числа. Например, число 0xAB01 представимо в виде

A B 0 1

1010 1011 0000 0001

Пробелы, конечно, надо опустить: 1010101100000001.

Строковые константы (strings) также играют в языке С важную роль. Строковая константа или просто строка представляет собой набор символов, заключенный в двойные кавычки. Например, строковая константа "Это строка". Особенностью представления строковых констант в языке С является то, что в памяти компьютера отводится на 1 байт больше, чем требуется для размещения всех символов строки. Этот последний байт заполняется нулевым значением, т. е. байтом в двоичной записи которого одни нули. Этот символ так и называется - нулевой байт и имеет специальное обозначение '\0', В английском языке слово *строка* имеет два перевода - "line" и "string". Под термином "line" понимается строка в текстовом редакторе; под термином "string" - строковая константа.

Нельзя путать строковые константы с символьными константами. Так "a" - это строковая константа, содержащая одну букву, в то время как 'a' -символьная константа, или просто символ. Отличие "a" от 'a' в том, что строка "a" содержит еще один символ '\0' в конце строки и, таким образом, занимает в памяти 2 байта, в то время как 'a' - только 1 байт.

В языке С есть символьные константы, которые не соответствуют никакому из печатных символов. Так, в коде ASCII символы с номерами от нуля до 31 являются управляющими символами, которые нельзя ввести с клавиатуры. Для использования таких символов в языке С вводятся так называемые управляющие константы (backslash charchter constans). Мы с ними уже встречались выше в функции printf(). Фрагменты программы а) и б) эквивалентны по своему действию:

```
а)   ch='\n';                б)   printf ("\n");
     printf("%c", ch);
```

и вызывают перевод строки.

Управляющие символы представлены в следующей табл 2.3.

Управляющие символы

Таблица 2.3

Управляю-щий символ	***Значение***
\b	BS, забой
\f	Новая страница, перевод страницы
\n	Новая строка, перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\"	Двойная кавычка
\’	Апостроф
\\	Обратная косая черта
\0	Нулевой символ, нулевой байт
\a	Сигнал
\N	Восьмеричная константа
\xN	Шестнадцатеричная константа
\?	Знак вопроса

Если за символом обратной косой черты следует символ не из этой таблицы, то эта пара воспринимается просто как соответствующий символ.

Другой способ представления таких констант, и печатных символов в том числе, состоит в том, что указывается после обратной косой черты номер в кодировке ASCII в восьмеричной или шестнадцатеричной системе счисления. Операторы

```
printf("\a");
printf("\07");
printf("\7");
printf("\0x7");
```

вызовут один и тот же эффект - машина издаст звуковой сигнал.

Спецификатор const

Часто в программах встречаются константы, которым имеет смысл дать имя. Например, мы можем знать, что в нашей программе максимальная длина считываемой строки равна 512 байт. Если мы везде в программе будем на месте этой длины писать 512, т.е., например,

```
if (i < 512)
```

то встают две проблемы :

- что обозначает это 512 - знает только программист (и то может через полгода забыть) - это так называемое "магическое число" и
- если мы напишем программу, где такой цикл встречается часто, а потом вдруг выяснится, что наша длина строки не 512, а, скажем, 1024 байта, то все вхождения этих 512 в тексте программы нужно будет изменить на 1024.

Решение этих проблем состоит в использовании идентификатора, инициализированного значением 512. При этом он должен продолжать оставаться константой, т.е. его нельзя изменить в программе. Такие константы определяются с помощью модификатора const.

Например в нашем случае

```
const int maxlen = 512;
```

и, соответственно,

```
if (i < maxlen)
```

Теперь ясно, что такое maxlen, и в случае изменения величины maxlen нужно изменить только одну строку в программе. Поскольку такая константа (ее называют именованной или символической константой) не может быть изменена после своего определения, то она обязательно должна быть инициализирована.

Такая запись будет ошибкой

```
const int maxlen; // нет инициализации
```

и такая - тоже (само собой)

```
maxlen = 1024;      // это константа!
```

Дальше мы еще не раз будем сталкиваться со спецификатором const.

Здесь в первый раз мы сталкиваемся с разделом, где раньше широко применялся препроцессор, а теперь его применение не рекомендуется.

Дело в том, что раньше спецификатора const в языке С не было. Он появился с появлением стандарта ANSI и оттуда перешел в С++. В то же время препроцессор позволяет определять текстовую подстановку с помощью директивы #define.

Например, если мы напишем

#define MAXLEN 512

(слева - имя, справа - текст, без точки с запятой в конце), то все вхождения имени MAXLEN в тексте программы перед компиляцией заменятся на 512 (и вообще на текст до конца строки).

Например,

if (i < MAXLEN)

заменится на

if (i < 512)

Никаких проверок при такой замене не делается - просто текстовая подстановка. Недостатки такого подхода очевидны :

- компилятор не знает имени MAXLEN - оно уже заменено препроцессором на 512. Стало быть, нет объекта с таким именем, и у него нет типа. Отсюда следует, что, если у нас будет какая-то ошибка, относящаяся к оператору с MAXLEN, то сообщения компилятора к этому имени относиться не будут и скорее всего будут непонятными.
- никакой отладчик не позволит вам посмотреть или как-либо иначе использовать значение MAXLEN. Вы вообще никогда его не увидите.

Если же мы будем пользоваться const, то этого не случится - наше maxlen - объект в программе (только неизменяемый). У него есть тип и его значение можно посмотреть в отладчике.

Работая с С++, всегда для констант используйте const и никогда не определяйте константы с помощью #define. Но для того, чтобы понимать программы на С, вам

нужно этот материал знать - там это используется сплошь и рядом (и по инерции остается во многих программах на C++).

Тип перечисление

Тип перечисление определяет множество символических целых констант. Эти константы называются элементами перечисления. Отличие от эквивалентных им описаний со спецификацией const состоит в том, что нет адреса памяти, связанного с каждым элементом перечисления.

Перечисление описывается служебным словом enum и разделенным запятыми списком элементов перечисления. Список заключен в фигурные скобки. По умолчанию первому элементу присваивается значение 0, а каждому последующему - на единицу больше, чем значение предыдущего элемента.

Различают именованные и неименованные перечисления. Неименованные перечисления просто связывают некоторый список констант со значениями:

```
enum  { false, true} ;
```

Теперь false значит 0, а true значит 1.

```
i = false; значит i = 0;
```

Значение может явно присваиваться элементу перечисления. Если каким-то из элементов значение присваивается, а каким-то нет, то значение тех элементов, которым присваивания не было, на единицу больше предыдущего.

Например :

```
enum  false,fail = 0, pass, true = 1 ;
```

Здесь false и fail будут 0, а pass и true - 1.

Именованные перечисления задают уникальный целочисленный тип и могут использоваться как спецификация типа для определения переменных:

```
enum boolean  {false, true} ;
boolean found = false;
```

Переменная found будет типа boolean.

В этом случае множество элементов перечисления будет представлять собой исчерпывающий список значений, которые можно присваивать объектам такого типа.

Например, found = true; будет правильным присваиванием,

а found = 100; вызовет ошибку.

Итак, на настоящий момент мы с вами познакомились с основными встроенными типами C++ (т.н. простыми типами). Теперь мы вернемся к типам данных через несколько лекций, чтобы рассмотреть сложные типы (массивы, указатели, ссылки и т.п).

Выражения

Выражение в языке С - это некоторая допустимая комбинация переменных, констант и операций.

Операции бывают 3 типов:

- унарные (участвуют 1 операнд);
- бинарные (участвуют 2 операнда);
- тернарная (такая операция в языке одна – операция условия, участвуют 3 операнда).

Выражение строится из одной или нескольких операций. Объекты этих операций называются операндами. Для операций используются определенные обозначения. Так, например, в C++ для операции проверки на равенство используется обозначение "==".

Вычисление выражения состоит в выполнении одной или нескольких операций, приводящих к результату. За исключением нескольких особых случаев, обычно связанных с присваиванием, результат выражения является rvalue (значением). Тип данных выражения обычно определяется типами его операндов. Когда в выражении присутствует более одного типа данных, то происходит преобразование типа в соответствии с определенными правилами, о которых будет рассказано далее.

Выражение, в котором участвуют две или более операции, называется составным (например, x+y-z). Порядок применения операций определяется приоритетом

операции (какая операция выполняется раньше, а какая позже) и ее ассоциативностью (слева направо она выполняется или справа налево).

Простейшее выражение - просто операнд без операции - т.е. константа или переменная. Например 3.14159 или index. Тип выражения соответствует типу данной константы или переменной.

Операции

Теперь мы рассмотрим предопределенные операции языка C++. Каждая операция применяется к выражениям и в ее формате будем писать просто «в».

Арифметические операции

К арифметическим операциям языка C относятся:

- вычитание и унарный минус;
+ сложение;
* умножение;
/ деление;
% деление по модулю;
++ увеличение на единицу (increment);
-- уменьшение на единицу (decrement).

Операции одинакового приоритета выполняются:

- бинарные – слева направо;
- унарные - справа налево.

Конечно же, для того чтобы изменить порядок операций, могут использоваться круглые скобки.

Бинарными арифметическими операциями являются +, -, *, / а также операция взятия остатка %. Операции *, / и % имеют более высокий приоритет, чем + и - (как и в Паскале). Например, x + y * z трактуется как x + (y*z), что естественно. Есть два отличия от Паскаля

- Операция % берет остаток аналогично операции mod в Паскале (т.е 22 % 6 будет 4) и применима только к целым

- Если операция / применяется к целым, то результатом деления будет тоже целое, а остаток отбрасывается (22 / 6 будет 3). Это аналогично паскалевскому div.

```
#include <stdio.h>
/* Пример. Целочисленное деление */
main()
{
int x, y;
printf(" Введите делимое и делитель:");
scanf("%d%d", &x, &y):
printf("\nЦелая часть %d\n", x/y);
printf("Остаток от деления %d\n", x%y);
}
```

Есть и унарные операции + и -. Их приоритет выше, чем у бинарных

-x * y значит (-x) * y а не -(x * y).

