

Технический Университет Молдовы

Сергей Г. Истрати

**Примечание ! Содержание этой книги переведено с румынского на русский язык не совсем компетентными “переводчиками”. Прошу не считать этот перевод официально-правильным. Советую использовать румынский оригинал с сайта или из библиотеки.**

## ПРОГРАММИРОВАНИЕ

Введение в языки С и С++

*Цикл методического курса предмета «Программирования»*

Кишинев 2004

## Примечание

Данная работа предназначена для студентов-заочников I и II курсов дневного обучения Технического Университета Молдовы, которая изучает предмет «Программирования» и в особенности для студентов Факультета Радиоэлектроники и Телекоммуникаций кафедры Оптоэлектронные Системы специализации 1871 Инженерии и Менеджмента в Телекоммуникациях и 2105 Оптоэлектронные Системы.

Автор: преподаватель Сергей Г. Истрати

Ответственный редактор: лектор унив., доктор Павел Нистирюк

Рецензент: академик, доктор Виктор И. Борщевич

Т.У.М. 2003

## Содержание

Введение	6
1.Алфавит языка	8
2.Структура программы	13
3.Типы данных	18
3.1.Простые predetermined типы данных. Константы	19
3.1.1.Целочисленные константы	19
3.1.2.Вещественные константы	22
3.1.2.Символьные константы	24
3.1.3.Последовательность символов	25
4.Переменные	25
4.1.Имена переменных (идентификаторы)	25
4.2.Описание переменных	26
4.3.Инициализация переменных	26
5.Операции и выражения	27
5.1.Арифметические операции	27
5.2.Операция «присваивание»	28
5.3.Операции инкремент/ декремент	28
5.4.Отношения и логические операции	29
5.5.Логические позиционные операции	32
5.6.Операция размера	34
5.7.Операция «запятая»	35
5.8.Условные выражения	36
5.9.Конверсии типов	36
5.10.Преимущества операций	39
6.Операторы	40
6.1.Типы операторов	41

6.2.Операторы выражения	42
6.3.Операторы разветвления (условные)	43
6.3.1.Оператор разветвления IF и IF-ELSE	43
6.3.2.Оператор безусловного скачка GOTO	47
6.3.3.Оператор отбора SWITCH	48
6.3.4.Оператор прерывания BREAK	50
6.4.Многократные (циклические) операторы	52
6.4.1.Циклический оператор FOR	52
6.4.2.Циклический оператор WHILE	54
6.4.3.Циклический оператор DO_WHILE	56
6.4.4.Оператор продолжения CONTINUE	58
7.Массивы	58
7.1.Описание массивов	58
7.2.Доступ к элементам массива	60
7.3.Инициализация массивов	62
7.4.Примеры изучения массивов	64
8.Последовательность признаков	66
8.1.Массивы последовательности	70
9.Структуры в C/C++	71
9.1.Объявление переменных структурного типа	72
9.2.Введение переменных дополнительного типа	74
9.3.Использование структур	75
9.4.Составляющие структуры	76
9.5.Массивы структур	77
10.Функции в C/C++	80
10.1.Передача параметров функции	84
10.2.Возврат размеров функции	87

10.3.Прототип функции	90
10.4.Локальные переменные и область видимости	91
10.5.Глобальные переменные	95
10.6.Столкновения локальных и глобальных переменных	97
11.Указатели	98
11.1.Указатели и функции	103
12.Картотека в C/C++	110
12.1.Открытие картотек	112
12.2.Функции внесения / считывания картотек	118
12.2.1.Внесение / считывание признаков	118
12.2.2.Внесение / считывание последовательностей	120
12.2.3.Вход / выход с форматом	123
12.2.4.Картотеки и структуры	125
Приложение1.Функции входа/выхода в C/C++	129
Приложение2.Математические функции	142
Приложение3.Функции, использованные в обработке последовательности строк	155
Библиография	171

## Введение

Цель данной работы – ознакомление студентов с главными методами программирования в языках С и С++.

Язык программирования С был создан Дэнисом М. Риччи в 1972 и детально описан в книге «Язык программирования С» Риччи и Брайном Б. Керниганом. Существование языка в соответствии с правилами, описанными в книге, носит название «Стандарт С К&R» и представляет реализацию минимального стандарта. В 1983 был создан новый стандарт С Американским Национальным Стандартным Институтом, названным «Стандарт-ANSI-С». Позже был разработан язык С++, как производная языка С.

В таком виде язык С++ обладает множеством возможностей языка с более глубокими возможностями.

Данная работа содержит как описание инструментов языка ANSI-С, которые выполнены компиляторами С++, так и описание инструментов программирования языка С++.

Так же в работе уделено большое внимание практическим примерам решения различных типов задач с детальным изучением.

Представленная работа – часть серии дидактико-методических работ, открытых высшим преподавателем – Сергеем Г. Истрати, направленных на процесс оптимального обучения студентов предмету программирования. Были созданы следующие работы:

- Серия лекционных курсов по программированию. Язык Паскаль.
- Серия лекционных курсов по программированию. Язык С (данная работа).
- Методическое руководство по выполнению лабораторных работ.
- Методическое руководство по выполнению индивидуальных работ.

- Методическое руководство по выполнению курсовых работ.

Все эти работы могут быть найдены в Интернете по адресу  
[www.istrati.com](http://www.istrati.com)

## Язык программирования С.

### 1. Алфавит языка.

Языком программирования называется язык, посредством которого возможна передача компьютеру метод решения определенной задачи. А методом решения задачи, как известно, называется алгоритм. Вся информатическая теория в действительности занимается разработкой новых компьютеров, языков программирования и алгоритмов. Задача любого языка высшего уровня – предоставить нам наиболее удобный синтаксис, благодаря которому можем описать данные, с которыми работает наша программа, и операторы, которые должны быть выполнены для решения определенной задачи.

В языке программирования С, как и в любом другом языке программирования, есть свой алфавит и спецификация употребления символов. Алфавитом языка программирования называется набор символов, разрешенных для употребления и признанных ЭВМ, с помощью которых могут быть образованы величины, выражения и операторы этого языка программирования. Алфавит любого языка программирования содержит самые простые элементы с лингвистическими символами, а синтаксис языка определяет, каким образом комбинируются элементы словаря для получения правильных фраз (инструкции, выдержки из инструкции, объявления типов, переменных, констант, функций, процедур и т.д.). Элементы словаря составлены из символов. Любой символ представлен компьютером в единственном виде одним натуральным числом, находящимся между 0

и 127, называемым кодом ASCII. Большинство элементов алфавита языка программирования C можно разделить на 5 групп:

1) Символы, используемые для создания идентификаторов и ключевых слов. В содержание этой группы входят заглавные и строчные буквы латинского алфавита (английского) и знак подчеркивания «\_». Следует заметить, что одни и те же заглавные и строчные буквы (прим.: А и а) понимаются, как различные знаки, т.к. у них различные коды ASCII.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A b c d e f g h i j k l m n o p q r s t u v w x y z \_

2) Строчные и заглавные буквы русского алфавита (или алфавита другого языка) и арабские цифры:

А Б В Г Д Е Ё Ж З И Й К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э  
Ю Я

А б в г д е ё ж з и й к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я

0 1 2 3 4 5 6 7 8 9

3) Специальные символы, употребляемые для вычислений и передачи компилятору ряда операторов .

Таблица 1. Специальные символы

Знак	Название	Знак	Название
,	Запятая	)	Круглая открытая скобка
.	Точка	(	Круглая закрытая скобка
;	Точка с запятой	}	Открытая фигурная скобка
:	Двоеточие	{	Закрытая фигурная скобка
?	Знак вопроса	<	Меньше
'	Апостроф	>	Больше
!	Восклицательный знак	[	Квадратная открытая скобка
	Вертикальная черта	]	Квадратная закрытая скобка
/	Слеш	#	Диез
\	Обратный слеш	%	Процент
~	Деструктор	&	Амперсанд
*	Звездочка	^	Логическое

			отрицание
+	Плюс	=	Равно
-	Минус	“	Кавычки

4) Символы управления и разделения. В состав этой группы входит область (blank), табуляционные символы, символы перехода в новый ряд, возврат каретки, новая линия и новая страница. Эти символы предназначены для разделения тем, определяемых пользователем, таких как константы и идентификаторы. Последовательность символов разделения воспринимается компилятором, как единственный символ. (Прим.: несколько последовательных пространств).

5) Наряду с группами символов, обнаруженных языком C, широко используется последовательность управления, т.е. комбинации специальных символов, употребляемых в функциях ввода или вывода информации. Последовательность управления составлена обратной косой чертой (\), которая обязательно находится на первом месте, после которой следует комбинация из латинских букв и цифр.

Таблица 2. Последовательность управления

Последовательность управления	Название	Десятичный эквивалент
\a	Звуковой сигнал	007
\b	Возвращение на шаг	008
\t	Горизонтальная табуляция	009
\n	Перевод	00A

	строки(новая строка)	
\v		00B

Последовательность видов \ddd и \ xddd (здесь через d обозначена любая цифра) позволяет написать компьютерный код, как последовательность восьмеричных и шестнадцатеричных цифр. Например, символ перевода каретки может быть трактован по-разному: \r- главная последовательность управления, \015 - восьмеричная последовательность управления, \x00D - шестнадцатеричная последовательность управления.

Кроме того, в языке зарегистрированы дежурные слова, называемые ключевыми, которые могут быть употреблены в строго определенном смысле: int, float, double, char, long, signed, unsigned, const, volatile, sizeof, if, else, goto, case, default, for, while, do, break, continue, near, for, void, return, pascal, cdecl, interrupt, auto, extern, static, register, union, enum, typedef, asm, \_cs, \_ds, \_es, \_ss, \_AH, \_AX, \_BX, \_BL, \_CH, \_CL, \_CX, \_DX, \_DL, \_BP, \_DI, \_SI, \_SP.

Ключевые слова определяют семантический смысл операторов C. Ключевые слова начинаются со знака \_ (подчеркивание), употребляются для доступа к отрезкам данных и к журналам ЭВМ.

Под синтаксисом языка программирования подразумевается, вообще, совокупность правил агрегирования лексических единиц для формирования более сложных структур (операторов, заявлений, программ и т.д.). Структура программы языка программирования C так же обладает своими правилами синтаксиса, как например: штамп, сопровождаемый второстепенными функциями, после которых следует тело главной функции (которая содержит в своем составе

декларативную часть). Для подробного описания этих компонентов необходимы, конечно, и другие правила.

## 2. Структура программы.

Для получения полного представления программы в С проанализируем конкретный пример. Предположим, что существует одномерный массив  $x$  с  $n$ -м количеством элементов целого типа и необходимо составить программу, которая вычисляет и выводит сумму элементов целого массива. Это типичная задача для обработки одномерных массивов.

В представлениях языка С каждый выявленный алгоритм осуществляется единичной программой, называемой функцией. Стиль программирования в С характеризуется стремлением выделить большое число функций, не очень объемных, таких, как обработка данных в этих функциях, не зависящих от остальных частей программы. Это делает программу достаточно понятной и дает возможность легкого введения изменений в некоторые функции не касаясь остальных. Из выделенных функций, единственная, с которой начинается выполнение программирования, называется главной и носит название – `main`. Все остальные функции носят произвольные названия, которые программист задает сам. Они могут быть записаны в первоначальную картотеку до функции `main` (в произвольном порядке) или могут находиться в различных картотеках на магнитной опоре. В последующей программе, которая осуществляет решение задачи входит функция `main()` и функция называемая `suma()`. Проанализируем эту программу:

```
#include<stdio.h>
```

```

#include<conio.h>
int suma (int y[10], int m){
int i, suma=0;
for (i=0; i<m;i++){
suma+=y[i]; } return (suma); }
void main (void) {
int w,n,i,x[10]; clrscr( );
printf(“Изберите значение массива n,10\n”);
scanf(“%d”,&n);
printf(“Изберите массив x[%d]\n”,n);
for(i=0;i<n;i++)
printf(“Выберите элемент x[%d]\n”,i);
scanf(“%d”,&x[i]); }
w=suma(x,n);
printf(“suma=%d\n”,w);
getch( ); }

```

Первые 2 строки: `#include<stdio.h>` и `#include<conio.h>` не являются инструкциями языка C. Символ «#» указывает, что они являются директивами процессора. Процессор осуществляет предварительную обработку текста программы до компиляции. В данном случае директивы дают нам представление, что в компилируемой картотеке необходимо введение информации из картотек системы TurboC `stdio.h` (Standart Input/ Output Header – раздел введения-удаления) и `conio.h` (Console Input- Output Header – раздел введения – удаления консоли).

Существование этих директивов условное, потому как в тексте программы употреблены присоединенные функции `printf( )` и `scanf( )`, информация о которых находится в указанных картотеках. Следующая

строка `int suma (int y[10],int m)` содержит объявление функции `suma()` с 2-мя параметрами: массив `y[10]` и простая целочисленная переменная `m`. Здесь необходимо отметить факт, что в языке C любая программа начинается с главной функции `main()`, вне зависимости от того, сколько вспомогательных функций содержится в программе. Имея ввиду этот факт, функция `suma()` будет проанализирована в то время, когда ее вызов будет осуществлен телом главной функции `main()`.

Программная строка: `void main (void)` определяет раздел главной функции под названием `main()`. Слово `void` перед функцией означает, что эта функция не будет возвращать значения в процессе ее выполнения. Круглые скобки, что следуют за `main()`, указывают компилятору, что она является функцией, а слово `void` в скобках – что функция не использует параметры. Пара фигурных скобок: первая, что открывается после `main()` и соответственно та, что закрывается после функции `getch()`; определяют операторы, которые составляют тело главной функции. В языке C пара фигурных скобок `{}` определяют последовательность операторов, находящихся в едином целом. Следующая строка содержит описание переменных, употребленных в главной функции `main()`: `int w,n, i,x[10]`; которые передают компилятору, что в программе будут употреблены целочисленные переменные `w,n` и `i` и массив `x` из 10 целочисленных элементов. После описания данных следует инструкция обращения к функции `clrscr()`, зачисляемой в библиотеку Turbo C.