Следует заметить, что в результате выполнения арифметических операций можно выйти за диапазон типа. Например,

```
unsigned char uc = 32, uc1 = 10;
uc = uc * uc1;
```

В результате uc должно получить значение 320, а верхний диапазон unsigned char, как известно, равен 255. Что при этом будет - зависит от машины, в Borland C++ uc получит значение 64 в результате каких-то усечений. Таких ошибок лучше не допускать - в C++ нет ошибки range check error, как в Паскале и программа продолжит свою работу - как, неизвестно.

Операция деления по модулю % дает остаток от целочисленного деления. Операция % может применяться только к целочисленным переменным. В следующем примере вычисляется целая часть и остаток от деления двух целых чисел.

Операции отношения и логические операции

Логические операции с приоритетами

Таблица 1

Операция|	Назначение	Использование
!	логич.НЕ	! в
< <= > >=	меньше меньше или= больше больше или=	в < в в <= в в > в в >= в
== !=	равно не равно	в == в в != в
&&	логич.И	в && в
||	логич.ИЛИ	в || в

Приоритет операций отношения и логических операций ниже, чем у арифметических операций (i < j+1 трактуется как i < (j+1)).

Логические операции в языке С соответствуют классическим логическим операциям AND(&&), OR (||) и NOT (!), а их результат - соответствующим таблицам, которые принято называть таблицами истинности:

X	Y	X ANDY	X OR Y	NOT X	X XOR Y
1	1	1	1	0	0
1	0	0	1	0	1
0	1	0	1	1	1
0	0	0	0	1	0

Операция XOR называется операцией "исключающее или". В языке С нет знака логической операции XOR, хотя она может быть реализована с помощью операций AND, OR и NOT. Однако в дальнейшем мы будем рассматривать побитовые операции, среди которых операция "исключающее или" уже есть.

Одним из основополагающих идейных отличий С и С++ от Паскаля, так сказать, на повседневном уровне является то, что все операции отношения и логические операции возвращают целое значение -

1 - если условие истинно

0 - если условие ложно

Пример

```
#include <iostream.h>
#include<conio.h>
void main()
{clrscr();     //очистка экрана, функция описана в conio.h
int tr=(101<=105);
int fal=(101>105);
cout<<tr<<'\n';
cout<<fal<<'\n';
getch();    // задержка , функция описана в conio.h
}
```

Программа выведет на экран:

```
1
0
```

В С++ нет булевского типа. Логические операции тоже оперируют с целыми величинами, или с величинами, которые можно преобразовать в целые (в том числе и с плавающей точкой). При этом истиной считается любое ненулевое значение, а ложью - ноль. Отсюда ясно, что функция, возвращающая целое или приводимое к

целому значение и вообще любое выражение, приводимое к целому, может в C++ использоваться в логических условиях.

Обратите внимание на запись равенства в C++. Одной из основных ошибок программистов, переходящих на C и C++ с Паскаля служит такая запись

```
if (i=0)   ...
```

Компилятор здесь не выдаст ошибки (только предупреждение), но программа, скорее всего, будет работать неправильно. Операция "=" это операция присваивания в C++. Необходимо писать

```
if (i==0)   ...
```

Почему компилятор здесь не выдаст ошибки - мы рассмотрим при знакомстве с операциями присваивания.

Логические операции && и || аналогичны по принципу действия паскалевским and и or. Результат && - истина (1) если оба ее операнда истинны (не равны 0). Результат || истина если хотя бы один из ее операндов истина. Гарантируется, что операнды вычисляются слева направо. Вычисление прекращается, как только определится, является ли выражение истинным или ложным. Это значит, что для выражения

```
выр1 && выр2
```

выр2 не будет вычисляться, если значение выр1 ложно (равно 0)
а для выражения

```
выр1 && выр2
```

выр2 не будет вычисляться, если значение выр1 истинно (не равно 0) Следует отметить, что, в отличие от Паскаля, приоритет && и || ниже, чем у операций равенства и отношения и поэтому если нужно, к примеру, проверить, попала ли переменная в диапазон от 0 до 100, то такая запись корректна

```
if (i1 >= 0 && i1 <= 100)
```

и не требует дополнительных скобок.

Логическое отрицание ! преобразует свой операнд в истину (1) если он равен 0 и в ложь (0) если он не равен нулю.

Выражение (!found) возвращает истину, пока found равен нулю.

Можно писать if(!found)

а можно

```
if(found == 0)
```

Каким образом писать - дело программиста, но в сложных выражениях лучше вместо ! явно проверять на равенство.

Так же точно вместо

```
if (found != 0)
```

можно просто написать

```
if (found)
```

Операции присваивания

Еще одним коренным и чрезвычайно важным отличием C++ от Паскаля, которое обуславливает компактность программ на C++ является то, что здесь присваивание является не оператором, а операцией, которая может находиться на любом месте в выражении. Об этом всегда нужно помнить.

В C++ различаются простое и составное присваивание.

Простое присваивание обозначается как =. Левый операнд операции присваивания должен быть lvalue-выражением, т.е. обозначать объект в памяти, которому присвоится значение правого операнда.

Вот пример :

```
int i; i = 0;
```

Результат операции присваивания есть значение того выражения, которое присваивается левому операнду. Это rvalue. Тип результата - тип левого операнда. Т.е. присваивание действует "насквозь" справа налево.

Что следует из того, что присваивание является операцией?

- В одном операторе можно сцеплять несколько операций присваивания при условии, что все операнды имеют один и тот же тип. Например:

```
int i,j,k;
i = j = k = 0;
```

i,j и k присваивается 0. Порядок выполнения присваиваний - справа налево.

- можно короче записывать выражения

Например, в фрагменте программы

```
char ch;      // получаемый символ
char next_char();// выдает следующий символ
if ((ch = next_char()) != '\n')    ...
```

присваивание значения символьной переменной ch совмещается с проверкой.

Вообще здесь нужно знать меру, чтобы не запутать программу.

Теперь мы можем объяснить, почему, когда мы записали при проверке на равенство

```
if (i=0)   ...
```

компилятор не выдал ошибки. Он воспринял выражение в скобках как операцию присваивания и сделал проверку в соответствии с результатом этой операции. При выполнении программы сначала i присвоится ноль, а затем осуществится проверка if (0) и операторы в фигурных скобках не выполнятся никогда.

Составное присваивание - это новая для вас операция. Она также позволяет записывать выражения короче. Общий вид составного присваивания следующий :

```
a op= b
```

где op - одна из десяти операций (одни из них вы уже знаете, о других мы еще будем говорить) :

+ - * / % << >> & ^ |

Каждая составная операция эквивалентна следующему присваиванию : a = (a) op (b);

за тем исключением, что здесь a вычисляется два раза, а при использовании составного присваивания - один раз.

Например,

i *= 4 эквивалентно i = i * 4

Обратите внимание на скобки вокруг b:

запись i *= y+1

эквивалентна записи

i = i * (y+1) а не i = i * y + 1

Приоритет операций присваивания очень низкий, ниже, чем у арифметических и логических операций.

Операции инкремента и декремента (увеличения и уменьшения)

В C++ есть две необычных операции, предназначенных для уменьшения и увеличения переменных. Операции инкремента ++ и декремента -- дают компактную и удобную запись для увеличения или уменьшения переменной на единицу.

Каждая такая операция имеет две формы : префиксную и постфиксную. Например :

```
int c;
++c;      // префиксный инкремент
c++;      // постфиксный инкремент
```

В обоих случаях значение c увеличится на единицу. Но выражение ++c увеличивает c до того, как его значение будет использовано, а c++ - после того. Когда, как в нашем примере, c++ стоит в операторе отдельно, то разницы нет. Но если инкремент или декремент поставить в выражение, то разница появится :

Если c равно 5 то запись x = c++; эквивалентна операторам

```
x = c;   c = c + 1;
```

и x будет равен 5, а запись

```
x = ++c;
```

эквивалентна операторам

```
c = c + 1; x = c;
```

и x будет равен 6. (c будет равно 6 в обоих случаях).

Аналогично обстоят дела с декрементом --. Он уменьшает свой операнд на единицу.

Инкремент и декремент - это присваивания и их операнд должен быть **lvalue**.

Здесь будет ошибка x = (a+b)--;

(значение выражения a+b - rvalue и не связано с объектом в памяти)

и здесь тоже

++a++;

(значение, которое выдает присваивание (и в том числе инкремент) тоже не связано с объектом в памяти).

В отличие от операций простого и составного присваивания, инкремент и декремент имеют высокий приоритет - такой же, как унарный минус или логическое отрицание !.

Для лучшего усвоения материала приведем еще один пример программы с операциями инкремента.

```
#include <stdio.h>
/* Пример  */
main()
{
int x=5;
int y=60:
x++;
++y;
printf("x=%d y=%d\n", x, y);
printf("x=%d y=%d\n", x++, ++y);
```

```
}
```

Результатом работы этой программы будет следующее:

x=6, y=61;

x=6, y=62.

Обратите внимание на то, что напечатанное значение **x** не изменилось при втором обращении к функции **printf()**, а значение **y** увеличилось на единицу. На самом деле значение переменной **x** также увеличилось на единицу, но уже после выхода из функции **printf()**.

Операция sizeof

Операция sizeof возвращает размер (в байтах) выражения или спецификации типа. Она встречается в двух вариантах :

```
sizeof(тип)

sizeof выражение
```

Например, если мы напишем

```
int i,x,y; x = sizeof(int); y = sizeof i;
```

то значения x и y будут одинаковы и в среде Турбо Си++ равны 2. Значение, возвращаемое sizeof, совместимо с любым целочисленным типом. Некоторые другие примеры применения sizeof мы рассмотрим позже.

Условная операция

Операция условие - единственная операция языка С, имеющая три операнда. Эта операция имеет вид:

(выр1) ? (выр2) : (выр3)

Вычисляется выражение (выр1). Если это выражение имеет ненулевое значение, то вычисляется выражение (выр2). Результатом операции будет значение выражения (выр2).

Если значение выражения (выр1) равно нулю, то вычисляется выражение (выр3) и его значение будет результатом операции. В любом случае вычисляется

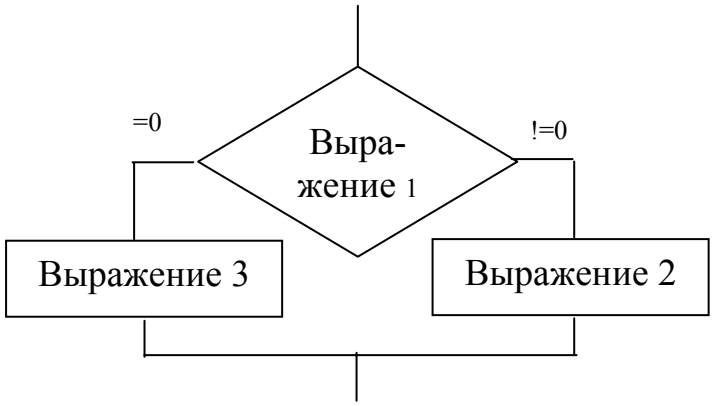


Рис. Схема выполнения операции

только одно из выражений: (выр2) и (выр3). Схема выполнения операции представлена на рис. **.

Например, операцию условие удобно применять для нахождения наибольшего из двух чисел x и y:

```
max = (x>y) ? x: y ;
```

или для нахождения абсолютной величины числа x:

```
abs = (x>0) ? x: -x ;
```

Если второй и третий операнды являются величинами типа lvalue, т.е. могут стоять в левой части операции присваивания, то и результат операции условие является величиной типа lvalue. С помощью этой операции можно в одну строчку решить задачу: наибольшее из чисел x или y заменить значением 1:

```
(x>y) ? x: y = 1;
```

Заключать в скобки первое выражение в условной операции не обязательно, так как приоритет ?: очень низкий (более низкий приоритет имеют только присваи-

вание и операция запятая), но лучше так делать, поскольку в этом случае условие легче воспринимается.

Условная операция часто позволяет сократить программу.

Операция запятая

Выражение с запятой - это несколько выражений, разделенных запятыми. Эти выражения вычисляются слева направо. Типом и значением результата являются тип и значение правого выражения. Такое выражение может быть в любом месте, где допустимо выражение.

Например, запись

```
x =( i=0, j = i+4, k = j);
```

эквивалентна записи

```
i=0; j=i+4; k=j; x=k;
```

Возможность на месте одного выражения записывать несколько, разделенные запятыми, тоже позволяет записывать программу более коротко. Более подробно мы рассмотрим это при знакомстве с операторами C++.

Приоритет операции запятая ниже, чем у любой другой операции в C++. Отметим, что запятые, разделяющие аргументы функции, переменные в определении и т.д. не являются операциями-запятыми и не обеспечивают вычислений слева направо.

Побитовые (поразрядные) операции

Побитовые операции (их еще называют поразрядными) интерпретируют свои операнды как упорядоченный набор битов. Каждый бит может содержать 0 или 1. Эти операции позволяют программисту проверять и устанавливать определенные разряды или подмножества разрядов.

Поразрядные операции можно проводить с любыми целочисленными переменными и константами. Нельзя использовать эти операции с переменными типа float, double и long double. Результатом побитовой операции будет целочисленное значение.

Поразрядными операциями являются:

& AND,

| OR,

^ XOR,

~ NOT,

<< сдвиг влево,

>> сдвиг вправо.

Исследуем, как работает каждая операция. Сразу отметим, что такие операции есть и в Паскале (побитовые not, shl, shr, xor, and и or).

Поразрядная операция НЕ ("~") инвертирует разряды своего операнда. Каждый единичный бит обнуляется, каждый нулевой бит устанавливается в единицу.

```
unsigned char bits = 151; 1 0 0 1 0 1 1 1
bits = ~bits;     0 1 1 0 1 0 0 0
```

Поразрядные операции сдвига ("<<",">>") сдвигают разряды первого операнда на некоторое число разрядов (определяемое вторым операндом) влево или вправо.

```
unsigned char bits = 1;     0 0 0 0 0 0 0 1
bits = bits << 1;       0 0 0 0 0 0 1 0
bits = bits << 2;       0 0 0 0 1 0 0 0
bits = bits >> 3;       0 0 0 0 0 0 0 1
```

Выходящие за границу разряды отбрасываются. Операция сдвига влево << вставляет нулевые биты с правого края объекта.

У вас может возникнуть вопрос - какая связь между сдвигом и обозначаемыми так же операциями ввода-вывода, которые мы рассмотрели на первой лекции? Эта связь - только в одинаковом обозначении. Здесь мы прикоснулись к одной из мощных возможностей C++ - перегрузке операций. Ее мы будем рассматривать в свое время, а пока нужно просто запомнить, что, когда первым операндом операции << или >> (или нескольких сцепленных таких операций) является поток cin или cout, то

это рассматривается как операция ввода или вывода. Иначе это будет операция сдвига.

Поразрядной операции И ("&") необходимы два целочисленных операнда. Для каждого разряда ее результат равен 1, если в обоих операндах по этому разряду хранится 1, иначе результат для этого разряда 0.

```
unsigned char result;
unsigned char b1 = 101;   //0 1 1 0 0 1 0 1
unsigned char b2 = 175;   //1 0 1 0 1 1 1 1
result = b1 &b2;                    //0 0 1 0 0 1 0 1
```

Поразрядной операции исключающее ИЛИ ("^") необходимы два целочисленных операнда. Для каждого разряда ее результат равен 1, если один, но не оба, операнда содержат 1 в этом разряде, иначе результат для этого разряда 0.

```
result = b1 ^ b2;       1 1 0 0 1 0 1 0
```

Поразрядной операции ИЛИ ("|") необходимы два целочисленных операнда. Для каждого разряда ее результат равен 1, если один или оба операнда по этому разряду содержат 1, иначе результат для этого разряда 0.

```
result = b1 | b2;       1 1 1 0 1 1 1 1
```

Поразрядные операции не следует путать с логическими. Логические операции проверяют свои операнды на истинность (равны те нулю или не равны) и дают опять же в результате нуль или единицу.

Есть особая техника работы с поразрядными операциями, применяемая для установки или сброса определенного бита в значении или проверки, установлен или нет определенный бит.

Если надо установить значение старшего разряда переменной типа char равным нулю, то удобно применить операцию & (AND):

```
ch=ch&127;
```

Пусть ch='A'

'A' 11000001

127 01111111

'A'&127 01000001

Если же мы хотим установить старший разряд равным единице, то удобна операция OR:

ch =ch | 128;

'A' 11000001

128 10000000

'A'|128 11000001

Поразрядные операции удобны также для организации хранения в сжатом виде информации о состоянии on/off (включен/выключен). В одном байте можно хранить 8 таких флагов.

Если переменная ch является хранилищем таких флагов, то проверить, находится ли флаг, содержащийся в третьем бите, в состоянии on, можно следующим образом:

if(ch & 4) printf("3 бит содержит 1, состояние on");

Эта проверка основывается на двоичном представлении числа 4=00000100.

Сводная таблица приоритетов и ассоциативности операций

Для обобщения данного материала и для удобства справок приведем таблицу, в которой сведены все операции, известные нам и те, которые мы еще будем рассматривать и их приоритеты.

Операции языка C++

Таблица 2

Операция	Выполняется	Назначение
::	Справа налево	Глобальная видимость
::	(П)	(унарная)
	Слева направо	Область видимости

	(Л)	класса (бинарная)
-> [] () {}	Л Л Л Л	Выбор элемента Индексирование массива Вызов функции Конструирование типа
Sizeof ++ -- ~ ! + - * & () new delete	П П П П П П П П	Размер в байтах Инкремент, декремент Побитовое НЕ Логическое НЕ Унарный минус, унарный плюс Косвенность, взятие адреса Приведение типа Выделение, освобождение памяти
->* .*	Л	Выбор элемента по указателю
* / %	Л	Умножение, деление, взятие остатка
+ -	Л	Сложение, вычитание
>> <<	Л	Сдвиги влево, вправо
< <== > >=	Л	Операции отношения
== !=	Л	Равно, не равно
&	Л	Побитовое И
^	Л	Побитовое исключ. ИЛИ

|	Л	Побитовое ИЛИ
&&	Л	Логическое И
||	Л	Логическое ИЛИ
? :	Л	Условная операция
= += -= *= и т.д.	П	Присваивания
,		Операция запятая

Преобразования типа

Если операнды операции принадлежат разным типам, то они приводятся к некоторому общему типу. Преобразования типа - один из наиболее трудных для понимания вопросов в программировании на C++

Различаются неявные и явные преобразования типов.

1. Неявные преобразования типов

Неявные преобразования транслятор выполняет без вмешательства программиста. Они применяются всякий раз, когда смешиваются различные типы данных. Такие преобразования выполняются согласно предопределенным правилам, называемым стандартными преобразованиями :

- Присваивание значения объекту преобразует это значение в тип объекта, передача значения как параметра функции (мы рассмотрим передачу параметров функции позже) преобразует это значение в тип параметра функции. Если тип объекта требует для своего представления меньше битов, (является более узким типом), чем тип присваиваемого объекту значения, то говорят, что происходит сужение типа.