Эта функция обладает назначением очищения экрана. После нее следует функция `printf()`, которая выводит комментарий на экран. В данном случае функция `printf()` (`“Выберите размер массива n<10\n”`); выводит на экран предложение об избрании величины массива и

символа `\n` перехода в другой ряд. Следующая функция `scanf(“%d”,&n);` функция ввода, которая осуществляет введение клавиатурой значений величины массива `n`. Символ `%d`

Определяет целочисленность считываемой величины, а символ `&` определяет адрес памяти, куда будет вписано значение `n`. Функция `printf(“Выберите массив x[%d]\n”,n);` так же выводит на экран комментарий и предлагает выбор значений элементов массива `x` с уже известным размером `n`. Следующая инструкция – цикловая инструкция `for`. Это сложная инструкция, задачей которой является повторение последовательности инструкций несколько раз с различными размерами параметров.

```
for(i=0;i<n;i++) {  
    printf(“Выбрать элемент x[%d]\n”,i);scanf(“%d”,&x[i]);}
```

Здесь слово `for` – резервируемое слово, `i` - параметр цикла, который изменяет свои значения от 0 до `n` с шагом « $\Rightarrow$ » 1 благодаря инструкции инкрементирования `i++`. Тело цикла, повторяемое `n` раз ограничено парой открытых и закрытых фигурных скобок и состоит из 2-х функций: первая `printf( )` , которая выводит комментарий для избрания значений текущего элемента массива, и вторая `scanf( )` , которая осуществляет занесение значений текущего элемента массива с клавиатуры в память. В этом случае, в конце выполнения цикла все элементы массива `x[n]` будут принимать значения, что создает возможность для вычисления общей суммы элементов массива. После выполнения цикловой оператора `for` следует оператор присвоения `w=suma(x,n);` В правой части этого присвоения находится функция `suma( )`. Именно величина этой функции после выполнения с

использованием местных параметров  $x$  и  $n$  будет присвоена переменной  $w$ . Проанализируем функцию `suma()`:

```
int suma (int y[10],int m ) {int i, suma=0;
for(i=0;i<m;i++) {suma+=y[i]; } return(suma); }
```

Как и при объявлении любой функции в языке C в первую очередь следует штамп функции:

`int suma (int y[10], int m)`, где слово `int` перед именем функции `suma` является видом значений, перешедших в функцию главной программы `main()`. Круглые скобки после имени функции определяют список формальных параметров, используемых функцией. Эти параметры представляют целочисленный массив  $y$  с длиной 10 элементов и целочисленной переменной  $m$ . Следует отметить, что во время вызова функции `suma()` из среды главной программы `main()`, значения актуальных параметров (в данном случае  $x$  – изучаемый массив и  $n$  – его величина) является присвоением формальных параметров функции (в нашем случае массив  $y$  и его значение  $m$ ) и необходимо выполнение следующего условия: актуальные параметры должны соответствовать по качеству, позиции и по виду с формальными параметрами. Открытая скобка после штампа функции и соответствующая ей закрытая, после оператора `return()`; определяет тело функции `suma()`, которая так же обладает размером объявления данных. Здесь `int i`, `suma=0`; объявляет местную целочисленную переменную  $i$  и выделяет функцию `suma()` со значением 0. Следующая инструкция – цикловая инструкция `for`, которая содержит в своем теле, ограниченном фигурными скобками единственную инструкцию – сложную инструкцию присвоения `suma+=y[i]`; эквивалентную инструкции `suma=suma+y[i]`; которая вычисляет сумму элементов массива. Цикл будет повторяться  $m$  раз с

различными значениями параметра  $i$ , где  $m$  – величина массива, т.е. количество элементов массива, а  $i$  – порядковый номер текущего элемента массива. Передача этого значения главной функции `main()` осуществляется оператором `return(suma);`. После этого оператора значение будет включено туда, куда была призвана функция `suma()`, в нашем случае это инструкция присвоения `w=suma(x,n);`

Итак, значение суммы элементов массива будет присвоено переменной `w`. `printf(“suma=%d\n”,w);` Последняя инструкция программы – позывная функции `getch()`, которая останавливает выполнение программы с целью выявления результата до того пока не будет выбрана клавиша Enter.

Так может быть описана главная структура программы в C, т.е. любая программа начинается с содержания функциональных библиотек, которые будут употреблены в программе, после этого следует объявление всех вспомогательных функций, используемых в программе, которые содержат: штамп функции, раздел объявления переменных, местных констант, после которых следует тело функции; после объявления всех вспомогательных функций следует тело главной функции `main()`, ограниченной парой фигурных скобок, которые содержат описание переменных, а так же операторы главной программы.

### 3. Типы данных.

Программа в языке C содержит описание действий, которые должны быть выполнены компьютером, и описание данных, которые обработаны этими действиями. Действия описаны с помощью операторов, а данные – с помощью объявлений (или определений). Под типом данных подразумевается множество значений, которые могут

быть присвоены одной переменной или константе. Типы данных в С могут быть подразделены на 2 категории: простые (элементарные) или сложные (структурные). В целом, типы данных доступно описаны программистом и являются специфическими в той программе, где появляются. Существуют еще типы элементарных данных с более общим интересом, называемые предопределенными типами, определение которых считается известным и не является задачей программиста.

### 3.1. Простые предопределенные типы данных. Константы.

Программа С содержит в качестве пояснения буквенные и численные значения. Такие значения, появляющиеся в программе, называются константами. Константа – это буквенные или численные значение, которое всегда определено и в течение программы остается неизменным. Вид константы определяется по форме ее внесения, а ее значение заключается в ней самой.

#### 3.1.1. Целые константы.

Целая константа – это число, внесенное в программу без десятичной доли и без степенного указателя. Целые константы в С могут быть : десятичными, восьмеричными и

шестнадцатеричными. Порядковая система констант опознаваема компилятором по форме их внесения. Если константа состоит из цифр 0...9 и первая цифра не является 0, тогда константа считается десятичной. Например: 123, 45, 37. Если константа состоит из

употребляемых цифр 0...7 и первая цифра – 0, тогда константа считается восьмеричной. Например: 045, 037. Если константа составлена с помощью цифр 0...9 и букв a...f или A...F и начинается с 0x или 0X, тогда константа считается шестнадцатеричной. Например: 0x45, 0x37. На этих примерах константы составлены из одних и тех же цифр и различных значений, которые определяются на базе порядковой системы.

Для употребления констант целого вида используются различные зарезервированные слова, которые определяют диапазон значений и объем памяти, сохраненной для константы.(Таблица 3)

Таблица 3. Диапазон целого типа

Вид	Объем памяти	Диапазон значений
int	2	-32768...32767
Short (short int)	2	0...255
Long (long int)	4	- 2 147 483 648...2 147 483 647
Unsigned int	2	0...65 535
Unsigned long	4	0...4 294 967 295

В зависимости от значения константы, компилятор выделяет в компьютере для ее представления 2 или 4 байта памяти. Для значений:-32768...32767 выделяется 2 байта, где первый выступает, как знак константы, а остальные 15 бит определяют ее значение. В этом случае константа является вида int (целого).Для значений 0...65535 выделяется 2 байта памяти, а все 16 битов определяют значение константы. Такая константа имеет вид unsigned int (целый без знака). Константы этого

диапазона, имеющие знак «- » , исследуются, как без знака, в которых применяется операция «единичный минус».

Для значений от  $-2\,147\,483\,648$  и до  $2\,147\,483\,647$  выделяется 4 байта, в которых первый бит растолковывается, как знак, а оставшиеся 31, как порядковое значение. Такие константы имеют вид `long` (долгий, длинный).

Для значений от 0 до  $4\,294\,967\,295$  выделяются 4 байта, все 32 бита которых растолковываются, как значение. Такая константа имеет вид `unsigned long` (долгий без знака). Для этого вида в негативных константах применяется операция «единичный минус». Аналогично выделяется память для восьмеричных и шестнадцатеричных констант, полученных в соответствующем десятичном диапазоне.

Константы, занесенные в программу и имеющие численность большую чем  $4294967295$  ведут к подзагрузке, хотя компилятор не осуществляет предупреждений, а в память заносятся внутренние биты усеченной константы. Программист обладает возможностью пояснить компилятору, что для любой константы необходимо вывести 4 байта и пояснить их толкование без знака (`unsigned`). Для этого используются специальные модификаторы, введенные после строчной буквы константы. Чтобы объяснить, что константа обладает видом `long`, необходимо ввести модификатор `L` или `l` (соответственно `l` или `l`).

В стандарте К&R С ввод констант имеет следующий вид:

`#define name value`, который размещается до функции `main()`, где `#define` – директива компилятора, `name` – имя константы, `value` – ее значение. Например: `#define K 35` или `#define зарплата 700`. Между директивой, именем и значением константы необходимо указание хотя бы одного пробела. В выше описанных примерах

подразумеваемый компилятор будет присваивать переменные K и зарплате значения целого вида, поскольку величины 35 и 700 обладают синтаксисом и форматом целочисленных констант.

Некоторые компиляторы C, содержащие стандарт ANSI-C создают возможность для ввода констант 2-х видов: первый – с использованием директивы #define, описанный выше. Вторым видом использует зарезервированное слово const для ввода константы, описания ее вида и присвоения переменных следующего вида:

const int name = value; который размещается после функции main(), где const – зарезервированное слово для ввода констант, int – зарезервированное слово для обозначения целочисленных констант, value – значение константы. Например:

```
main() {  
    const int K=35;  
    ...;  
    const int зарплата=700;  
    ...; }
```

### 3.1.2. Вещественные константы.

Вещественные константы представляют десятичные дробные величины, которые могут быть записаны в 2-х видах: вид с точкой и порядковый вид. Вещественная константа в виде с точкой обозначается, как десятичная дробь со знаком или без него, в которой целая и дробная части определяются точкой. Если знак константы пропущен, она считается положительной. Вещественная константа порядкового вида удобна для ввода очень больших либо очень маленьких значений. В C, как и большинстве других языков

программирования, для таких значений используются вещественные константы порядкового вида, которые имеют вид:

`mantisa_e_ordinul` или `mantisa_E_ordinul`

Среди этих обозначений в качестве мантиссы может быть введена либо десятичная целочисленная константа, возможно со знаком, который определяет знак «+». Например:  $7.32E +14=7.32* 10^{14}$ ;  $55.000000E-3= =0.055$ ;

Для распознавания констант целого типа используется много слов, которые распознают диапазон значений в памяти зарезервированной для константы.

Таблица 4. Диапазон целого типа

Вид	Объем памяти (байты)	Диапазон значений
Float	4	3.4E-38... 3.4E+38
Double	8	1.7E-308... 1.7E+308
Long double	10	3.4E-4932... 3.4E+4932

Вещественные константы обладают уровнем определенного приближения, который зависит от компилятора. В таком случае цифра 6.12345678912345 в диапазоне приближения для вида float будет трактована компилятором, как 6.123456, этот вид еще называется видом единичного приближения и имеет приближение 6 значений после точки. Вид double еще называется видом с двойным приближением и обладает приближенностью от 15-16 значений после точки. Синтаксис ввода констант вещественного вида следующий:

По стандарту K&R-C: #define PI 3.14

По стандарту ANSI-C: const float PI=3.14; включая и тот, что из K&R-C.

### 3.1.3.Символьные константы.

Символьная константа – это любой символ алфавита, взятый в апостроф. Пример: ‘символ’.

Значение символьной константы (char) может быть буквой, цифрой или другим символом клавиатуры. Для каждой символьной константы в памяти выделяется по одному байту. Большинство значений символьной константы таковы: заглавные и строчные буквы латинского алфавита, десять арабских цифр от 0 до 9 и специальные символы ! @ # \$ % ^ & \* ( ) \_ + = | \ / { } “ ‘ : ; ? > < . , ~ `

Существует 2 метода записи символов:

Первый метод: любой табличный символ кодов ANSI может быть представлен в форме символьной константы так: ‘\ddd’ или ‘\xННН’, где ddd – восьмеричный код, ННН – шестнадцатеричный код символов. Нули, стоящие перед кодом символа могут быть опущены.

Второй метод: Часто употребляемые символы, которые не выводятся на экран, не должны быть закодированы. В таком случае употребляются их резервные обозначения. Эти символы в C относятся к классу контрольных символов. Если контрольные символы встречаются, например, в строке удаления, тогда они вызывают ответное действие. Синтаксис ввода констант символьного вида следующий:

По стандарту K&R-C: #define Lit ‘C’.

По стандарту ANSI-C: const char Lit = ‘C’; включая и тот, что из K&R-C.

### 3.1.4. Последовательность символов.

Последовательность – это непрерывный ряд алфавитных символов, взятых в кавычки. В отличие от других языков программирования, язык C не содержит особых типов данных, которые не содержат символьную последовательность. Язык C оперирует последовательностями как работал бы со множеством символьных данных собранных в устройстве, называемом массив. Здесь каждый знак последовательности является отдельным компонентом массива вида `char`. Вид массива – это структурный вид C и будет изучен в разделе « Структурные виды».

*Заметка:* Некоторые компиляторы C и C++ содержат специальные типы данных для оперирования с последовательностью символов и состоит из библиотечных функций для обработки последовательностей.

## 4. Переменные.

Переменная – это величина, которая в процессе выполнения программы может получать различные значения. Для переменных программист должен определить обозначения свойственных характеристик, которые называются идентификаторами. Зачастую идентификаторы имеют символьные названия или просто имена.

### 4.1.Имена переменных (идентификаторы).

Именем переменной (идентификатором) является цепь алфавитных символов и цифр, которые начинаются с буквы или с символа подчеркивания. В Turbo C размещаются следующие символы для образования идентификаторов: заглавные буквы A...Z, строчные буквы a...z, арабские цифры 0...9, включая и символ подчеркивания «\_».