Например :

```
void f(int);
int i = 3.14;  // 3.14 преобразуется к int (i==3)
```

f(3.14); // в f передается 3

В обоих случаях константа 3.14 типа double неявно преобразуется транслятором в тип int - дробная часть при этом будет отброшена. Компилятор Borland C++ не выдаст даже предупреждения.

Сужение типа может привести к потере данных, как мы только что видели. Аналогично преобразование беззнакового типа к знаковому тоже считается сужением типа, например :

unsigned char uc = 200;

signed char c;

c = uc;

c здесь не будет равняться uc, т.к. самый левый бит будет трактоваться как знаковый. Значение c будет -56.

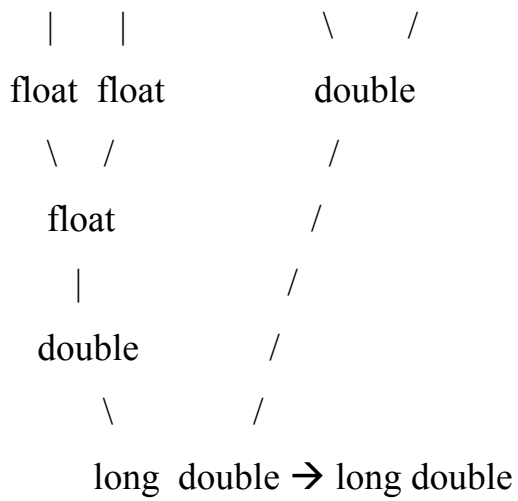
- В арифметическом выражении более широкий тип данных становится результирующим типом. Это действие называется расширением типа.

Пример преобразований типов переменных при вычислении выражений. Пусть определены переменные следующих типов:

```
char        ch;
int         i;
float       f;
double          d;
long double r;
```

При вычислении выражения произойдут следующие автоматические преобразования типов:

```
r =  ch * 2 + (i -  0.5) +  ( f  +  d )  -  7
        char  int   int  float   float  double int
        |    |   |   |     |      |    |
        int   int  float  float  double double
          \   /  \   /       \    /         |
           int   float       double    double
```



2. Явные преобразования типов

Тип результата вычисления выражения можно изменить, используя конструкцию «приведение», имеющую следующий вид:

(тип) выражение

Здесь «тип» - один из стандартных типов данных языка С.

Например, если вы хотите, чтобы результат деления переменной x типа int на 2 был типа float, напишите

(float) x/2

Значение же выражения

(float) (x/2)

будет другим. Например, при x=5 результат выражения (float) x/2 будет равен 2,5. А результат выражения (float) (x/2) равен 2.

Пробелы и круглые скобки в выражениях можно расставлять так, как вы считаете нужным. Это делается для удобства чтения программы на языке С. При компиляции лишние пробелы просто игнорируются.

Второй способ записи - это появившаяся в С++ аналогичная Паскалю функциональная запись вида

тип(выражение)

Предыдущий пример можно записать в виде

```
i = i + int(3.14);
```

Обычно так читать удобнее.

Следует отметить, что такая запись доступна только для типов, имя которых является простым (выражается одним словом). Что значит не простое имя - мы выясним при рассмотрении сложных типов С++.

С помощью явного преобразования можно вообще отключить проверку типов, поскольку С++ допускает явное преобразование к любому типу данных. Следует отметить, что вся ответственность за ошибки здесь лежит на программисте.

Явные преобразования типов повышают вероятность ошибок в программах и если в них нет особой необходимости, то лучше ими не пользоваться.

Операции () и [ ]

В языке С круглые и квадратные скобки также рассматриваются как операции. Причем эти операции имеют наивысший приоритет.

Поразрядные операции порождают еще несколько сложных операций присваивания:

|=, &=, ^=, <<=, >>=.

Порядок вычислений

В С++ не фиксируется порядок вычисления операндов операции (за исключением операций || && ?: и запятая. Это может оказаться существенным.

Например, в выражении

```
x = f() + g()
```

функция f() может быть вызвана раньше g() и наоборот.

Еще более хитрый пример

```
i = 1;
a[i] = i++;
```

В одном компиляторе будет взято a[1], в другом a[2]. Такое выражение (и вообще любое выражение, где значения операндов изменяются при его вычислении) называется выражением с побочным эффектом.

Нужно стараться избегать зависимости значения выражения от порядка вычислений.

Форматный ввод и вывод данных

Форматированный ввод и вывод на консоль осуществляют функции printf ( ) и scanf ( ). Форматированный ввод и вывод означает, что функции могут читать и выводить данные в разном формате, которым можно управлять.

Функция printf() имеет прототип в файле STDIO.H

```
int printf(char *управляющая_строка,  ...);
```

Управляющая строка содержит два типа информации: символы, которые непосредственно выводятся на экран, и команды формата (спецификаторы формата), определяющие, как выводить аргументы. Команда формата начинается с символа %, за которым следует код формата. Команды формата следующие:

%c - символ,

%d - целое десятичное число,

%i - целое десятичное число,

%e - десятичное число в виде x.xx e+xx,

%E - десятичное число в виде x.xx E+xx,

%f - десятичное число с плавающей занятой xx.xxxx,

%F - десятичное число с плавающей запятой xx.xxxx,

%g - %f или %e, что короче,

%G - %F или %E, что короче,

%o - восьмеричное число,

%s - строка символов,

%u - беззнаковое десятичное число,

%x - шестнадцатеричное число 5a5f,

%X - шестнадцатеричное число 5A7F,

%% - символ %,

%p - указатель,

%n - указатель.

Кроме того, к командам формата могут быть применены модификаторы l и h, например:

%ld - печать long int,

%hu - печать short unsigned,

%Lf - печать long double.

Между знаком % и форматом команды может стоять целое число. Оно указывает на наименьшее поле, отводимое для печати. Если строка или число больше этого поля, то строка или число печатается полностью, игнорируя ширину поля. Нуль, поставленный перед целым числом, указывает на необходимость заполнить неиспользованные места поля нулями. Вывод

```
printf("%05d", 15);
```

даст результат 00015.

Чтобы указать число десятичных знаков после целого числа, ставится точка и целое число, указывающее на количество десятичных знаков. Когда такой формат применяется к целому числу или к строке, то число, стоящее после точки, указывает на максимальную ширину поля выдачи.

Выравнивание выдачи производится по правому краю поля. Если мы хотим выравнивать по левому знаку поля, то сразу за знаком % следует поставить знак минуса. В прототипе функции многоточием обозначен список аргументов - переменных или констант, которые следуют через запятую и подлежат выдаче в соответствующем формате, следующем по порядку.

Scanf() - основная функция ввода с консоли. Она предназначена для ввода данных любого встроенного типа и автоматически преобразует введенное число в заданный формат. Прототип из файла STDIO.H имеет вид

```
int scanf (char *управляющая_строка, ...);
```

Управляющая строка содержит три вида символов: спецификаторы формата, пробелы и другие символы. Команды или спецификаторы формата начинаются с символа %. Они перечислены ниже:

%c - чтение символа,

%d - чтение десятичного целого,

%i - чтение десятичного целого,

%e - чтение числа типа float,

%h - чтение short int.

%o - чтение восьмеричного числа,

%s - чтение строки.

%x - чтение шестнадцатеричного числа,

%p - чтение указателя,

%n - чтение указателя в увеличенном формате.

Символ пробела в управляющей строке дает команду пропустить один или более пробелов в потоке ввода. Кроме пробела, может восприниматься символ табуляции или новой строки. Ненулевой символ указывает на чтение и отбрасывание (discard) этого символа. Все переменные, которые мы вводим, должны указываться с помощью адресов, как и положено в функциях языка С.

Строка будет читаться как массив символов, и поэтому имя массива без индексов указывает адрес первого элемента.

Разделителями между двумя вводимыми числами являются символы пробела, табуляции или новой строки. Знак * после % и перед кодом формата дает команду прочитать данные указанного типа, но не присваивать это значение. Так,

```
scanf("%d%*c%d", &i, &j);
```

при вводе 50+20 присвоит переменной i значение 50, переменной j - значение 20, а символ + будет прочитан и проигнорирован.

В команде формата может быть указана наибольшая ширина поля, которая подлежит считыванию. К примеру,

```
scanf("%5s", str);
```

указывает на необходимость прочитать из потока ввода первые 5 символов. При вводе 123456789 массив str будет содержать только 12345, остальные символы будут проигнорированы. Разделители: пробел, символ табуляции и символ новой строки - при вводе символа воспринимаются, как и все другие символы.

Если в управляющей строке встречаются какие-либо другие символы, то они предназначаются для того, чтобы определить и пропустить соответствующий символ. Поток символов 5plusl0 оператором

```
scanf("%dplus%d", &i, &j);
```

присвоит переменной i значение 5, переменной j - значение 10, а символы plus пропустит, так как они встретились в управляющей строке. К недостаткам, правда преодолимым, функции scanf() относится невозможность выдачи приглашения к вводу, т. е. приглашение должно быть выдано до обращения к функции scanf().

Одной из мощных особенностей функции scanf() является возможность задания множества поиска (scanset). Множество поиска определяет набор символов, с которыми будут сравниваться читаемые функцией scanf() символы. Функция scanf() читает символы до тех пор, пока они встречаются в множестве поиска. Как только символ, который введен, не встретился в множестве поиска, функция scanf() переходит к следующему спецификатору формата. Множество поиска определяется списком символов, заключенных в квадратные скобки. Перед открывающей скобкой ставится знак %. Чтобы увидеть, как используется эта возможность, рассмотрим пример.

Пример 8.

```
#include <stdio.h>
/* Форматный ввод с использованием
множества поиска */
main (void)
{
char s[10], t[10];
scanf ("%[0123456789]%s", s, t);
```

```
printf ("\n%s..%s", s, t);
}
```

Введем следующий набор символов:

"123abc456"

На экране программа выдаст

123..aЬc45б

Так как **a** не входит в множество поиска (оно состоит только из цифр), то ввод по первому спецификатору формата прерывается и начинается ввод по второму спецификатору формата.