Сточки зрения длины имени отсутствуют какие-либо ограничения, для компилятора Turbo C имеют значение только первые 32 символа. Прим.: А, в, х, у, сумма, гамма, Text\_no, бета, a 1, b\_1.

#### 4.2. Описание переменных.

Все переменные программы должны быть описаны. Их описание выполняется, как правило, в начале программы с помощью оператора описания.

Инструкции описания имеют следующий общий вид записи:

tip v1, v2,...,vn; где tip – определяет вид значений, которые могут получать переменные v1,v2,...,vn. В качестве вида используется одно из уже известных ключевых слов: int, float, double или char. Точка с запятой являются знаком окончания инструкции. Между ключевым словом, что определяет вид переменной и листом имен переменных минимально необходим формуляр. Например:

```
main( ) {  
float зарплата, сумма, итого;  
int time ,count;  
char знак, буква ; ... }
```

#### 4.3. Инициализация переменных.

Приписывание переменной начального значения во время компиляции называется вводом. При описании переменных компилятор может сообщить о необходимости инициализации. Примеры операторов описания ввода переменных:

```
main() {  
char черта ='\'', буква ='Т';  
int год =2000, месяц =9;  
float альфа, бета, гамма =1,7e-12;... }
```

В последнем операторе описания введена только переменная гамма, поскольку на первый взгляд кажется, что все 3 сигнала « $\Rightarrow$ »  $1,7e-12$ . Поэтому лучше избежать смешения введенных и не- переменных в одну и ту же инструкцию описания.

Существуют случаи, когда использование невведенных переменных может привести к огромным ошибкам с логической точки зрения по вычислению определенных значений. Во время загрузки ЭВМ неиспользованные в оперативной системе ячейки памяти содержат совместные данные. Когда объявлена переменная и ей зарезервировано место в памяти, содержание этих ячеек памяти не изменяется до того, пока переменная не будет введена или до тех пор, пока ей не будет присвоено хотя бы одно значение. Поэтому, если переменная, которая не была введена или которой не было присвоено ни одно значение в течение программы, используется для вычисления определенного значения, результат будет неверным. Это дает возможность ввода любой переменной во время ее объявления. Данные введения предполагают присвоение значений 0 (ноль) для численных переменных и « » (пробел) для символьных или последовательного вида символов.

## 5.Операции и выражения.

### 5.1.Арифметические операции.

В качестве арифметических операций язык С использует следующее:

- |                    |     |
|--------------------|-----|
| 1) Сложение        | (+) |
| 2) Вычитание       | (-) |
| 3) Умножение       | (*) |
| 4) Деление         | (/) |
| 5) Взятие в модуль | (%) |

6) Унарный «+» (+)

7) Унарный «-» (-)

### 5.2.Операция присвоения.

Операция присвоения может быть простой и сложной, в зависимости от знака операции. Простая операция присвоения обозначается знаком = и употребляется для присвоения переменной значения определенного выражения. Пример:  $x=7$ ; В языке программирования С допускается употребление произвольного номера операции присвоения. Операции присвоения не обладают значимостью первой необходимости, в отличие от описанных ранее, и выполняются справа налево. Операция присвоения связывает переменную с выражением, присваивая первой значение последней, что заключается в праве знака присвоения.

Бинарные операции:  $+ - * / \% \gg \ll \& | ^$ , за которые отвечает сложная операция присвоения, которая обозначается как  $op=$ , где  $op$  — одна из вышеперечисленных операций, как например  $=$  и  $*=$  и т.д.

Сложная операция присвоения вида:  $\_переменная\_op\_выражение\_$  определяется такой простой операцией присвоения:  $\_переменная\_=_переменная\_op\_выражение\_$

Наприм.:  $v+=2$  равносильно  $v=v+2$ . Как правило, для сложной операции присвоения ЭВМ составляет более эффективную программу.

### 5.3.Операции инкремент (декремент).

Операция инкремент, обозначенная 2-мя знаками плюс, применяется по отношению к одной переменной и увеличивает значение на единицу. Операция декремент обозначается 2-мя знаками минус, уменьшая ее значение на единицу. Операция инкремент может

быть использована как префикс, так и как суффикс. Разница между ними заключается во времени замены значения переменной. Префиксная форма обеспечивает замену переменной до ее использования. Операции инкремент-декремент применяются по отношению к переменным упорядоченного вида (такого как `int`).  
Примеры:

Операция `x++`; обеспечивает автоувеличение переменной `x` на единицу. В выражении `s=y*x++`; автоувеличение переменной будет выполняться только после вычисления значения `s`.

А в выражении `s= y*x++`; сразу следует автоувеличение `x` и только после этого будет вычислено значение `s`.

И если для `m=5` напишем выражение `m+++2`, компилятор примет первые два плюса за суффиксную форму инкремента переменной `m`. Значение выражения будет `7`, и только после этого переменная увеличивается на один. Выражение `(i+j)++` ошибочное, т.к. операция инкремент обеспечивается только по отношению к имени переменной. Операция инкремент более предпочтительна, чем арифметическая.

#### 5.4.Отношения и логические операции.

Операции отношений: (сравнения)

- |                     |      |
|---------------------|------|
| 1) Больше           | (>)  |
| 2) Больше или равно | (>=) |
| 3) Меньше           | (<)  |
| 4) Меньше или равно | (<=) |
| 5) Равно            | (=)  |
| 6) Отлично          | (!=) |

Операции отношений образуют истинные или ложные значения в зависимости от того, в каком отношении находятся зависимые величины. Если выражение отношений истинно, тогда его значение равно 1, в противном случае – 0.

Операции отношений обладают меньшим преимуществом, чем арифметические и операции инкремент. Среди операций отношений первые 4 имеют такое же преимущество и выполняются слева направо. Последние 2 операции имеют еще меньшее преимущество и выражение получит значение 1 только тогда, когда первое и второе выражение истинны. Операции отношений могут быть использованы для базовых типов данных, за исключением последовательности символов. (Для сравнения последовательностей используются инкорпоративные функции). Пример выражения отношений:

$a > b$ ;

$(a + b) < 2.5$ ;

$7.8 \leq (c + d)$ ;

$a > b = 2$ ;

Выражение отношений  $a > b = 2$  всегда будет принимать значение 0 (ложно). Независимо от значения переменных подвыражение  $a > b$  нулевое или единичное значение и сравнение любого из этих значений с 2 даст нам ноль.

Таким образом очевидно, что в языке программирования Turbo C можно писать выражения, которые кажутся бессмысленными с точки зрения традиционных языков программирования.

Исследуем логические операции. Они, как правило, используются в качестве связи для единства двух и более выражений. Следующая таблица определяет логические операции в Turbo C:

Таблица 5. Логические операции:

Название логических операций	Обозначение в Turbo C
1.Соединение (И )	&&
2.Разделение (ИЛИ)	
3.Отрицание (НЕТ)	!

Если первое и второе выражение являются некоторыми выражениями, тогда:

Выр1 && выр2 – истинно, в случае, когда оба истинны

Выр1 || выр2 – истинно, в случае, когда хотя бы одно из выражений истинно

! Выр1 – истинно, когда выражение 1 ложно и наоборот.

Мы определили для выр1 и выр2 наиболее общие случаи, без указания, что это за выражения. Итак, имеем право написать: 5 && 2 . Значение этого выражения = 1 , т.к. в Turbo C ненулевое значение трактуется, как истинное, а нулевое, как ложное. Первенство логических операций выражается в нисходящем порядке, так : отрицание (!). соединение (&&). разделение (||). Логические операции уступают операциям отношений.

Вычисление логических операций, в которые входят только операции && оканчивается, если выявляется ложность применения следующей операции && . Исходя из определения семантики, операция &&, как только появляется ложное значение, не имеет смысла при продолжении вычислений. Аналогично для логических операций,

которые содержат только операции  $\|$  , вычисления оканчиваются одновременно с возникновением истинного значения. Примеры логических выражений:

$(5 > 2) \&\& 47$  – истинно;  $!(4 < 7)$  – истинно;  $4 < 7$  – истинно.

### 5.5. Поразрядные логические операции.

Поразрядные логические операции употребляются для работы с отдельными битами или с данными группами битов. Поразрядные операции не применяются к данным `float` и `double`. Поразрядные логические операции:

Таблица 6. Логические позиционные операции

Название логической позиционной операции	Знак операции в Turbo C
И позиционное	<code>&amp;</code>
ИЛИ позиционное	<code> </code>

Поразрядная операция И зачастую используется для выделения определенной группы битов.

Поразрядная операция ИЛИ применяется к каждой группе битов данных и , зачастую, используется для установления некоторых битов, например выражения:  $x = x | \text{mask}$  устанавливает в одном те биты  $x$ , которым соответствует один из  $\text{mask}$  . Не стоит путать поразрядные логические операции с логическими операциями `&&` и `\|` . Наприм.: выражение  $1 \& 2$  – имеет значение 0, а выражение  $1 \&\& 2$  – значение 1.

Так поразрядная ИЛИ осуществляет для каждой пары битов операцию сложения по модулю 2. Результаты выполнения операции

ИЛИ , исключительно поразрядной, определяются в следующей таблице.

Таблица 7.

Первый операнд	бит	Второй операнд	бит	Итоговый бит
0		0		0
0		1		1
1		0		1
1		1		0

ИЛИ исключительно поразрядная  $\wedge$

Перемещение влево  $\ll$

Перемещение вправо  $\gg$

Обратимость  $\sim$

Операция перемещения влево( $\ll$  ) и вправо ( $\gg$ ) выполняют перемещение первого операнда влево или вправо на одну бинарную позицию, определяемую вторым операндом.

Во время перемещения влево , биты, которые перемещаются, дополняются нулями. Наприм.:  $x \ll 2$  передвигается влево на 2 позиции, пополняя пустые позиции нулями, что равносильно умножению на 4. В общем виде передвижение  $x$  на  $n$  позиций влево равносильно умножению значений  $x$  на  $2$  в степени  $n$  .

Передвижение вправо величины без значения (unsigned) ведет к дополнению нулями свободных битов. В некоторых случаях передвижение вправо величины со знаком ведет к увеличению значения бита, в других же – биты дополняются нулями.

Turbo C различает значения битов и поэтому для положительных значений передвижение вправо на  $n$  позиций эквивалентно разделению на 2 в степени  $n$ .

Единичная операция обратимости ( $\sim$ ) [ тильда ] обращает каждую единичную позицию в ноль и наоборот – каждую нулевую позицию в единичную

### 5.6. Операция «размерность».

Единичная операция размерности, обозначенная ключевым словом `sizeof`, задает значение своему операнду в байтах. Операция размерности употребляется в виде:

`sizeof_ выражение` или `sizeof_ вид`

Значение операции размерности является целой константой, которая определяет предположительность результирующего вида выражения. Если в качестве операнда `sizeof`

`Используем_вид`, тогда мы получаем величину предметов указанного вида. В качестве `_вид` могут быть использованы те же виды предметов, которые используются для описания переменных.

Операция `sizeof` может быть использована везде, где допускается использование целой константы. Конструкции `sizeof_ выражение` и `sizeof_ вид` рассматриваются, как единое целое и, таким образом, выражение `sizeof_ вид-2` означает  $(sizeof( \_вид )-2)$ . Или например выражение `sizeof (a+b+c)+d` равносильно выражению `константа_ +d`, где константа обладает значением длительности вида результата `a+b+c`. Выражение `a+b+c` взято в скобки для указания необходимости длительности вида результата `a+b+c`, но не `a+b+c+d`.

## 5.7.Операция «запятая».

Операция «запятая» (,) служит для объединения некоторых выражений в одно единственное, и, таким образом, в Turbo C вводится определение выражения с запятой, которое имеет следующий вид: выражение\_,выражение\_,\_выражение\_,... Пара выражений, разделенная запятой, вычисляется слева направо. Вид и значение результата выражения с запятой является видом и значением крайнего правого выражения.

Например:  $k=a+b$ ,  $d=m+n$ ,  $5.2+7$  - это выражение с запятой, которое вычисляется слева направо. Ее значение 12.2 вида float. В процессе вычисления этого выражения  $k$  и  $d$  присваиваются соответствующие значения. Для выражения  $d=(k=5+2.5+3)$  значение переменной  $d$  будет крайне правого выражения-операнд.

Запятая в Turbo C употребляется в 2-х контекстах: как разделитель данных и как операция, которая определяет последовательное вычисление выражений. Поэтому допускается, например, такое выражение:

```
int a,b,c= (1,2,5),d;
```

где переменная  $c$  вводится, как постоянное выражение с запятой 1,2,5 и приобретает значение 5. Контекст операции «запятая» (разделитель или операция) компилятор «чувствует» после скобок. В скобках – операция «запятая», снаружи – разделитель.

В продолжении обнаружим, что аналогичная ситуация может возникнуть вместо действительных аргументов во время обращения к функции. Это обращение, содержащее 3 аргумента, где второй имеет значение 5, можно представить, например, следующим образом:

```
F(a, (t = 3,t+2),c);
```

## 5.8. Условные выражения.

Условные выражения обладают следующим условным видом:

$\text{выр1\_?}:\text{выр2\_}:\text{выр3\_};$

Значение условного выражения вычисляется таким образом: первым вычисляется  $\text{выр1\_}$ . Если оно отлично от нуля (является истинным), тогда вычисляется  $\text{выр2\_}$  и его значение будет значением целого выражения, в противном случае высчитывается  $\text{выр3\_}$ . Так операционные знаки ? и : определяет тройная операция, т.е. операция с 3-мя операндами. Например, для вычисления максимального  $z$  из  $a$  и  $b$ , целесообразно написать выражение:

$z = (a > b) ? a : b$ , где  $(a > b) ? a : b$  – условное выражение. Скобки в первом выражении не обязательны, т.к. приоритет тройного выражения ? : очень мал, еще более малозначителен приоритет операции присваивания. И все-таки желательно употреблять скобки, т.к. в этом случае условие визуально выделяется. Вышеупомянутое выражение можно записать немного более эффективно так:  $a > b ? (z = a) : (z = b);$

## 5.9. Типовые конверсии.

При написании выражений было бы кстати использование однородных данных, т.е. одноптипных переменных и констант. Так же, если в выражении сочетаются различные типы данных, тогда компилятор производит типовую автоматическую конверсию в соответствии с правилами, которые приводят к следующему: если операция производится над данными различного типа, тогда и те и другие данные приводят к высшему из этих 2-х типов. Такая конверсия

называется возрастанием типа. Последовательность упорядоченных типов от высших к низшим определяется в соответствии с внутренним присутствием данных и выглядит так: double, float, long, int, short, char. Модификатор unsigned измеряет типовой ранг соответственный знаку.