При задании множества поиска можно также использовать символ "дефис" для задания промежутков, а также максимальную ширину поля ввода

```
scanf("%10[A-Z1-5]%s", s);
```

Такой формат позволяет вводить в строку **s** заглавные буквы от A до Z, а также цифры от 1 до 5. Кроме того, длина строки ограничена 10 символами.

Можно также определить символы, которые не входят в множество поиска. Перед первым из этих символов ставится знак ^. И множество символов различает, естественно, строчные и прописные буквы.

Дополнительные возможности ввода/вывода в языке C++

Язык C++ имеет свою библиотеку ввода/вывода. Она находится в файле iostream.h. Этот файл содержит средства управления потоками ввода/вывода.

Ввод, идущий с клавиатуры пользователя, называется стандартным входным потоком или стандартным вводом. Он связывается с предопределенным в iostream.h потоком cin. Вывод, направляемый на экран пользователя,

называется стандартным выходным потоком или стандартным выводом. Он связывается с предопределенным в iostream.h потоком cout.

Операция вывода << направляет значение в стандартный выходной поток.

```
cout << index;
```

Для перехода на новую строку существуют два способа. Первый - это использовать определенный в iostream.h манипулятор endl. Манипулятор можно выводить в поток и при этом он меняет параметры вывода. Здесь endl вызовет переход на новую строку

```
cout << endl;
```

Второй способ - это явно вывести в поток символ новой строки. В C++ он записывается двумя символами : '\n'.

```
cout << '\n';
```

Одиночные кавычки ограничивают символ. Такой символ может быть внутри строки символов, например оператор

```
cout << "Программа на C++\n";
```

вызовет переход на новую строку после вывода данного сообщения.

В одном операторе вывода можно соединять несколько операций.

Например :

```
cout << "Значение index равно : " << index << endl;
```

Вывод осуществляется по порядку, считая слева направо.

Аналогично операция ввода >> читает значение из стандартного входного потока, например:

```
cin >> index;
```

Такие операции тоже можно соединять в одном операторе. Например, если в программе встретится следующий оператор :

```
cin >> i1 >> i2;
```

то программа будет ждать ввода с клавиатуры двух величин и первую из них поместит в переменную i1, а вторую - в переменную i2. Эти две вводимых величины можно разделять пробелом или табуляцией, а можно каждую из них вводить с новой строки - операция ввода сработает правильно.

Если программист забудет включить в программу файл iostream.h, то каждом появлении в программе cin или cout транслятор будет сообщать как об ошибке, поскольку cin и cout описаны в iostream.h.

Третий предопределенный поток из iostream.h называется cerr и является стандартным потоком сообщений об ошибках. Он тоже связан с экраном пользователя и нужен, чтобы сообщать пользователю об особых ситуациях и ошибках при выполнении программы.

Например

```
cerr << "Ошибка чтения диска" << endl;
```

Для облегчения понимания операций << и >> можно представлять их как воронки, через которые информация из программы (например, из переменной i1) выводится на экран (cout)

```
cout << i1;
```

или с клавиатуры (cin) попадает в программу (например, в переменную i2)

```
cin >> i2;
```

ОПЕРАТОРЫ ЯЗЫКА

Каждый оператор в языке должен заканчиваться точкой с запятой (;). Часто в качестве оператора выступает выражение. Выражением может служить и вызов функции, не возвращающей никакого значения. Чаще всего оператор-выражение – это выражение присваивания. В C++ нет отдельного оператора присваивания, только операция.

Пустой оператор

В языке существует пустой оператор. Его синтаксис:

```
;
```

Этот оператор используется там, где по синтаксису языка требуется оператор, а по смыслу программы никакие действия не выполняются.

Составные операторы и блоки

В ряде синтаксических конструкций языка можно задавать только один оператор, а логика программы требует нескольких. В этом случае, как и в Паскале, могут использоваться составные операторы

```
if (i < 0)
{
i = 200;
j = i + 600;
}
```

Составной оператор - это последовательность операторов, заключенная в фигурные скобки. Такие скобки аналогичны операторным скобкам begin...end в Паскале. Составной оператор может встречаться в программе везде, где может быть отдельный оператор. После правой закрывающей фигурной скобки в конце блока точка с запятой не ставится. Обратите внимание на отличие от Паскаля: в Паскале перед end после последнего оператора (у нас j := i+600) может и не стоять точка с запятой, а в C++ она обязательна, т.к. заканчивает оператор.

Если среди операторов в составном операторе имеются определения и описания, то такой составной оператор называется **блоком**.

Блок и составной оператор пользуются всеми правами операторов и могут вкладываться друг в друга. На глубину вложенности язык не накладывает ограничений.

Управляющие операторы

Управляющие операторы можно разделить на три категории:

1. Условные операторы if, if - else и switch.
2. Операторы цикла for, while и do - while.
3. Оператор безусловного перехода goto.

Условный оператор IF

Существует два варианта оператора if: так называемые полная и краткая формы. Полная форма этого оператора следующая:

```
If (условие) оператор1;
    else оператор2;
    следующий оператор программы;
```

Эта форма соответствует схеме алгоритма, приведенной на рис.1

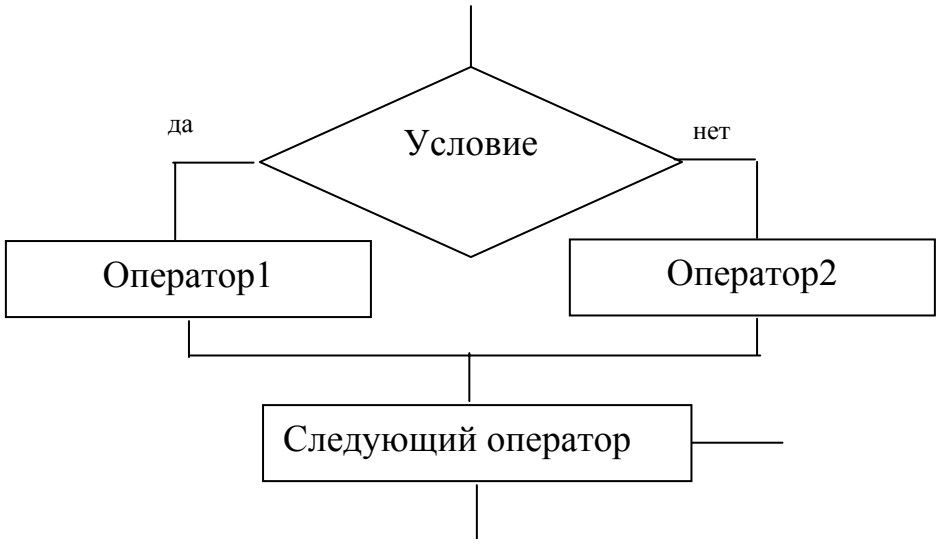


Рис.1 Полная форма оператора if

Если значение условия «истинно», то выполняется оператор (им может быть составной оператор - блок), следующий за условием. Если же условие принимает значение "ложно", то выполняется оператор, следующий за ключевым словом else.

В записи оператора if вторая часть (т. е. оператор else) может отсутствовать (краткая форма):

```
if(условие) оператор;
    следующий оператор программы;
```

Тогда, если условие принимает значение «ложно», выполняется сразу следующий оператор программы. В языке C++, как уже упоминалось, нет логического типа, поэтому в качестве условия может стоять произвольное выражение. В операторе if лишь проверяется, является ли значение этого выражения ненулевым (истинным) или нулевым (ложным). С помощью оператора if можно, например, вычислить значение функции sgn(x) - знак x. Функция sgn(x) принимает значение 1, если x > 0, значение -1, если x < 0, значение 0, если x = 0.

```
#include <stdio.h>
/* Пример  */
main()
{
int sgn;
float x;
printf(" Введите число:");
scanf("%f", &x);
if(x>0) { sgn=l;
     printf ("Число %f положительное  sgn = %d \n", x, sgn); }
if(x==0) { sgn=0;
      printf ("Число %f равно нулю   sgn = %d \n", x, sgn); }
if(x<0) { sgn=-1;
     printf ("Число %f отрицательное  sgn = %d \n", x, sgn); }
```

Часто встречается необходимость использовать конструкцию if-else-if:

```
if (условие) оператор;
else if (условие) оператор;
else if (условие) оператор;
…
        else оператор;
```

В этой форме условия операторов **if** проверяются сверху вниз. Как только какое-либо из условий принимает значение «истинно», выполнится оператор, следующий за этим условием, а вся остальная часть конструкции будет проигнорирована. Операторы **if** предыдущего примера могут быть записаны в другом виде:

```
#include <stdio.h>
/* Пример . */
main()
{
int sgn;
float x;
printf (" Введите число:");
scanf ("%f", &x);
if (x>0) {sgn=1; printf ("Число %f положительное \n", x); }
else if (x<0) {sgn= -1; printf ("Число %f отрицательное \n", x); }
else { sgn=0;printf ("Число %f равно нулю \n", x); }
}
```

В качестве условия оператора **if** может использоваться, как мы уже сказали, некоторое выражение. Так, для того чтобы проверить, равно число **x** нулю или не равно, можно написать

```
if (x==0) printf ("Число равно нулю");
else printf ("Число не равно нулю");
```

Тот же результат можно получить следующим оператором:

```
if(!x) printf ("Число равно нулю");
```

```
else printf ("Число не равно нулю");
```

Вложенным оператором if называется следующая конструкция:

```
if(x)
    if (y) оператор1;
    else оператор2;
```

В такой форме непонятно, к какому из операторов **if** относится **else**. В языке С оператор **else** ассоциируется с ближайшим **if** в соответствующем блоке. В последнем примере **else** относится к **if(y)**. Для того чтобы отнести else к оператору **if(x),** нужно соответствующим образом расставить операторные скобки:

```
if (x){
 if (y) оператор1;
}
else оператор2;
```

Теперь if(y) относится к другому блоку.

Оператор SWITCH

Язык С++ имеет встроенный оператор множественного выбора, называемый switch. Основная форма оператора имеет такой вид:

```
switch (выражение) {
 case constant1:
последовательность операторов
break;
case constant2:
последовательность операторов
break;
…
case constantN:
последовательность операторов
break;
```

```
default
последовательность операторов
}
```

Сначала вычисляется выражение в скобках за ключевым словом switch. Затем просматривается список меток (case constant 1 и т. д.) до тех пор, пока не находится метка, соответствующая значению вычисленного выражения. Далее происходит выполнение соответствующей последовательности операторов, следующих за двоеточием. Если же значение выражения не соответствует ни одной из меток оператора switch, то выполняется последовательность операторов, следующая за ключевым словом default.

Допускается конструкция оператора switch, когда слово default и соответствующая последовательность операторов можут отсутствовать.

Выполнение оператора break приводит к выходу из оператора switch и переходу к следующему оператору программы. Наличие оператора break в операторе switch необязательно. Что будет, если операторов break не будет? Ответ на этот вопрос дадут результаты работы следующих двух вариантов программы.

Вариант 1

```
#include <sidio.h>
/* Пример 15. */
/* Пример оператора switch с использованием break */
main()
{
char ch;
printf ("Введите заглавную букву русского алфавита:");
ch=getchar();
if(ch>='А' && ch<='Я')
switch(ch)
{
case 'А':
printf ("Алексеев \n");  break;
```

```
case 'Б':
printf(" Булгаков \n");  break;
case 'В':
printf  (" Волошин \n");  break;
case 'Г':
printf("Гоголь\n");  break;
default:
printf ("Достоевский, Зощенко и другие \n");  break;
}
else printf ("Надо было ввести заглавную русскую букву\n"):
}
```

Вариант 2

```
#include <stdio.h>
/* Пример 16. */
/* Пример оператора switch без использования break */
main()
{
char ch:
printf ("Введите заглавную букву русского алфавита:");
ch=getchar();
if(ch>='А'&&ch<='Я')
switch(ch)
{
case 'А': printf ("Алексеев \n");
case 'Б': printf(" Булгаков \n");
case 'В': printf(" Волошин \n");
case 'Г': printf  (Гоголь \n");
default: printf ("Достоевский, Зощенко и другие \n");
}
```

```
else printf ("Надо было ввести заглавную русскую букву \n"):
}
```

Предположим, вы запустили первую программу и ввели букву Б. Результатом работы программы будет следующая строка:

```
Булгаков
```

Выполнился только один оператор, соответствующий метке 'Б'. В случае запуска другой программы и ввода буквы Б результат работы программы будет следующий:

```
Булгаков
Волошин
Гоголь
Достоевский, Зощенко и др.
```

Мы видим, что выполнились все операторы, начиная с метки 'Б', включая тот, который следует за словом default.

Оператор break заканчивает последовательность операторов, относящихся к каждой метке. Далее выполняется следующий оператор программы. Если же оператор break отсутствует, то выполнение продолжается до первого оператора break или до конца оператора switch.

3. ЦИКЛЫ

Циклы необходимы, когда нам надо повторить некоторые действия несколько раз, как правило, пока выполняется некоторое условие. В языке С известно три вида оператора цикла:

- цикл с предусловием (while (выражение_условие) тело_цикла;)
- цикл с постусловием (do тело_цикла while(выражение_условие);
- итерационный цикл (for (инициализация; выражение_условие; список_выражений;) тело_цикла;

Тело_цикла не может быть описанием или определением. Это либо отдельный оператор (в том числе и пустой), либо составной оператор, либо блок.

Выражение_условие – это скалярное выражение (логическое или арифметическое выражение), определяющее условие продолжения выполнения итераций.

Цикл FOR

Основная форма цикла **for** имеет следующий вид:

```
for (инициализация ; выражение_условие ; выражения_модификации)
тело_цикла;
```

или в общем виде

```
for (выражение1 ; выражение2 ; выражение 3) тело_цикла;
```

Инициализация -последовательность определений и выражений, разделенных запятыми (если принять во внимание, что запятая есть операция, то, можно говорить о выражении вообще). Все они выполняются только 1 раз при входе в цикл. В простейшей форме инициализация используется для присвоения начального значения параметру цикла.

Выражение_условие - обычно выражение, которое определяет, когда цикл должен быть завершен: завершение цикла происходит при нарушении истинности этого выражения (как уже говорилось, истиной в языке, считается любое ненулевое значение).

Выражения_модификации – состоит из выражений, разделенных запятыми. Эти выражения вычисляются на каждой итерации после выполнения тела цикла и до следующей проверки выражения условия.

Эти три раздела заголовка цикла должны быть разделены точкой с запятой. Выполнение цикла происходит до тех пор, пока условное выражение истинно. Как только условие становится ложным, начинает выполняться следующий за циклом for оператор. Схема работы оператора показана на рис. **.

Рис. **. Схема выполнения оператора for

Простейший пример оператора цикла **for**:

```
for ( i=0; i<10; i++ ) printf (“%d \n", i);
```

В результате выполнения этого оператора будут напечатаны в столбик цифры от 0 до 9. Для печати этих цифр в обратном порядке можно использовать следующий оператор:

```
for ( i=9; i>=0; i--) printf ("%d\n", i);
```

Цикл **for** похож на аналогичные циклы в других языках программирования, и в то же время этот оператор в языке С гораздо более гибкий, мощный и применим во многих ситуациях.

В качестве параметра цикла необязательно использовать целочисленный счетчик. Приведем фрагмент программы, выводящей на экран буквы русского алфавита:

```
unsigned char ch;
for (ch='А'; ch<='Я'; ch++) printf ("%c ", ch);
```

Следующий фрагмент программы

```
for (ch='0'; ch!='N';) scanf ("%c", &ch):
```

будет выполняться до тех пор, пока с клавиатуры не будет введен символ 'N'. Заметим, что место, где должно быть приращение, пусто. Случайно или намеренно может получиться цикл, из которого нет выхода, так называемый бесконечный цикл.

Приведем три примера таких циклов.

```
for (;;) printf ("Бесконечный цикл\n");
for (i=1;1;i++) printf(" Бесконечный цикл\n");
for (i=10;i>6;i++) printf ("Бесконечный цикл\n");
```

Тем не менее, для таких циклов также может быть организован выход. Для этого используется оператор **break,** который мы встречали выше. Если оператор **break** встречается в составном операторе цикла, то происходит немедленное прекращение выполнения цикла и начинается выполнение следующего оператора программы.

```
# include <stdio.h>
/* Пример 17. */
main()
{
unsigned char ch;
for(;;)
{
```

```
ch=getchar();      /* Прочитать символ */
if(ch=='Q') break;         /* Проверка символа */
printf("%c", ch);   /* Печать символа */
}
}
```

В этой программе будут печататься введенные символы до тех пор, пока не будет введен символ 'Q'.

Возможно, и это синтаксически правильно, наличие пустого оператора (отсутствие оператора) в цикле for. Например

```
for(i=0;i<10000;i++);
```

Приведем еще несколько примеров оператора for. Все эти операторы, каждый по-своему, решают задачу суммирования квадратов первых k натуральных чисел:

```
for (int j=1, s=0; j<=k; j++) s+=j*j;
for (int j=1,s=0; j<=k; s+=j*j++);
for (int j=0,s=0; j<=k; )s+=++j*j;
```

Обычно принято, что область действия имен, определенных в части инициализации цикла, - от места размещения цикла **for** до конца блока, в котором он используется.

Пример.

```
for (int i=0; i<3;i++) cout<<”\t”<<i;
for (;i>0;i--) cout<<”\t”<<i;
```

На экране будет выведено:

0 1 2 3 2 1

Если во второй цикл поместить определение той же переменной **i**, то будет выдано сообщение об ошибке.

3.2. Циклы WHILE И DO-WHILE

Следующий оператор цикла в языке С - это цикл **while**. Основная его форма имеет следующий вид:

while (условие) оператор;

где оператор может быть простым, составным или пустым оператором. «Условие», как и во всех других операторах, является просто выражением. Цикл выполняется до тех пор, пока условие принимает значение «истинно» (т.е. отлично от нуля). Когда же условие примет значение «ложно», программа передаст управление следующему оператору программы. Так же, как и в цикле **for**, в цикле **while** сначала проверяется условие, а затем выполняется оператор – это, так называемый, цикл с предусловием.

Тип «условия» должен быть арифметическим или приводимым к нему.

Рассмотрим пример программы, печатающей таблицу значений функции $y=x^2+1$ в определенном пользователем диапазоне и с задаваемым шагом.

```
#include <stdio.h>
Int main()
{float xn, xk,dx;
printf("Введите диапазон и шаг изменения аргумента",);
scanf("%f%f%f", &xn, &xk, &dx);
printf("|    x    |    y    |\n");
float x=xn;
while(x<=xk)
{  printf("|  %5.2f  |  %5.2f  |", x, x*x+1);
X+=dx;
}
return 0;
}
```

В отличие от предыдущих циклов в цикле **do - while** условие проверяется в конце оператора цикла. Основная форма оператора **do-while** следующая:

```
do {
последовательность операторов
} while (условие);
```

Сначала выполняется последовательность операторов, составляющая тело цикла, а затем вычисляется выражение. Если оно истинно (не равно нулю), тело цикла выполняется еще раз.Фигурные скобки необязательны, если внутри них находится один оператор. Тем не менее, они чаще всего ставятся для лучшей читаемости программы, а также чтобы не спутать (программисту, а не компилятору) этот цикл с оператором **while**.