Для операции присвоения (сложной или простой) результат вычисления выражений справа приводит к виду переменной, которой присваивается это значение. В это время имеет место возрастание или убывание типа.

Возрастание типа присвоения производится, как правило, без потерь, в то время, как его понижение может существенно исказить результат по причине того, что этот элемент высшего типа не входит в зону памяти элемента внутреннего типа. Для сохранения точности вычислений при выполнении арифметических действий, все данные типа float преобразуются в double, что сокращает возможность повторной ошибки. Конечный результат преобразуется в тип float, если он обусловлен соответствующим оператором описания. Например, данные записаны так:

float a,b,c; и выражение  $a*b+c$

Для вычисления значения переменные a,b и c будут обратимы в double, а результат будет типа float. Так же, если данные записаны как: float a,b; double c; тогда результат  $a*b+c$  будет типа double по причине возрастания типа.

Помимо автоматической конверсии типов, осуществляемой компилятором Turbo C дает возможность программисту явного указания типа, которому необходимо задать определенное значение или выражение. Добиваемся этого, используя операцию конверсии типа, которая имеет следующий общий вид:

(\_тип\_)\_выражение

Использование одной из таких видов конструкций гарантирует, что значения выражения будут обратимы в вид, заключенный в скобки перед выражением. В этой операции в качестве \_типа\_ могут быть использованы те же ключевые слова, как и в инструкциях описания типа, наряду с допустимыми модификаторами. Исследуем 2 выражения в Turbo C:

$d = 1.6 + 1.7$      $d = (\text{int})1.6 + (\text{int})1.7$     с условием, что переменная  $d$  – целочисленная. В результате выполнения первого выражения значение переменной  $d$  будет 3, т.к.  $1.6 + 1.7$  не нуждается в конверсии типов и выдает результат 3 типа `float`, а убывание типа до `int`, реализованное во время выполнения операции присвоения, задает переменной  $d$  значение 3 по причине урезания дробной части. В результате вычисления выражения второе значение переменной  $d$  будет 2, т.к. оно указывает изложенную конверсию констант `float` в типе `int` до их сложения. Во время выполнения операции присвоения не будет иметь место типовая конверсия, т.к. тип переменной совпадает с типом выражения.

Операция типового приведения чаще всего используется в случаях, когда в контексте не предполагается автоматическая типовая конверсия. Например: для уверенности в правильности работы функции `sqrt` она нуждается в аргументе типа `double`. Поэтому в описании: `int n`; для извлечения квадратного корня из  $n$  необходимо писать: `sqrt((double)n)`;

Следует отметить, что во время приведения к необходимому виду преобразуется значение  $n$ , но не меняется ее содержание, т.к. она остается вида `int`.

## 5.10.Качества операций.

Качества операций исследовались в процессе их изложения. В данном параграфе описаны вышеизложенные операции и указан порядок их выполнения. Для сравнения качеств операций представим их в таблице, в порядке убывания качеств. Колонка таблицы «Порядок выполнения» определяет последовательность выполнения для операций с одними качествами. Например в выражении  $k=d+=b-=4$ ; последовательность выполнения операций будет определяться в порядке справа налево и в результате  $b$  уменьшится на 4,  $d$  увеличится на  $b-4$ ,  $k=d+b-4$ . Порядок оценки, описанный простой операцией присвоения, будет следующий:  $b=b-4$ ;  $d=d+(b-4)$ ;  $k=d$ ;

Таблица 8. Количество операций

Кол-во	Операционный знак	Тип операции	Порядок выполнения
1	()	Выражение	Слева направо
2	! ~ ++ -- - + (единичный ) sizeof (тип)	Единичные	Справа налево
3	* / %	Множественные	Слева направо
4	+-	Дополнительные	Слева направо
5	<< >>	Перемещение	Слева направо

6	< > <= >=	Связи	Слева направо
7	= !=	Связи (равенство)	Слева направо
8	&	И позиционная	Слева направо
9	^	Исключающая ИЛИ	Слева направо
10		Включающая ИЛИ	Слева направо
11	&&	И логическое	Слева направо
12		ИЛИ логическое	Слева направо
13	?:	Условие	Слева направо
14	== *= /= += -= &=  = >>= <<= ^==	Присвоение простое и сложное	Справа налево
15	,	Запятая	Слева направо

## 6.Операторы.

Под оператором в С подразумевается какой-нибудь ввод, который оканчивается точкой с запятой, значение которого определяет действие компилятора, во время обработки первоначального текста программы

или действия процессора во время исполнения программы. Операторы могут быть разделены на 2 группы:

- 1) Периодический оператор компилирования
- 2) Периодический оператор выполнения программы

К компиляторным операторам относятся операторы, которые характеризуют данные программы, а к операторам выполнения – операторы, которые определяют действия обработки данных, следуя данному алгоритму. Компилируемые операторы были обнаружены в разделах описания и объявления данных. Впоследствии будут изучены операторы выполнения.

Каждый оператор имеет свой синтаксис и семантику. Синтаксис передает метод правильного ввода конструкций оператора. А семантика – это описание действий, выполненных оператором.

### 6.1. Виды операторов.

Оператор представляет единство выполнения программы. Операторы могут быть простыми, сложными и блоками.

Простой называется оператор, который в своем наличии не содержит другого оператора. К простым операторам относятся: выражения продолжения `continue`, окончания `break`, возврата `return` безусловного скачка `goto`, которые поясним далее.

Сложным называется оператор, которая в своем наличии содержит другие операторы. К сложным операторам относятся условный оператор `if-else`, операторы цикла `for`, `while`, `do while` и оператор отбора `switch`.

Блоком называется ряд операторов, взятых в фигурные скобки ({}). Они выполняются последовательно в порядке их ввода в блок. В начале блока могут быть описаны внутренние переменные. В этом случае говорится, что переменные локализованы в блок, существуют и работают только внутри блока и теряются вне его.

В Turbo C сложный оператор и блок могут быть использованы где угодно, где допускается использование простого оператора. В продолжении, сложный оператор может содержать другие сложные операторы, а блоки могут содержать как сложные операторы, так и блоки.

Любой оператор может быть замечен идентификатором, называемым ярлыком. Ярлык отделяется от оператора двоеточием, и так, в общем виде оператор имеет вид: ярлык\_ : тело\_ оператора;

Ярлык употребляется только в случае использования в этом операторе безусловного скачка при помощи goto. В вышеописанном примере может отсутствовать ярлык\_ или тело\_ оператора или обе. В случае отсутствия тела оператора речь идет о пустом операторе, т.е. таком операторе, который не выполняет ни одного действия. В случае отсутствия оператора и наличия ярлыка, имеет место пустой ярлыковый оператор. Пример: Empty:: роль окончания будет играть правая фигурная скобка закрытия (}). Например: label: { k=a=b; k+=8; }

## 6.2. Оператор «выражение».

Любое выражение станет оператором, если оно будет оканчиваться (;). В таком случае оператор «выражение» будет обладать следующим видом: выражение;

Как выяснить, всякий ли оператор может быть ярлыком? Оператор «выражение» относится к классу простых операторов языка Turbo C. В результате выполнения в программе оператора «выражения» вычисляется значение выражения согласно операциям, которые определены в ней. Обычно может быть одна и более операций присвоения, и тогда оператор «выражение» в языке Turbo C имеет тоже значение и в остальных языках. Проанализируем отрывок из программы, где использован оператор «выражение»: Прим. 1: `int x,y,z; x=-3+4*5-6; y=3+4%5-6; z=-3*4%5-6/5;` В итоге  $x=11$ ,  $y=1$ ; а  $z=0$ . Напишем порядок выполнения этого отрывка в последовательности с качествами операции, используя круглые скобки:

Для `x=-3+4*5-6`:  $x=((( -3)+(4*5)-6);$

Для `y=3+4%5-6`:  $y=((3+(4\%5)-6);$

Для `z=-3*4%5-6/5`:  $z=((( -3)*4)\%(-6))/5);$

Прим.2: `int x=2,y,z; x*=3+2, x*=y=z=4;`

Здесь оператор «выражение» содержит выражение с запятой, состоящее из 2-х подвыражений. В первую очередь вычисляется первое подвыражение, после – второе. В итоге получим:  $z=4, y=4, x=40$ . Замечаем, что при объявлении `x` имеет значение 2: `x=(x*(3+2)*(y=(z=4)));`

### 6.3.Операторы разветвления (условные).

#### 6.3.1.Условные операторы IF и IF-ELSE.

В разветвленных структурах вычисления, некоторые этапы не всегда выполняются в одном и том же порядке. В зависимости от определенных проведенных условий во время вычислений для

выполнения избираются различные последовательности операторов. Для описания таких процессов в С используются условные операторы. Условный оператор избирает единственный оператор из собственных альтернатив, который потом и выполняет. Такие операторы `if` и `if-else`. Оператор `if` – сложный и его синтаксис допускает один из следующих форматов:

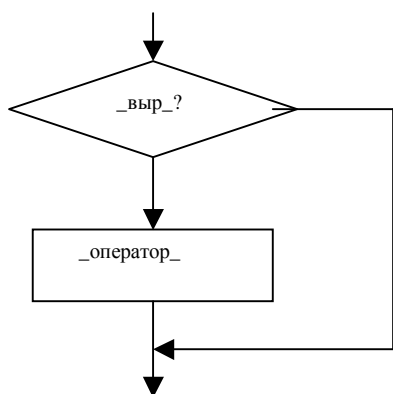
`if_(выражение)_оператор ИЛИ`

`if_(выражение)_оператор1_else _оператор2;`

В условных операторах отсутствует (;), т.к. конструкция `оператор_`, которая была описана уже включает в себя этот знак (;). Если `оператор_` простой, он включает в себя этот знак (;), а если оператор представлен блоком, тогда этот блок будет определен фигурными скобками, где правая скобка будет играть финальную роль. Оператор `if` работает в следующем порядке:

1. Формат `if(выражение)_оператор`. Сначала вычисляется значение выражения . Если его результат «ИСТИННО»

(т.е. `выражение !=0`), тогда выполняется `_ оператор`, в



противном случае оператор пропускается и нет ни одного действия. Арифметическая схема формата `if(выражение)_`

оператор представлена на рисунке. Пусть  $d$  равно  $c$ . Тогда увеличим  $d$  на 1, а  $c$  на 3. В остальных случаях  $d$  и  $c$  остаются неизменными. Условный оператор в этом случае:

```
if(d==c)++d, c+=3;
```

В качестве оператора\_ здесь используется оператор

«выражение»с запятой. Опишем этот оператор следующим образом:

```
if(d==c)++d; c+=3;
```

Разница заключается в том, что во втором примере у нас 2 оператора: `if` и оператор «выражение» `c+=3`;

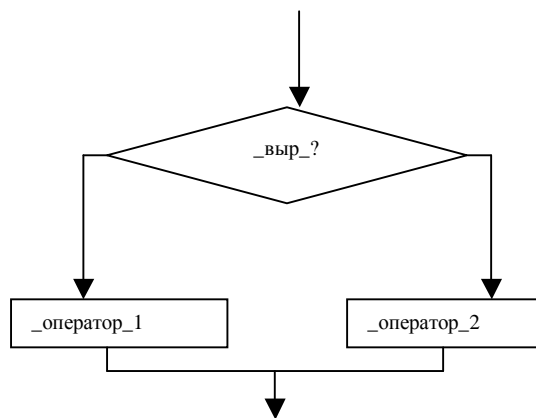
В этом случае, если  $(d==c)$  будет иметь истинное значение, тогда все остается без изменений:  $d$  увеличивается на 1, а  $c$  на 3. В противном случае  $d$  не изменится, а только  $c$  увеличится на 3.

Рассмотрим этот пример снова: `{++d; c+=3; }`

В случае, когда  $(d==c)$  будет иметь ложное значение,  $d$  и  $c$  останутся неизменными, т.к. эти операторы включены в фигурные скобки, т.е. образуют блок и, с точки зрения логики, рассматриваются как единственный оператор блок.

2. Формат `if(выражение)_оператор1_else_оператор2`.

Как и в предыдущем случае, в первую очередь вычисляется значение выражения. Если оно отлично от 0, т.е. истинно, тогда выполняется оператор1, в противном случае выполняется оператор2. Схематически этот оператор представлен на рисунке.



Например, пусть необходимо вычислить  $z$ , который равняется максимальному из 2-х чисел  $a$  и  $b$

Тогда можно написать: `if(a>b)z=a; else z=b;` Присутствие `(;)` после `z=a` необходимо, т.к это оператор, входящий в состав `if`. 0

Первая краткая форма оператора `if` дает возможность выполнения или не- определенной операции, в то время, как вторая предоставляет выбор и выполнение одной операции из многих. Проанализируем операторную цепь:

`If(выражение1)_оператор1_else_if(выражение2)_оператор2_else_if(выражение3)_оператор3_else_ оператор4...`

Такая цепь допустима, т.к. на месте оператора в операторе `if` может быть всякий оператор, включая и `if`. Это сложный ввод. Для таких процессов в C существует специальный оператор, который будет изучен чуть позже. Рассмотрим 2 примера:

а) `if (n>0) if (a>b)z=a; else z=b;`

б) `if (n>0) { (a>b)z=a; } else z=b;`

Различие состоит в том, что в примере а) оператор `if` в кратком виде, который выполняет оператор формы `if-else`; В случае б) оператор `if-else` в полной форме, имея в качестве оператора1 краткую форму `if`. Отрывок б) отличается от а) только наличием скобок, которые

определяют блок, к тому же, очевидно, играет большую роль в толковании этих операторов.

### 6.3.2. Оператор безусловного скачка GOTO.

Оператор `goto` дает возможность передачи контроля выполнения программы оператору, отмеченного ярлыком. Оператор `goto` имеет форму: `goto_ярлык;`

После данного оператора очевидно выполняется оператор, ярлык которого совпадает с ярлыком `goto`. Использование этого оператора в Turbo C не чем не отличается от использования в других логарифмических языках. Так же желательно наиболее сокращенное использование это, т.к. язык Turbo C к структурированным языкам. В тоже время, имея в распоряжении оператор `goto` и `if`, программист может выполнять операции любой сложности. Прим.: Использование операторов `if` и `goto` для образования циклов. Эта программа вычисляет значение  $y$ , которое равняется  $n/(n+5)$ , где  $n=1, \dots, 50$

```
#define lim 50
man( ) {
int n=0; float y=0;
m1:++n;
if(n<=lim) {
y+=n/(n+50); goto m1; }}

```

Первая скобка программы предназначена предпроцессору, которому указывается значение константы `lim=50`. Так поступают в случае необходимости изменения суммарности предела. Для этого необходимо

только изменение директивы #define, а соответственные подстановки в тексте программы процессор выполнит без нашего участия.

Программа состоит из одной функции main( ), тело которой определено с помощью наружных фигурных скобок. Инструкция if дана здесь в краткой форме, имея в качестве оператора – блок, содержащий оператор goto, что создает возможность для цикловых вычислений суммы.

В завершении отметим, что использование if и goto для образования циклов является знаком недостаточной культуры в программировании. Данный пример показывает только, как можно образовать цикл с помощью операторов if и goto , но ни в одном случае не служит правильным примером для копирования.

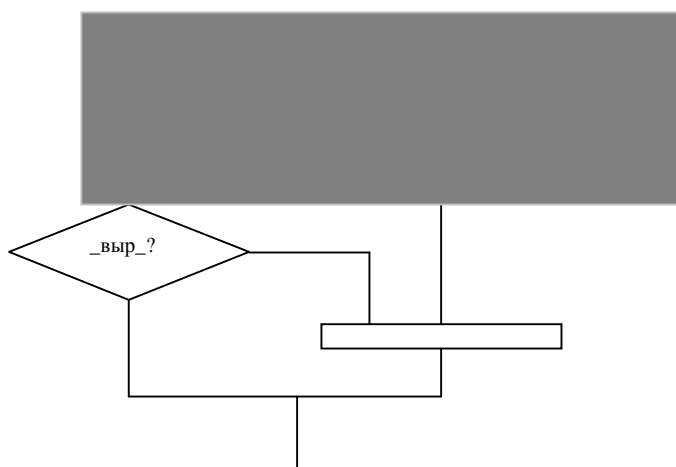
### 6.3.3.Оператор выбора SWITCH.

Оператор выбора предназначен для одного варианта из множества возможных, установленных программой. Любое условие может быть представлено с помощью операторной цепи if\_else\_if...\_else\_, однако в случае представления более 3-х условий рационально использовать SWITCH. Данный оператор выбора является сложным и имеет следующий вид:

```
switch (выражение) {  
  case выр_конст1:оператор1;  
  case выр_конст2: оператор2;  
  case выр_конст_n: оператор_n;  
  default: оператор ; }
```

После ключевого слова `switch` в скобках написано выражение, значение которого должно быть всегда `int` или `char`. Далее в фигурных скобках записываются операторы-варианты, выделенные префиксами: `case` выражение\_конст: где выражение константы так же должно быть целочисленным, а все префиксы различными. В одном варианте может находиться префикс `default`, присутствие которого не является обязательным, но в большинстве случаев этот префикс представлен конструкцией `switch`.

Оператор отбора работает в следующем порядке. В начале находится значение выражения в скобках, затем это значение сравнивается с постоянными выражениями `case` префиксов, и выполняется вариант `I`, для которого эти значения совпадают. После выполнения операторов, находящихся в соответствующих `case`-ах, либо выполняется оператор варианта с префиксом `default` (если `default` присутствует), либо не выполняется ни один из вариантов (если `default` отсутствует). Алгоритмическая схема выполнения оператора `switch` представлена на рисунке.



Изучаем пример применения оператора `switch`. Записывается отрывок программы, который печатает 4 стихотворные строки, начиная со строки `k`:

```
switch(k) {  
  case1: printf("A fost o data ca-n povesti,\n");  
  case2: printf("A fost ca niciodata,\n");  
  case3: printf(" Din rude mari imparatesti,\n");  
  case4: printf(" O preafrumoasa fata,\n");  
  default printf("Стихотворение не содержит такой строки»);}
```

В этом примере используется функция `printf( )`, обеспечивающая печать формативной строки. Если бы `k=3`, то результат был бы:

```
“ Din rude mari imparatesti  
O preafrumoasa fata.  
Стихотворение не содержит такой строки».
```

#### 6.3.4.Оператор прерывания `BREAK`.

В практике программирования иногда возникает потребность выполнения только одного варианта `case` без тех, что следуют за ним, т.е. необходимо логическое прерывание, установленное в работе оператора `switch`. Такой оператор - `BREAK` выполнение которого приводит к завершению оператора `switch`. В этом случае оператор `switch` будет обладать следующим видом:

```
Switch(выражение) {  
  case выр_конст1: оператор1; break;  
  case выр_конст2: оператор2;break;  
  case выр_конст_n: операторn ;break;
```

```
default: оператор; break; }
```

Оператор `break` записывается в вариантах тогда, когда в нем есть необходимость. Его выполнение ведет к переходу контроля к следующему оператору после `switch`. На первый взгляд кажется, что нет необходимости использования оператора `BREAK` после варианта `default`. `Default` может находиться в любом месте оператора `switch`, даже на первом месте среди вариантов, и тогда `BREAK` необходима для предотвращения выполнения остальных вариантов. Наприм., изменим формулировку предыдущей программы следующим образом: пусть необходима печать строки `k` из тех 4-х стихотворных строк:

```
switch (k) {  
  case1: printf(" A fost o data ca-n povesti,\n"); break;  
  case2: printf(" A fost ca niciodata,\n"); break;  
  case3: printf(" Din rude mari imparatesti,\n"); break;  
  case4: printf(" O preafrumoasa fata.\n"); break;  
  default: printf(" стихотворение не содержит такой строки \n"); }
```

Необходимо заметить, что операторы из вариантов `case` или `default` могут отсутствовать. Это необходимо, когда нам надо получить этот же результат при переходе на различные префиксы. Пример:

```
switch (L) {  
  case 'C':  
  case 'c': printf("Компьютер\n"); break; }
```

В том случае, когда `L='c'` будет напечатано слово «компьютер». В случае, когда `L='C'`

Будет выполнен оператор, следующий после первого `case`, но, т.к. здесь отсутствует оператор, будут выполнены операторы, находящиеся ниже, до того, пока не встретится оператор `BREAK`.

## 6.4. Многократные (циклические) операторы.

Вышеизложенные операторы передают операции, которые должны быть исполнены, следуя алгоритму, и каждая из них выполняется только однажды. В случае, когда один из этих операторов должен быть выполнен  $n$  раз с различными значениями параметров, используются циклические операторы в С:

- 1) Циклический оператор с параметром (FOR)
- 2) Циклический оператор предшествующий условию (WHILE)
- 3) Циклический оператор с поусловием (DO-WHILE)

### 6.4.1. Циклический оператор FOR.

Цикл FOR обладает следующей характеристикой:

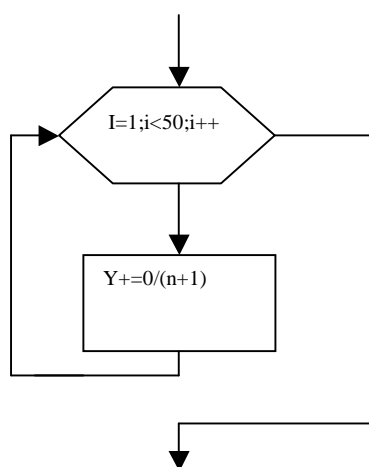
Число повторений цикла известно с начала его выполнения; управление циклом выполняется при помощи целочисленной переменной, называемой параметром цикла, который в данном циклическом процесс присваивает последующие значения от исходного, данного до финального значения. Синтаксис оператора следующий: `for(выражение1;выражение2;выражение3)` оператор; где выражение1 – первоначальное выражение параметра цикла, выражение2 – контрольное выражение, выражение3 – выражение инкремента/ декремента (коррекции) параметра цикла.

Данный цикловой оператор работает следующим образом: в начале вычисляется первоначальное выражение. Затем, если контрольное выражение правдиво, выполняется оператор. После выполнения

оператора осуществляется выражение коррекции и заново контролируется контрольное выражение, истинность которого ведет к повторному выполнению оператора. Если контрольное выражение обладает ложным значением, выполнение цикла `for` оканчивается, т.е. контроль передается оператору программы, следующей за оператором `for`. Например, вычислить:  $y = \sum_{i=1}^{50} \frac{1}{i(i+1)}$ ; где  $i=1 \dots 50$ ;

```
y=0; for (i=1;i<=50;i++) { y+=1/(i(i+1)) };
```

Здесь  $i=1$  – первоначальное выражение,  $i \leq 50$  – контрольное выражение,  $i++$  - выражение коррекции. Фигурные скобки определяют тело цикла (оператора). В случае, когда тело цикла состоит только из одного оператора, фигурные скобки необязательны. Алгоритмическая схема выполнения оператора показана на рисунке.



Из нее видно, что оператор `for` – цикл с условием: допускается либо выполнение цикла заново, либо он не учитывается перед началом выполнения, очевидно, может случиться, что тело цикла не будет выполнено ни один раз.

Иногда возникает необходимость выхода из цикла до окончания. Для этого в теле цикла, в том месте, где желается выход из него,

используется оператор **BREAK**, после выполнения которого имеет место передача контроля следующему оператору после цикла.

Язык **C** не ограничивает виды операторов в рамках тела цикла. В этом случае тело цикла может состоять из сложных и простых операторов, в частности, тело одного цикла может быть другим циклом. В некоторых алгоритмах появляются ситуации, когда необходимо проникновение одного цикла в другой. Например, при обработке матрицы внешний цикл отвечает за обработку строк, а другой, внутренний – столбцов. В этом случае синтаксис будет следующим:

```
For(i=1;i<=n;i++){  
  For (j=1;j<=m;j++) {  
    тело цикла } };
```

где  $n$  – количество строк в матрице,  $m$  – кол-во столбцов, внутренние фигурные скобки определяют тело цикла с параметром  $j$ , а внешние фигурные скобки – тело внешнего цикла с параметром  $i$ .

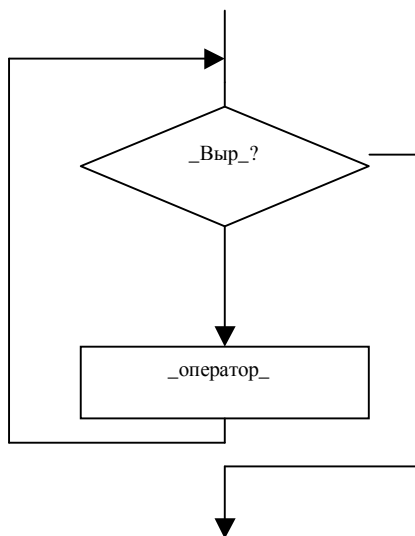
#### 6.4.2.Циклиеский оператор **WHILE**.

Цикл **WHILE** используется, когда не известно количество повторений цикла и нет возможности выполнения цикла хотя бы один раз. Цикловой оператор **while** обладает следующим видом:

```
while (выражение) оператор;
```

Циклический оператор **while** работает в следующем виде: Если выражение правдиво (или отлично от 0, что с точки зрения языка Turbo **C** одно и то же), тогда оператор выполняется однократно, и потом выражение заново тестируется.

Такая последовательность действий, что заключается в тестировании выражения и выполнении оператора, периодически повторяется до того, пока выражение не станет ложным (с точки зрения языка Turbo C будет равно 0). Оператор называется телом цикла и в большинстве случаев, представляет блок, в состав которого входит несколько операторов. Алгоритмическая схема существования оператора while:



Заметьте, что оператор WHILE – это цикл с условием. Контрольный текст выполнен до входа в тело оператора. Поэтому вероятно, что тело цикла не будет выполнено ни один раз. Кроме того, чтобы не допустить бесконечного повторения цикла, а только определенного кол-ва раз, при каждом новом выполнении цикла необходимо изменить переменную параметра, входящую в состав выражения. В отличие от цикла for, где переменная может быть только целочисленной, параметр цикла while может быть и вида float, т.е шаг цикла может быть отличен от 1 и даже дробным числом. Прим.:

```
i=1; while(i<=50) { y+=i(i+1);i++; };
```

Как и в случае цикла for тело цикла while может так же быть циклом. Например.:

```
i=1; while (i<=n) {j=1; while(j<=m) {тело цикла;j++; } i++ };
```

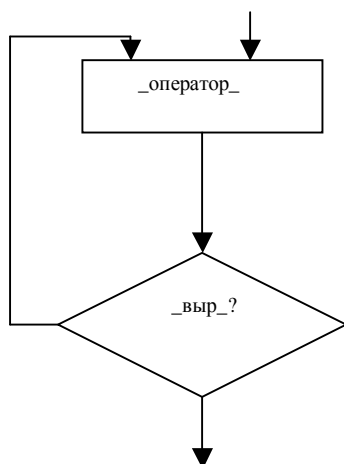
### 6.4.3.Циклический оператор DO\_WHILE.