Оператор **do-while** называется оператором цикла с постусловием. Какое бы условие ни стояло в конце оператора, набор операторов в фигурных скобках один (первый) раз выполнится обязательно. В циклах **for** и **while** оператор может не выполниться ни разу.

```
#include <stdio.h>
#include <stdlib.h>
/* Пример 18. */
/* Игра " угадай число "*/
/* Программа выбирает случайное число от 1 до 100, вы должны угадать его
*/
main()
{
int s, x;
int n=0;
randomize();
s=random(l00)+l;
do {
printf ("Введите число от 1 до 100: ");
scanf ("%d", &x);
n++;
if (s<x) printf ("Загаданное число меньше\n");
if (s>x) printf ("Загаданное число больше\n");
} while (s-x);
printf ("Вы угадали число !\n");
```

```
printf ("Затратили на угадывание %d попыток\n", n);
}
```

3.3. Вложенные циклы

Когда один цикл находится внутри другого, то говорят, что это вложенные циклы. Часто встречаются вложенные циклы, например, при заполнении таблиц (перемещение по строкам, а внутри строки – по столбцам). В качестве примера рассмотрим программу печати таблицы умножения целых чисел.

```
include <stdio.h>
/* Пример 19. */
main()
{
int i, j;
for (i=l;i<10;i++)          //внешний цикл
  {
    for (j=l;j<5J++)              //вложенный цикл
    printf ("%d*%d = %2d ", i, j, i*j);
    printf("\n");
  }
}
```

3.4. Использование оператора BREAK в циклах

Оператор break имеет два применения. Первое - окончание case в операторе switch. Второе - немедленное окончание цикла, не связанное с проверкой обычного условия окончания цикла. Когда оператор break встречается внутри оператора цикла, то происходит немедленный выход из цикла и переход к выполнению оператора, следующего за оператором цикла.

```
#include <stdio.h>
/* Пример 20. */
main()
{
```

```
int i;
for (i=0;i<1000;i++)
        {
          printf ("%d - %d \n", i, i*i*i);
          if(i*i*i >= 10000) break;
        }
}
```

3.5. Оператор CONTINUE

Еще один полезный оператор - оператор **continue**. Если оператор **continue** встретился в операторе цикла, то он передает управление на начало следующей итерации цикла. В циклах **while** и **do-while** - на проверку условия, в цикле **for** - на приращение. Этот оператор необходим, если вы хотите закончить текущую итерацию цикла и не выполнять оставшиеся операторы, а сразу перейти к следующей итерации цикла. Например, его можно использовать в программе, печатающей натуральные числа кратные семи.

```
#include <stdio.h>
/* Пример 21.*/
rnain()
{
int i;
for(i=l;i<1000;i++)
  {
   if (i%7) continue;
    printf (%8d", i );
   }
}
```

3.6. Оператор GOTO

Язык С обладает всеми возможностями для написания хорошо структурированных программ. Апологеты структурного программирования считают дурным то-

ном использование оператора **goto**, без которого было тяжело обойтись в таких языках, как FORTRAN или BASIC. Тем не менее оператор **goto** в языке С есть, и иногда он может быть полезен, хотя без него можно обойтись в любой ситуации. Для использования оператора **goto** надо ввести понятие метки (**label**). ***Метка*** - это идентификатор, за которым следует двоеточие. *Метка должна находиться в той же функции, что и оператор* ***goto***. Одно из полезных применений оператора **goto** - это выход из вложенных циклов:

```
for () {
while ( ){
for ( ) {
…
goto exit;
…
}
}
}
exit: printf ("Быстрый выход из вложенных циклов");
```

4. МАССИВЫ

Ранее рассмотренные типы данных в языке С называются иногда базовыми или встроенными. На основе этих типов данных язык С позволяет строить другие типы данных и структуры данных.

Массив - одна из наиболее простых и известных структур данных. Под массивом в языке С понимают набор данных одного и того же типа, собранных под одним именем. Каждый элемент массива определяется именем массива и порядковым номером элемента, который называется ***индексом***. Индекс в языке С всегда целое число.

УКАЗАТЕЛИ И МАССИВЫ

Когда компилятор обрабатывает оператор определения переменной, например, **int i=10** он выделяет память в соответствии с типом (**int**) и инициализирует ее указанным значением (**10**). Все обращения в программе к переменной по ее имени (**i**) заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются *указателями*.

Итак, указатели предназначены для хранения адресов областей памяти. В C++ различают три вида указателей:

- указатели на объект;
- на функцию;
- на тип void,

Виды указателей отличаются свойствами и набором допустимых операций. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим конкретным типом.

Указатель на функцию содержит адрес в сегменте кода, по которому располагается исполняемый код функции, то есть адрес, по которому передается управление при вызове функции. Указатели на функции используются для косвенного вызова функции (не через ее имя, а через обращение к переменной, хранящей ее адрес), а также для передачи имени функции в другую функцию в качестве параметра. Указатель функции имеет тип «указатель функции, возвращающей значение заданного типа и имеющей аргументы заданного типа»:

```
тип (*имя) ( список типов аргументов );
```

Например, объявление:

```
int (*fun) (double, double)
```

задает указатель с именем fun на функцию, возвращающую значение типа int и имеющую два аргумента типа double.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа (основного или составного). Простейшее объявление указателя на объект (в дальнейшем называемого просто указателем) имеет вид:

```
тип *имя;
```

где тип может быть любым, кроме ссылки (о ссылках рассказывается далее) и битового поля, причем тип может быть к этому моменту только объявлен, но еще не определен (следовательно, в структуре, например, может присутствовать указатель на структуру того же типа).

Звездочка относится непосредственно к имени, поэтому для того, чтобы объявить несколько указателей, требуется ставить ее перед именем каждого из них. Например, в операторе

```
int *a. Ь. *c;
```

описываются два указателя на целое с именами **a** и **c**, а также целая переменная b. Размер указателя зависит от модели памяти. Можно определить указатель на указатель и т. д.

*Указатель на **void*** применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов раз личных типов).

Указателю на **void** можно присвоить значение указателя любого типа, а также, сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

Указатель может быть константой или переменной, а также указывать на константу или переменную. Рассмотрим примеры:

```
'4
int i;            // целая переменная
const int ci = 1;    //целая константа
int *pi;              // указатель на целую переменную
```

```
const int *pc1;    // указатель на значение, которое нельзя изменить
                   //(константу)
int * const cp2=&i;        // указатель-константа на целую переменную
const int * const cpc=&ci: // указатель-константа на целую константу
```

Как видно из примеров, модификатор **const**, находящийся между именем указателя и звездочкой, относится к самому указателю и запрещает его изменение, а **const** слева от звездочки задает постоянство значения, на которое он указывает. Для инициализации указателей использована операция получения адреса **&**.

Величины, типа указатель подчиняются общим правилам определения области действия, видимости и времени жизни.

Инициализация указателей

Указатели чаще всего используют при работе с динамической памятью, называемой некоторыми программистами кучей (перевод с английского языка слова **heap**). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти, называемым *динамическими переменными,* производится только через указатели. Время жизни динамических переменных - от точки создания до конца программы или до явного освобождения памяти. В C++ используется два способа работы с динамической памятью. Первый использует семейство функций **mal1oc (calloc)** и достался в наследство от C, второй использует операции **new** и **`delete`.**

При определении указателя надо стремиться выполнить его инициализацию, то есть присвоение начального значения. Непреднамеренное использование неинициализированных указателей - распространенный источник ошибок в программах. Инициализатор записывается после имени указателя либо в круглых скобках, либо после знака равенства.

Существуют следующие способы инициализации указателя:

- Присваивание указателю адреса существующего объекта:
    - с помощью операции получения адреса:

```
int a=5;  // целая переменная
Int* p=&a         // в указатель записывается адрес a
int* p (&a); // то же самое другим способом
```

- с помощью значения другого инициализированного указателя:

```
int* r = p:
```

- с помощью имени массива или функции, которые трактуются как адрес (будет подробно изложено дальше):

```
int Ь[10]:
int* t=b;  // присваивание адреса начала массива
void f(int a )  {(/* ... */ }      // определение функции
void (*pf)(int); // указатель на функцию
pf=f;        // присваивание адреса функции
```

2. Присваивание указателю адреса области памяти в явном виде:

```
char* vp  = (char *) 0xB8000000;
```

Здесь 0xB8000000 - шестнадцатеричная константа, (cha r *) - операция приведения типа: константа преобразуется к типу «указатель на char».

Пример.

Для примера определим указатель - константу **key_byte** и свяжем его с байтом, отображающим текущее состояние клавиатуры ПЭВМ IBM PC:

```
char * const key_byte =(char *)1047;
```

Значение указателя невозможно изменить, он всегда показывает на байт с адресом 1047 (шестнадцатеричное представление 0x417). Это так называемый байт состояния клавиатуры. Так как значение указателя-константы изменить невозможно, то имя указателя-константы можно считать наименованием конкретного фиксированного адреса участка основной памяти. Содержимое этого участка с помощью разыменования указателя-константы в общем случае доступно как для чтения, так и для изменений. Следующая небольшая программа демонстрирует это.

```
#include <iostream.h>
void main()
{char *const key_b=(char *)1047;
cout<<"\n Байт состояния клавиатуры - "<< *key_b;
*key_b='Ё';
cout<<"\n Байт состояния клавиатуры - "<< *key_b;
}
```

3. Присваивание пустого значения

```
int* suxx = NULL
int* rulez = 0;
```

В первой строке используется константа NULL, определенная в некоторых заголовочных файлах C как указатель, равный нулю, хотя можно использовать и просто 0 (см. 2 строку)

4. Выделение участка

- с помощью операции new, синтаксис:

new имя_типа

или

new имя_типа (инициализатор)

```
int* n = new int;      //1
int* b = new int (10);        //2
int* q = new int [10];        //3
```

Операция позволяет выделить и сделать доступным участок в основной памяти, размеры которого соответствуют типу данных, заданного **имя_типа**.

- с помощью функций mal1oc() и calloc(). Синтаксис:

имя_указателя=(тип_указателя)malloc(объем_в_байтах);

- объем_в_байтах - количество байтов ОП, выделяемое адресуемому зна-

чению.

```
int* u = (int *)malloc(sizeof (int));                //4
```

имя_указателя=(тип_указателя)calloc(число_элементов, размер_элемента_в_байтах);

- **число_элементов -** фактическое число элементов, под которые выделяется память;
- **размер_элемента_в_байтах -** размер одного элемента в байтах.

```
int* u = (int *)calloc(1, sizeof (int));             //5
```

Для работы с функциями malloc() и calloc() необходимо директивой препроцессора #include подключить файл alloc.h

В операторе 1 операция **new** выполняет выделение достаточного для размещения величины типа **int** участка динамической памяти и записывает адрес начала этого участка в переменную **n**. Память под саму переменную **n** (размера, достаточного для размещения указателя) выделяется на этапе компиляции.

В операторе 2, кроме описанных выше действий, производится инициализация выделенной динамической памяти значением 10.

В операторе 3 операция **new** выполняет выделение памяти под 10 величин типа int (массива из 10 элементов) и записывает адрес начала этого участка в переменную **q**, которая может трактоваться как имя массива. Через имя можно обращаться к любому элементу массива. О массивах рассказывается в следующем разделе.

В операторе 4 делается то же самое, что и в операторе 1, но с помощью функции выделения памяти **mal1oc()**, унаследованной из библиотеки С. В функцию передается один параметр - количество выделяемой памяти в байтах. Конструкция **(int*)** используется для приведения типа указателя, возвращаемого функцией, к требуемому типу. Если память выделить не удалось, функция возвращает 0.

Операции с указателями

С указателями можно выполнять следующие операции:

- разыменование, или косвенное обращение к объекту (*),

- присваивание,
- сложение с константой,
- вычитание,
- инкремент (++),
- декремент (--),
- сравнение,
- приведение типов.

Операция разыменования

Операция разадресации, или разыменования, предназначена для доступа *к* величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины (если она не объявлена как константа):

```
char a;                 // переменная типа char
char * p = new char;    /* выделение памяти под указатель и под динамическую переменную типа char */
*p = 'Ю'; a = *p;  // присваивание значения обеим переменным
```

Как видно из примера, конструкцию ***имя_указателя*** можно использовать в левой части оператора присваивания, так как она является L-значением, то есть определяет адрес области памяти. Для простоты эту конструкцию можно считать именем переменной, на которую ссылается указатель. С ней допустимы все действия, определенные для величин соответствующего типа (если указатель инициализирован). На одну и ту же область памяти может ссылаться несколько указателей различного типа. Примененная к ним операция разыменования даст разные результаты.

Пример.

```
//адреса и указатели
#include <iostream.h>
```

```
void main()
{ unsigned long L=0x12345678;
char *cp=(char *)&L;
int *ip=(int *)&L;
long *lp=(long *)&L;
cout<<hex;
cout<<"\n Адрес L=" <<&L;
cout <<"\n cp= "<<(void *)cp<<"\t*cp= 0x"<<(int)*cp;
cout <<"\n ip= "<<(void *)ip<<"\t*ip= 0x"<<*ip;
cout <<"\n lp= "<<(void *)lp<<"\t*lp= 0x"<<*lp;
}
```

На экран возможен следующий вывод:

```
Адрес L=0x8fe10ffc
cp= 0x8fe10ffc          *cp= 0x78
ip= 0x8fe10ffc          *ip= 0x5678
lp= 0x8fe10ffc          *lp= 0x12345678
```

Обратите внимание, что значения указателей совпадают и равны по адресу переменной **L**.

Примечание.

Пример выполнен в системе Турбо Си, поэтому длина типа **int** два байта.

В программе при выводе результатов в поток **cout** (по умолчанию он связан с экраном дисплея) использован новый для нашего изложения элемент - манипулятор **hex** форматирования выводимого значения. Этот манипулятор обеспечивает вывод числовых кодов в шестнадцатеричном виде (в шестнадцатеричной системе счисления).

В программе потребовалось явное приведение типов. Так как адрес **&L** имеет тип **unsigned long ***, то при инициализации указателей его значение явно преобразуется соответственно к типам **char ***, **int ***, **long ***. При выводе значений указателей они преобразуются к типу **void ***, ибо нас не интересуют длины участков памяти, связанных со значениями указателей.

При выводе значения ***cp** использовано явное преобразование типа **(int)**, т.к. при его отсутствии будет выведен не код (0x78), а соответствующий ему символ '**x**' кода ASCII. Еще один неочевидный результат выполнения программы связан с аппаратными особенностями IBM PC - размещением числовых кодов в памяти, начиная с младшего адреса. За счет этого пары младших разрядов шестнадцатеричного числового кода размещаются в байтах памяти с меньшими адресами. Именно поэтому ***ip** равно **0x567**8, а не **0x1234** и ***cp** равно **0x78**, а не **0x12**. Сказанное иллюстрирует рис. **.

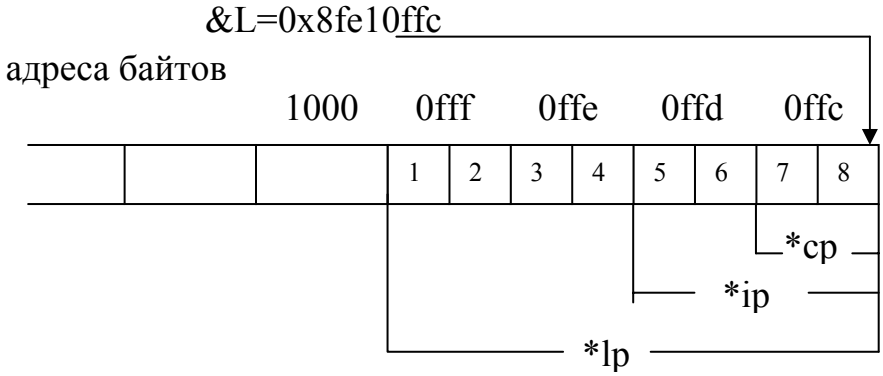


Рис. **. Схема размещения в памяти ПЭВМ IBM PC переменной L типа unsigned long

Приведение типов для указателей

Синтаксис операции явного приведения типа прост: перед именем переменной в скобках указывается тип, к которому ее требуется преобразовать. При этом не гарантируется сохранение информации, поэтому в общем случае явных преобразований типа следует избегать.

При смешивании в выражении указателей разных типов явное преобразование типов требуется для всех указателей, кроме void*. При использовании указателей

этого типа операция преобразования типов выполняется по умолчанию. В отличие от других типов, тип **void** предполагает отсутствие значения. Указатель типа **void *** отличается от других указателей отсутствием сведений о размере соответствующего ему участка памяти. Указатель типа **void *** как бы создан «на все случаи жизни», но как всякая абстракция, ни к чему не может быть применен без конкретизации, которая в данном случае заключается в приведении типов.

Присваивание без явного приведения типов допускается в двух случаях:

- указателям типа void*;
- если тип указателей справа и слева от операции присваивания один и тот же.

Таким образом, неявное преобразование выполняется только к типу void*. Значение 0 неявно преобразуется к указателю на любой тип. Присваивание указателей на объекты указателям на функции (и наоборот) недопустимо. Запрещено и присваивать значения указателям-константам, впрочем, как и константам любого типа (присваивать значения указателям на константу и переменным, на которые ссылается указатель-константа, допускается).

Арифметические операции с указателями

Арифметические операции с указателями (сложение с константой, вычитание, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, последовательно размещенными в памяти, например, с массивами.

Инкремент перемещает указатель к следующему элементу массива, *декремент - к* предыдущему. Фактически значение указателя изменяется на величину sizeof(тип). Если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа, например:

```
short * p = new short [5]:
```

```
p++; // значение p увеличивается на 2
 long * q = new long [53];
q++ // значение q увеличивается на 4
```

Разность двух указателей — это разность их значений, деленная на размер типа в байтах (в применении к массивам разность указателей, например, на третий и шестой элементы равна 3). **Суммирование двух указателей не допускается.**

Пример.

```
//для поддержки ввода-вывода
#include <iostream.h>
//для задержки в программе с пом getch() и очистки экрана
#include <conio.h>
void main()
{ double a1=10.1, a2=20.2;
  double* pa1=&a1, *pa2=&a2;
  clrscr();    //очистка экрана
  cout<< "\n&a1-&a2= "<<&a1-&a2;
  cout<<"\npa1-pa2= "<<pa1-pa2;
  cout<<"\n(int)&a1-(int)&a2= "<<(int)&a1-(int)&a2;
  getch(); //задержка до нажатия любой клавиши
}
```

Вывод на экран может иметь следующий вид:

```
&a1-&a2= 1
pa1-pa2= 1
(int)&a1-(int)&a2= 8
```

Из результатов видно, что определенные последовательно объекты a1 и a2, имея тип double, размещаются в памяти рядом на «расстоянии» 8 байт. Однако, разность адресов и указателей (&a1-&a2 == pa1-pa2) равна 1.

При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе:

```
*p++ = 10:
```

Операции разыменования и инкремента имеют одинаковый приоритет и выполняются справа налево, но, поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе **p**, будет записано значение **10**, а затем указатель будет увеличен на количество байт, соответствующее его типу. То же самое можно записать подробнее:

```
*p = 10; p++:
```

Выражение (*p)++, напротив, инкрементирует значение, на которое ссылается указатель.

Унарная операция получения адреса & применима к величинам, имеющим имя и размещенным в оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой перемен ной. Примеры операции приводились выше.