Циклический оператор DO\_WHILE используется в случае, когда не известно число повторений цикла, но, в то же время необходимо, чтобы цикл был выполнен хотя бы один раз. Циклический оператор do\_while имеет следующий вид:

do оператор while (выражение);

Оператор DO\_WHILE работает следующим образом: сначала выполняется оператор , затем вычисляется значение выражения. Если значение выражения истинно, тогда оператор выполняется заново, если выражение ложно выполнение цикла оканчивается.

Алгоритмическая схема выполнения оператора do\_while представлена на рисунке.



Оператор do\_while определяет цикл с постусловием, т.к. контроль необходимости выполнения повторного оператора имеет место после выполнения тела цикла, и таким образом, тело цикла всегда

выполняется, как минимум единожды. Аналогично, как и для цикла `while`, программист должен заботиться об окончании цикла, изменяя его параметр в теле цикла. Параметр цикла `do_while`, как и параметр цикла `while` может быть вида `float`, что позволяет оперировать десятичными шагами в организации цикла.

Пример 1: `i-1; do { y+=i(i+1);i++;} while (i<=50);`

Цикл `do_while` так же может быть следующим:

```
i=1; do { j=1;  
do {тело цикла ;j++;} while (j<=m)  
i++;}  
while (i<=n);
```

Итак, известны эти 3 цикловых оператора `FOR`, `WHILE` `DO_WHILE`: можем употреблять их в различных целях отдельно, или вместе в определенных случаях. В случае, когда известно кол-во повторений цикла, шаг индекса =1, используем циклический оператор `FOR`. В случае, когда кол-во повторений цикла неизвестно, а тело цикла должно быть выполнено хотя бы один раз, используем циклический оператор `DO_WHILE`. И в случае, когда не известно кол-во повторений цикла, но тело цикла должно быть выполнено 0 или более раз, в зависимости от условия употребляем оператор `WHILE`. Более того, в случае составляющих циклов, возможна комбинация цикловых операторов:

`For_while; do_while- while; и т.д.`

#### 6.4.4. Оператор продолжения CONTINUE.

Оператор CONTINUE используется в теле цикла с целью передачи контроля в начале цикла. Существуют случаи, когда при выполнении некоторых условий, необходимо прервать текущие итерации и перейти к выполнению следующей итерации цикла. В таких случаях используется оператор continue. Оператор продолжения имеет следующий вид: continue;

Оператор CONTINUE может быть реализован во всех 3-х видах циклов. Но не в операторе SWITCH. Он служит для выхода за пределы оставшейся стороны текущей итерации цикла, что непосредственно содержит. Если условие цикла допускает новую итерацию, она выполняется, в противном случае – заканчивается. Разберем следующий пример:

```
int a,b; b=0
for (a=1;a<100;a++) { b+=a; if (b%2) continue;
... обработка парной суммы... }
```

В случае, когда сумма будет непарная, оператор продолжает передачу контроля следующей итерации цикла for без выполнения следующей части тела цикла, где имеет место обработка парной суммы.

## 7. Массивы.

### 7.1. Описание массивов.

Массив представляет упорядоченную последовательность элементов одного вида. То, что массив целочисленный и составлен из нескольких

элементов, позволяет нам назвать переменную вида массива, как переменную сложного вида.

Массив можно охарактеризовать именем, видом, размером. Общий формат описания массива:

вид имя [d1][d2]...[dn]; где:

Вид – общий вид всех элементов массива,. Вид массива может быть уже определенным типом данных: целочисленный, действительный, символьный.

Имя – имя массива. В качестве имени массива используется любой идентификатор. Кроме того, т.к. именем массива является идентификатор, над ним распространяется все то , что указано в разделе «Имена переменных (идентификаторов) ».

d1, d2, dn - размеры массива. Они определяют кол-во элементов, присутствующих в массиве. Размером массива может быть выражение «константа» с целочисленным результатом. В зависимости от кол-ва размера массивы классифицируются:

#### 1. Одномерные массивы (с 1 размером);

Одномерный массив представляет ряд равномерно упорядоченных элементов в одной строке. Каждый элемент одномерного массива имеет 1 координату; порядковый номер элемента в ряду.

#### 2. Двумерные массивы (с 2-мя размерами).

Представляет структуру, образованную из строк и столбцов. Каждый элемент двумерного массива обладает 2-мя координатами: номер строки и номер столбца.

#### 3. Трехмерные массивы (с 3-мя размерами).

Представляет структуру, равносильную кубу в объеме с 3-мя размерами: длина, ширина и высота. Каждый элемент трехмерного



доступа к любому элементу массива следующий: имя [i1][i2]...[in]. Где имя – имя массива, i1 – показатель элемента в размере1, i2 – показатель элемента в двукратном размере и n – показатель элемента в n-кратном размере. В самых частых случаях используются одно- и двумерные массивы. Доступ к элементу одномерного массива происходит следующим образом: имя[i], где имя – имя массива, I – порядковый номер элемента массива. Пример:

Vector[5]; допускается элемент с порядковым номером 5 из массива vector

Fraza[20]; допускается элемент с показателем 20 из массива fraza .

Доступ к элементу двумерного массива осуществляется через имя [i][j], где i – номер строки, в которой находится элемент, j – номер столбца, в котором находится элемент.

Пример: matrix[4][7]; допускается элемент 4 строки и7 столбца массива matrix

y[0][0]; допускается первый элемент массива, т.е. 0-ая строка, 0-ой столбец

В случае, когда массив является простого вида, присвоение значений элементу осуществляется, как и в случае присвоения значений простой переменной. Пример:

x[0]=7.125;

vector[19]+=1;

matrix[1][1]=5.5;

fraza[3]='b';

spase[3][5][2]=8;

В случаях массива структурного типа, присвоение значений и доступ к элементам массива происходит в соответствии с правилами присвоения и доступа для структурных переменных.

Элемент массива может возникать в любом выражении, где допустимо присутствие переменной вида, совмещенного с видом значения элемента.

### 7.3.Инициализация массивов.

Зачастую необходимо, чтобы элементы массива обладали значениями прямо во время описания массива.

Процесс присвоения значений элементов массива во время его описания называется инициализацией массива. Синтаксис инициализации одномерного массива:

Вид имя [d]={v0,v1,v2,...,vn-1}; , где вид – вид массива, имя – имя массива, v0,v1,v2,...vn-1- соответственные значения элементов имя[0], имя[1] и т.д. Прим.:

```
int x[8]={1,3,15,7,19,11,13,5};
```

В этом случае элементы массива будут обладать следующими значениями: x[0]=1; x[1]=3; x[2]=15; x[3]=7; x[4]=19; x[5]=11; x[6]=13; x[7]=5;

Очевидно, что показатели массива изменяются, начиная с 0. Т.к. при описании массива максимальное значение показателя массива совпадает с номером элемента массива минус один.

При инициализации массива не обязательно указывать значения массива. Компилятор определит кол-во элементов после описания массива и образует массив с соответствующим значением. Например:

```
int x[]={1,3,15,7,19,11,13,5};
```

Элементы массива будут принимать значение, так же, как и в предыдущем случае. Изучи еще некоторые примеры первоначальности массивов:

```
float vector [4]={1.2,34.57,81.9,100.77}; //vector - массив из 4 элементов типа float
```

```
int digit[5]={1,2,3}; // digit – целочисленный массив из 5 наименований, последним 2 элементам присваивается значение 0.
```

```
char m[5]={`A`,`B`,`C`,`D`}; //m – массив из 5 символов, последний элемент обладает значением ноль-символ.
```

```
float const y[4]={25,26,17,18}; //ввод массива y[4], элементы которого – константы типа float и не могут изменяться в течение выполнения программы.
```

Исследуем инициализацию двумерного массива:

```
int a[3][3]={ {1,4,2},  
             {7,5,3},  
             {8,6,9} };
```

Инициализация двумерного массива выполняется построчно. Элементы этого массива обладают следующими значениями:  $a[0][0]=1$ ;  $a[0][1]=4$ ;  $a[1][0]=7$ ;  $a[1][1]=5$ ;  $a[1][2]=3$ ;  $a[2][0]=8$ ;  $a[2][1]=6$ ;  $a[2][2]=9$ ;

При инициализации данного массива каждая строка заключается в фигурные скобки. Если в указанных нами строках не хватает элементов для составления строк, на месте элементов, которым не хватало значений появляется 0.

Если в данном примере опустить внутренние фигурные скобки, результат будет тот же. Если отсутствуют внутренние фигурные скобки, элементам присвоятся значения в последовательном порядке,

считанные с листа. Составление массива образуется построчно. Элементы массива, которым в листе не хватило значений, получают значения 0. Если в листе больше значений, чем элементов, тогда такой лист считается ошибочным. Выше изложенное относится ко всем видам массива. Примеры:

Инициализация двумерного массива:

```
int a[3][3]={1,4,2,7,5,3,8,6,9};
```

Три метода, эквивалентной инициализации трехмерных массивов:

```
int p[3][2][2]={ {1,2},{3,4} },{ {5,6},{7,8} },{ {9,10},{11,12} };
```

```
int p[3][2][2]= { {1,2,3,4}, {56,7,8}, {9,10,11,12} };
```

```
int p[3][2][2]= {1,2,3,4,5,6,7,8,9,10,11,12};
```

#### 7.4.Примеры обработки массивов.

Представим 2 метода обработки одно- и двумерных массивов.

Пример 1: Обработка одноименного массива.

Дан одномерный массив  $x$  с  $n$  элементов. Сравните сумму первой половины массива со средним арифметическим второй его половины:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
void main (void){
int x[20],n,k,i,s=0;
float m,s1=0,r=0;
printf("\nИзберите величину массива n<=20\n");
scanf("%d",&n);
```

```

for(i=0;i<n;i++){
printf(“изберите элемент %d\n”,i);
scanf(“%d”,&x[i]);}
printf(“Начальный массив :\n”);
for(i=0;i<n;i++){
printf(“%d”,x[i]);}
if(fmod(n,2)= =0) k=floor(n/2);else k=floor(n/2)+1;
for(i=0;i<n;i++){
if(i<k) s+=x[i]; else {s1+=x[i]; r++;} }
m=s1/r;
printf(“\nСумма первой половины %d\n”,s);
printf(“Среднее арифметическое второй половины %f”,m);
getch( );}

```

Пример 2: Обработка двумерного массива:

Дан двумерный массив  $Y[n,n]$ . Вычислить произведение положительных парных элементов заштрихованной площади:

```

#include<stdio.h>
#include<conio.h> n/2
#include<math.h>
#include<stdlib.h>
void main (void){
int x[20][20], n,i,j,p,k;
printf(“\Изберите величину массива n n<=20\n”);
scanf(“%d”,&n);
printf(“\nИзберите элемент массива \n”);
for(i=0;i<n;i++){
for(j=0;j<n;j++){

```

```

printf(“\nИзберите элемент x[%d][%d]\n”,i,j);
scanf(“%d”,&x[i][j]);} }
printf(“\nНачальный массив :\n”);
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf(“%d”,x[i][j]);}
printf(“\n”);}
p=1;k=floor((float)n/float(2));
for(i=0;i<n;i++){
for(j=k;j<n;j++){
if ((x[i][j]>0)&&(fmod(x[i][j],2) != 0)) p=p*x[i][j];
}} printf(“\nПроизведение элементов заштрихованной площади=%d
k+%d\n”,p,k);
getch( );}

```

## 8. Последовательность символов.

Называем рядом символьную последовательность алфавита, т.е. предложение. В отличие от других языков программирования язык С не содержит специальных типов данных, которые отличают последовательность символов. Язык С оперирует с последовательностями, как работал бы с рядом данных символьного типа, размещенных в массиве. Здесь каждый символ последовательности – отдельная составляющая массива. Так, для определения переменной, значением которой будет последовательность символов, в языке С необходимо объявить массив вида `char` с длиной равной возможному максимальному кол-ву символов в числовом ряду.

Следующий пример показывает, как может быть объявлен такой массив с целью избрания имен клавиатурой и затем их выявление на мониторе:

```
void main (void) {  
char a[20];int i;  
printf(“Выбрать имя “);  
for(i=0;i<20;i++){  
scanf(“%c”,a[i]);  
printf(“Ваше имя :”);  
for(i=0;i<20;i++){  
printf(“%c”,a[i]); }  
}
```

В этой программе замечено много недостатков и неудобств. Для избрания имени здесь использован цикл с 20 повторениями, который читает по одному символу с клавиатуры и вписывает в соответствующую ячейку массива `a[20]`. Следует заметить, что с помощью этой программы можно выбрать только имена, состоящие из 20 символов, или имя из `k` символов с `20-k` площадями после них. А ввод имени происходит с помощью функции `printf( )`, включенной в цикл, которая выводит на монитор по одному символу из избранного имени.

Заметка: Во время объявления массива типа `char` для описания числового ряда определяется значение массива на одна ячейку большее максимальной длины предвиденной последовательности, т.к. последняя ячейка массива зарезервирована для символа «`/0`».

Очевидно невозможно программировать в языке высокого уровня, используя такой механизм. По причине того, что программы по обработке текстовой информации очень распространены, язык С

использует механизм, облегчающий работу с символьной последовательностью. Здесь последовательности представлены специальным типом массива, что позволяет ввод и вывод последовательностей, как единого целого. Для ввода числового ряда в память компьютера используется функция `gets( )`. Эта функция обладает следующим синтаксисом:

`gets(имя)` ; где имя – параметр функции и представляет имя переменной числового ряда, т.е. массива вида `char`. Прим.:

```
void main (void){
int i; char name[15];
printf(“Выбрать имя:”);
gets(name);
printf(“Ваше имя:”);
for(i=0;i<15;i++)
printf(“%C”,name[i]);}
```

Здесь функция `gets( )` предусматривает первые 14 символов, выбранных на клавиатуре, как значение последовательности с именем имя, а последняя ячейка массива будет содержать символ `“\0”`. Во время работы функции `gets()` выполнение программы прекращается. Функция `gets()` ждет, пока пользователь не изберет числовой ряд на клавиатуре. Для того, чтобы выбранный текст был присвоен, как значение переменной последовательности, пользователь должен нажать клавишу `ENTER`. После этого выбранное выражение станет значением переменной последовательного вида, а каретка перейдет в следующую строку на мониторе. Именно во время нажатия клавиши `Enter`, компилятор `C` добавляет в конце последовательности `0`. В верхнем примере избрание последовательности предусматривает

избрание переменной последовательного вида, но не избрание многих переменных символьного вида. Хотя ввод имени остается неудобным. Здесь если будет избрано менее 15 символов клавиатуры, элементы массива имя [15], что следуют после символа 0 будут содержать приближенные значения.

Кроме функции `gets()` язык C содержит множество способов занесения предложения в память ЭВМ, как значение переменной последовательного вида. Используя функцию введения формата `scanf()`, можно применить следующий синтаксис:

`scanf("%S", имя);` которая ожидает избрания на клавиатуре последовательности символов, к которой затем (после нажатия клавиши ENTER) присваивается значение переменной имя. Здесь %S – формат последовательности символов.

Для объявления на мониторе последовательности символов используется функция `puts()`. Функция `puts()` может иметь в качестве параметров только символьную последовательность. Синтаксис функции `puts(параметр);` где в качестве значения используется последовательность символов, либо имя переменной символьного ряда.

Прим.:

`puts("Елена"); puts(имя);`

Большинство компиляторов C переводят каретку в другую строку после осуществления функции `puts()`. Хотя существуют и такие версии компиляторов, которые не выполняют такого перехода в новую строку. В этом случае используется символ по переходу в новую строку `"/n"`.

Прим.: `puts("Елена /n");`

Используя функции `gets()` и `puts()` вышеизложенный пример можно представить следующим образом:

```
void main (void){  
char name[15];  
puts(“Выбрать имя“);  
gets(name);  
puts(“Ваше имя:”);  
puts(name);}
```

### 8.1. Последовательные массивы.

Объявляя массив `char S[20]`, можно сохранить в нем значение последовательности символов. В случае необходимости обработки многого из нескольких последовательностей, удобно использовать последовательные массивы.

Последовательный массив – двумерный массив, составленный из строк и колонок. В каждую строку такого массива будет вписано по одной последовательности символов. Максимальное число последовательностей, вписанных таким образом в массив. Будет равен кол-ву строк в массиве. Например.: `char предложение [10][35]` - массив, в который может быть вписано 10 переменных последовательного вида, каждый из которых будет обладать максимальным значением 34 символа. Для допуска последовательности такого вида используется вид: `предложение[i]`, где *i* - номер ряда массива, где будет находиться последовательность. Для допуска символа последовательности *i* массива используется синтаксис: `предложение[i][j]`, где *j* - положение символа в последовательности *i*.

## 9. Структуры в C/C++.

До этого были изучены сложные типы данных, элементы которых принадлежат одним и тем же типам данных (простым или сложным). В случае определенного массива его элементы были одного типа: целочисленного, действительного, символьного и т.д.; без возможности присвоения различным элементам массива значения различных типов, хотя часто возникают ситуации, когда необходима обработка и сохранение определенной, более сложной информации, такой как расписание уроков, успеваемость студентов и т.д. Если разбирать случай с успеваемостью студента, то легко предположить, что возникает необходимость в следующей информации: имя студента, группа, оценка за экзамен, средний балл. Эти данные связаны между собой тем, что они принадлежат одному человеку. В последствии будет подтверждена их обработка, как единое сложное значение. Хотя типы данных отличаются друг от друга: имя и группа будут последовательного вида, а оценки за экзамен – целочисленного, а средний балл – действительного вида (float). Группировка этих составляющих в единую сложную переменную возможна при использовании нового типа данных, называемого в языке C – структура.

Первым шагом при группировке компонентов различных типов в единую сложную переменную, является объявление и описание структуры. Объявляя структуру, создается новый тип данных пользователя, который до еще не был известен компилятором. Объявление структур относится к объявлению типов до начала главной функции `main()` .

Объявление структуры начинается с ключевого слова “struct” , после чего следует имя структуры, которая еще называется тип «регистрация». Элементы переменной типа регистрации введены после имени переменной в фигурных скобках. Синтаксис описания элементов структуры аналогичен синтаксису объявления переменных: определяется имя и тип структурных элементов, разделенных символом «;». Синтаксис описания структуры в общем виде таков:

```
struct имя {тип_1имя_1;тип_2имя_2;...,тип_n имя_n ; } ;
```

Список элементов структуры носит название шаблона. Структура , сама по себе, не объявляет ни одной переменной. Элементы одной структуры не являются отдельными переменными, они – составляющие одной или более переменных. Такие переменные называются структурными и должны быть объявлены соответствующим структурным типом . Соответствующий шаблон опишет эти компоненты, и так будет определен объем резервной памяти для каждой структурной переменной регистрационного типа.

Если разобрать пример с успеваемостью студента, объявление структуры будет таким:

```
Struct stud{ char имя [20]; int вып1,вып2 ;float medie ; char група [10];};
```

### 9.1.Объявление переменных структурного типа.

Объявление структуры не резервирует пространство памяти для нее. До использования любого типа данных необходимо объявить соответствующую переменную. Невозможно использовать в программе структуру без объявления переменной регистрационного типа, равно,

как и невозможно использование определенного значения типа float до объявления переменной того же типа.

Синтаксис описания переменной-структуры:

```
struct имя _ структура имя _ переменная;
```

Здесь ключевое слово struct оповещает компилятор, что речь идет о структуре, а регистрационный тип stud определяет шаблон, по которому будет составлена переменная.

После регистрационного типа следует имя переменной, которое будет использовано в программе. Например, для получения доступа к данным об успеваемости студента необходимо объявление переменной:

```
struct stud a;
```

Теперь у нас есть переменная a, состоящая из 5 полей, для которых была зарезервирована память.

Если в программе необходимо использование нескольких переменных одного и того же регистрационного типа, возможно использование следующего синтаксиса:

Struct stud a,b,c; Здесь объявлены 3 переменные регистрационного типа stud a,b,c. В этом случае необходимо использование различного регистрационного типа, они будут объявлены отдельно.

Существует возможность объявления переменной регистрационного типа вместе с вводом структуры. Для этого имя переменной будет перемещено в фигурные скобки открытия и символа “;” в финале объявления структуры. Прим.:

```
struct stud {char имя [20] ;  
char группа [10] ;  
int вып1, вып2;  
float сред ;} a;
```

Здесь stud – регистрационный вид и имя нового типа данных, называемого структура. Элементы, из которой составлена структура еще называются полями, а имя переменной, которое будет использоваться в программе, и составленная ,следуя шаблону, из 5 составляющих.

## 9.2.Введение переменных регистрационного типа.

В случае, когда начальные значения компонентов переменной–структуры известны, возможно их присвоение во время объявления переменной. В случае объявления простой переменной регистрационного типа , введение будет частью объявления структуры:

```
struct stud {char имя[20]; char grupa[10];  
int вып1, вып2; float medie;}  
a={" Иванов", " SOE- 991", 8,7,7.5};
```

Здесь была описана структура stud и одновременно объявление переменной a с вводом переменных для ее компонентов.

Другой способ введения компонентов структуры – их ввод в тип объявления переменной регистрационного типа. Прим.:

```
struct stud {char имя[20];char grup[10];  
int вып1, вып2;float med;}  
main(){struct stud=a {" Иванов", "SOE-991", 8,7,7.5};}
```

Структура является глобальной, если она объявлена до главной функции main() и локальной, если объявлена внутри функции main() , или внутри другой функции. Хотя если необходимо выделение структуры, содержащей последовательности, она должна быть

объявлена до функции `main()` или как статичная переменная: `static struct stud;`

### 9.3.Использование структур.

Структура может быть обработана в программе в том случае, если существует объявленная переменная регистрационного типа. Эта переменная состоит из нескольких элементов, каждый из которых обладает значением различного типа. Хотя обращения к значениям, которые содержатся в структурных полях невозможно, используя непосредственно имя соответственного поля. Так невозможен доступ к значениям переменных регистрационного типа, используя только его имя. Следуя правилам синтаксиса языка C++, для доступа к элементам структуры необходимо введение имени переменной регистрационного типа и имя соответствующего поля, используя следующий синтаксис: `имя _ пер. имя _ поле`, где `имя _ пер` – имя переменной регистрационного типа, а `имя _ поле` – имя соответственного поля переменной.

Заметка: При обработке значений структурных полей будут использованы все функции, инструкции и применимые операции типа, к которому относятся структурные поля.

Прим.: Составить структуру, которая содержала бы информацию о студенте (имя, группа) и вычисляла бы среднее значение по результатам 2-х экзаменов.

#### 9.4. Составляющие структуры.

Структурный тип – сложный тип, а это значит, что переменная данного типа может быть составлена из нескольких простых или сложных элементов, которые, в свою очередь, так же могут содержать другие элементы. Это дает нам возможность использовать некоторые структуры в качестве полей для других структур. Такие структуры называются составляющими. Кол-во уровней составляющих структур теоретически можно определить, хотя не рекомендуется использовать множественные уровни составления по причине неудобства синтаксиса. В случае, когда структура А содержит в своем поле, которое, в свою очередь является структурой В, то структура А должна объявляться только после объявления структуры В. Следующий пример использует переменную а структурного типа stud составленную, которая содержит информацию о студенте и результатах сессии. Информация о результатах сессии сгруппирована в отдельной структуре под названием sesia и используется в качестве поля в структуре stud:

```
struct sesia {int
float med;};
struct stud {char name [20], group [10];
struct sesia nota;};
void main (void) {struct stud a;
puts(“Выбрать имя и группу“);
gets(a.name); gets(a.group);
puts(“Выбрать оценку по 3 экзаменам “);
scanf(“%d%d%d”,&a.nota.выр1 ,&a.nota.выр2,&a.nota.выр3);
```

```
a.nota.med=(a.nota.выр1 +a.nota.выр2 +a.nota.выр3 )/3;
```

```
printf(“\nmedia=%f”,a.nota.med);} .
```

Здесь был вычислен средний балл, исходя из результатов 3-х экзаменов. В этом примере легко заметить синтаксис обращения к значению составляющего поля структуры, используется название каждого составляющего поля, разделенных точкой, и записывается в убывающем порядке до назначенного поля. Синтаксис `a.nota.med` означает, что происходит обращение к полю `med`, которое относится к определенной структуре (`sesia`). Им, в свою очередь, является поле (`nota`) в составе другой структуры (`stud`) обращаемое непосредственно к переменной `a`.

### 9.5. Структурные массивы.

Во время объявления переменной регистрационного типа, в памяти регистрируется пространство для сохранения и обработки данных, которые будут содержаться в структурных полях, следуя шаблону только для одного ввода: один студент, персона ит.д. Наиболее часто необходима информация о группе людей, в нашем случае, о группе студентов. В этом случае необходимо объявление большего числа переменных регистрационного типа, каждый из которых представляет определенный ввод для каждого студента в отдельности.

Для систематически сохраненной информации среди множества вводов используется объявление структурного массива. В этом случае, каждый элемент одноименного массива структурного типа сохранит информацию о человеке, и максимальное кол-во людей, зарегистрированных таким образом, будет равно кол-ву элементов, содержащихся в массиве.

Одноименный структурный массив можно задать следующим образом:

`struct nume_structura имя_массив[N];` где `N` - кол-во элементов массива. Прим.: `struct stud x[10];`

Так были зарегистрированы 10 отделов памяти, которая, обладая необходимым объемом для сохранения целочисленной структуры для обработки информации об успеваемости студента.

Обращение в программе к элементу структурного массива будет осуществляться, как и в случае с простым массивом: будет указано имя массива с порядковым номером элемента из сохраненных скобок. Прим.: `x[3]` . Хотя такое обращение к структуре будет неправильным. Обработка структуры имеет место посредством обработки полей отдельных от нее. Доступ поля из переменной регистрационного типа , которая в это же время является полем массива, будет возможен, используя синтаксис: `имя _ массив`

`[k]. имя– поле,` где `k` - порядковый номер необходимого регистрирования.

Прим.: Для ввода в поле “name” из структуры “stud” имени студента с порядковым номером 2, используем следующий синтаксис: `struct stud x[10];dets(x[2].name).`

Замечание: Не стоит забывать, что исчисление элементов массива осуществляется с индекса 0. В нашем случае, составление массива из 10 структур, порядковый номер будет варьировать от 0 до 9.

Прим.: Дана база данных с `m` кол-вом вводов, содержащая информацию об успеваемости группы студентов. По оценкам за экзамен одной сессии вычислить средний балл для каждого студента. Необходимые поля: имя, группа, оценки за экзамен, средний балл:

```

#include<conio.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
struct stud{ char name[20], grupa[10];
int вып1,вып2;float med;};
void main (void) { clrscr();
struct stud x[50]; int i, N;
printf(“ Выбрать кол-во студентов\n”);
scanf(“%d”,&N);
printf(“Выбрать информацию про %d студентов:\n”,N);
for(i=0;i<N;i++){
printf(“ Выбрать имя студента %d\n”,i); scanf (“%s”,x[i].name);
printf(“Выбрать группу студента %d\n”, i); scanf(“%s”,x[i].grupa);
printf(“Выбрать 2 оценки для студента %d\n”,i);
scanf(“%d%d”,&x[i].вып1,&x[i].вып2 );
x[i].med=(x[i].вып1 +x[i].вып2 )/2;}
printf(“\nФинальный массив:\n”);
printf(“*****\n”);
printf(“** имя ** Группа **оценка1 **оценка2**среднее **\n”);
printf(“*****\n”);
for(i=0;i<N;i++){
printf(“** %10s**%7d**%7d**9.2f**\n”,
x[i].name,x[i].grupa,x[i].вып1 ,x[i].вып2 ,x[i].med);
printf(“*****\n”);
}
getch();}

```

## 10. Функции в C/C++.

Действующее решение проблем с помощью вычислительной техники предполагает использование всех возможностей и инструментов языка программирования. Во время изучения инструкций и типов в C++ возникают возможности решения некоторых сложных проблем, но, одновременно с возрастанием сложностей решаемых проблем, определяется объем программы и его сложность. В таком случае существует необходимость выделения конкретных задач программы, разделяя их в отдельные модули, называемые функциями.

Каждая функция программы должна выполнять одну задачу. Например, если находится средний балл группы студентов, тогда можно создать 3 функции: первая – для получения начальных значений о студентах, вторая – для вычисления среднего балла и третья – для вывода результатов. Используя эти функции, если в программе необходимо выполнение какой-либо задачи, обращаешься к соответствующей функции, обеспечивающей информацию, необходимую для обработки. Использование функций в C++ предполагает выполнение следующих базовых концепций:

А) Функции образуют последовательность операторов для выполнения определенной задачи;

Б) Главная программа, обращаясь к функции, обращается к ее имени, после которого следуют круглые скобки. Прим.: афиширование();

В) После окончания обработки информации большинство функций возвращают главной программе значения конкретного типа. Например.: `int` или `float`, которые могут употребляться при вычислениях;

Г) Главная программа передает функциям параметры (начальную информацию), включая в круглые скобки, которые следуют после имени функции;

Д) Язык C++ использует прототипы функции для определения типа значений, переброшенных функцией, а так же кол-ва и типов параметров, переданных функцией.

Вместе с измерением объема и усложнением программы использование функции является главным условием для правильного и действующего решения. В то же время, образование и использование функций является простыми процессами.

Во время образования программы необходимо зарезервировать каждую функцию для решения задачи. Если возникают случаи, когда функция выполняет несколько задач, она должна быть разделена на несколько более малых функций. Каждая созданная функция должна принимать единственное значение. Как и в случае с переменными, имя функции – идентификатор, который желательно должен соответствовать логическому знаку выполняемой задачи.

Функции в C++ походят на структуры с главной функцией `main()` . До имени функции указывается ее тип, а после имени - следует перечень параметров, описанных внутри круглых скобок. Тело функции, состоящее из операторов, располагается после описания параметров и окружено открытой и закрытой фигурными скобками. Синтаксис описания функции следующий:

```
тип_f имя_f (перечень параметров) {объявление переменной;  
операторы;}
```

где `тип_f` - тип функции или тип значений, переданных функцией, `имя_f` - имя функции. Аналогично, между функцией и главной

программой `main()` можно записать: `void main (void) {тело программы}` , где функция не возвращает результатов (слово `void` до функции `main()`) и не получает параметров из вне (слово `void` между круглыми скобками после функции `main()` ).

Следующие операторы определяют функцию с именем афиширование (), которая выводит на монитор сообщения: `void афиширование() (void){printf(“Hello World\n”);}`

Как было сказано, слово `void` , что предшествует афишированному имени, сообщает функции, что не надо возвращать хотя бы одно значение программе, а слово `void` после афишируемого сообщает имени (компилятору C++ и программисту, который считывает данный код), что функция не использует параметров (начальная информация для выполнения задачи). Следующая программа использует функцию `афиширование()` для афиширования сообщений на экране:

```
#include<stdio.h>
#include<conio.h>
void афиширование (void) {
printf(“\”Hello World“\n);
void main (void) {
puts(“до использования функции “);
афиширование ();
puts (“после использования функции “);
getch ();}
```

Как и любая программа в C++ этот пример будет выполняться , начиная с главной функции `main()` . Внутри программы оператор обращается к функции афиширования так: `афиширование()`; где круглые скобки после идентификатора сообщают компилятору, что в

программе используется функция `афиширование()`. Когда программа встретит обращение к функции, начнется выполнение всех операторов тела функции, и после этого выполнение главной программы будет продолжено оператором, размещенным обязательно после оператора обращения к функции. Следующая программа-пример содержит 2 функции, первая выводит на экран приветствие, а вторая – тему урока:

```
#include<stdio.h>
#include<conio.h>
void hello (void) {
printf(“\nHello\n”);}
void тема (void) {
printf(“Функции в C++\n”);}
void main (void) {
hello();
тема (); }
```

В результате выполнения этого примера будут выведены на экран 2 сообщения: ”Hello” и ”Функции в C++”, в том порядке, в котором появляются в программе.

Вышеизложенные функции выполняют очень простые задачи. В этих случаях программа могла быть составлена без использования функций, с включением этих же операторов в тело функции `main()`. Но функции были использованы для анализа описания и дальнейшего использования функций. Во время решения сложных проблем будет возможно упростить выполнение, разделив задачу на отдельные модули, выполняемые функциями. В этом случае легко заметить, что анализ и изменение функций намного проще, чем обработка объемной и сложной программы. Кроме того, функция, созданная для программы,

может быть использована без изменений и в другой программе. В этом случае могут быть составлены библиотеки функций, использование которых намного сократит время, используемое для составления программы.

### 10.1. Передача параметров функции.

Для изменения возможностей функции программы язык C++ допускает в них передачу информации. Начальная информация, переданная из программы функции в момент ее обращения, называется параметром. Если функция использует параметры. Они должны быть описаны во время описания функции. Во время описания параметров в функции определяется имя и тип каждого параметра следующим образом: тип \_ параметр имя \_ параметр;

Если функция содержит несколько параметров, они будут описаны вместе в круглых скобках после имени функции, разделенные запятой:

```
тип_имя функции_функция( тип_параметр1 имя_параметр1,  
тип_параметр2 имя_параметр2  
.....  
тип_параметрN имя_параметрN);
```

Функция следующего примера использует параметр целочисленного вида, который в результате своего выполнения будет выведен на экран:

```
#include<stdio.h>  
void число (int a) {  
printf(“Параметр= %d\n”,a);}  
void main (void) {  
число(1);
```

```
число(17);  
число(-145);} 
```

В результате выполнения этого примера будут приняты 3 фразы в качестве ответа:

```
Параметр=1  
Параметр=17  
Параметр=-145
```

Во время осуществления этой программы функция `число()` используется одинажды с различными значениями параметра. Каждый раз, когда в программе встречается функция `число()`. Значение параметра заменяется функцией и результат получается различный.

Параметры функции могут быть различных типов. В случае предыдущего примера параметр `a` – целочисленный. Если в программе будет использована попытка передать функции параметр другого типа, к примеру, `float` компилятор выдаст ошибку. Множество функций используют больше параметров, в этом случае надо указать тип каждого параметра. Следующий пример использует функция `рабочий()`, которая выводит на экран имя и сумму зарплаты одного рабочего:

```
# include<stdio.h>  
  
void рабочий (char имя[15], float зарплата)  
{printf(“Рабочий %s зарплата=%f лей\n”, имя, зарплата)  
void main(void) {  
рабочий (“Иванов”, 355.35);  
рабочий (“Петров”, 560.00);} 
```

в завершении этой программы функция `рабочий` используется 2 раза и выводится на экран:

Рабочий Иванов зарплата=355.35

Рабочий Петров зарплата=560.00

Замечание:

В некоторых книгах программирования языка C++, параметры которых передаются из программы в функцию называются действительными, а параметры, которые представлены в заголовке функции, к которым присваивают значения актуальных параметров, называются формальными параметрами. Итак, в предыдущем примере значения Иванов и 355.55 называются действительными параметрами, а char имя[15] и float зарплата называют формальными. Следующий пример использует функцию max(), параметры которой сравнивают 2 целых числа:

```
# include <stdio.h>

void max(int a, int b){
if(a<b) printf(“%d больше чем %d\n”,a,b);
else if(b<a) printf(“%d больше чем %d\n”,b,a);}
else printf(“%d равно %d\n,a,b”);}

void main(void){
max(17,21);
max(5,3);
max(10,10);}
```

Итак, во время использования параметров в функции необходимо соблюдать следующие правила:

- если функция использует параметры, она должна указать имя и тип каждого параметра;
- когда программа вызывает функцию, компилятор присваивает значение параметрам слева направо;

- значения, переданные из программы в функцию должны совпадать по месту и типу с параметрами из функции.

## 10.2 Возврат значения функции.

Задача каждой функции - это выполнение какого-то задания. В большинстве случаев, функции будут выполнять некоторые вычисления. После этого функция возвратит результат функции, из которой была вызвана, будучи главной функцией `main()` или любой другой. В то время, когда функция возвращает значение, обязательно надо знать её тип. Тип значения, возвращаемого функцией, указывается во время представления функции до её имени. Тип возвращаемого значения называется тип функции.

Функция из следующего примера суммирует 2 целых числа и возвращает ответ главной программе:

```
# include<stdio.h>
# include<conio.h>
int suma(int a, int b){
int R;
R=a+b;
Return (R); }
Void main(void){int k;
K=suma(15,8);
Printf(“suma=”,k);}
```

В этом случае, слово `int`, которое находится до имени функции `suma()`, во время своего описания, является типом возвращенного значения функции главной программы.

Функции используют оператор “return” для того, чтобы вернуть значение функций, из которых были вызваны. Когда компилятор встречает оператор “return”, он возвращает присвоенное значение и завершает выполнение предыдущей функции, контроль выполнения программы, будучи переданным функции, из которой была вызвана раньше функция. Если после оператора “return”, в функции существуют еще операторы, они будут игнорированы. Функция оканчивается вместе с выполнением оператора “return”.

Функция `suma()` из предыдущего примера составлена из 3 инструкций. Существует возможность уменьшить тело этой функции. Смотри следующие примеры:

```
int suma(int a, int b) {return (a+b);}
```

В этом случае, параметром для оператора “return” было использовано выражение, которое вычисляет сумму 2 чисел. В общем, параметр “return” использует в качестве параметра выражение, имеющее результате тип ,идентичный с типом функции, именно возвращенного функцией значения.

Не все функции возвращают значения целого типа. Следующий пример использует функцию `media()`, которая возвращает среднее арифметическое 3 целых чисел. Ответ функции может быть и реальным числом.

```
# include<stdio.h>
# include<conio.h>
float media (int a, int b, int c){
return(float(a+b+c)/3.0);}
float R;
R=media(5,7,10);
```

```
printf(“media=%f”,R);} 
```

В завершении выполнения этого примера, будет ответ  $R=7.333\dots$ . В функции была использована вариация (`float(a+b+c)`) целого числа в своем действительном эквиваленте для получения в ответе среднее арифметическое 3 чисел действительного типа. В этом примере слово `float`, которое находится до имени функции `media()` во время её описания, указывает тип возвращаемого значения.

Существуют ситуации, когда оператор `return` используется в функциях, которые не возвращают значений.

Следующий пример доказывает это:

```
# include<stdio.h>
# include<conio.h>
max(int a, int b){
if(a>b) printf(“%d>%d\n”,a,b);
else if(a<b) printf(“%d<%d\n”,a,b);
else printf(“%d=%d\n”,a,b);
void main(void){
max(17,21);
max(3,-7);
max(5,5);} 
```

Значение, возвращенное функцией, может использоваться в любом месте программы, где есть возможность использования одного значения одинакового типа с возвращенным значением, когда функция возвращает одно значение, это значение может быть присвоено одной переменной того же типа, используя оператор присвоения. Пример:  
`R=media(5,7,10);`

Имя функции может использоваться в случае, когда существует необходимость использования возвращенного значения функцией, например, возвращенное значение может быть использовано внутри афишируемой функции.

```
printf("media=%f",media(5,7,10))
printf("media=%f\n",r);
getch();}
float media(int a, int b, int c){
return (float(a+b+c)/3.0);}
```

### 10.3. Прототип функции.

Перед тем как вызвать функцию компилятор C++ должен распознать тип обратного значения, количество и тип параметра используемого функцией. В каждом примере искомой функции до описания функции должна быть сделана перед ее вызовом из главной программы. Еще бывают ситуации, когда некоторые функции в программе вызваны одна другой. В таких случаях возможна ситуация когда одна функция вызвана до своего описания.

Чтобы гарантировать случай, что компилятор C++ распознает значение каждой функции используемой в программе, используются прототипы функций. Прототип одной функции расположен в начале программы и содержит информацию об типе обратного значения, значение и тип параметра используемого функцией.

Следующий пример показывает как можно создать прототип для функции ранее используемой.

```
void афишаре (void);
```

```
void привет (void);
void тема (void);
void номер (void);
void работчий (void);
void max (int,int);
int suma (int,int);
float среднее (int,int,int);
```

Как видно из примера у каждой функции есть прототип который описывает тип обратного значения, количество и тип параметра. Важно не забывать о символах “;” в конце каждого прототипа. Их отсутствие идет в ошибки в компилятор, так же как идет в ошибки отсутствие прототипа функции если та появляется в программе раньше своего описания. Пример:

```
# include<stdio.h>
# include<conio.h>
float media (int,int,int);
void main (void){ float r;
clrscr( ); r=средняя (5,17,10);
printf (“средняя=%f/n”,r);
getch ( );}
float среднюю (int a, intb, intc){
return (float(a+b+c)/3.0);}
```

#### 10.4. Локальные переменные и область их видения.

Функции, использованные в предыдущем примере, выполняют лёгкие задачи. В момент, когда функции будут выполнять более

сложные задачи, возникнет потребность использования собственных переменных в функциях.

Переменные, представленные в функции, называются местными. Имя и значение одной местной переменной известны только функции, в которой она была объявлена. Даже факт, что местная переменная существует, известен только функции, в которой она была объявлена. Объявление переменных имеет место в начале функции, сразу после фигурной скобки, которая открывает ее тело. Имя местной переменной должно быть единственным только в функции, в которой она была объявлена. Переменная называется локальной из-за того, что она видна только в функции, в которой была описана.

Синтаксис объявления местной переменной следующий:

```
тип_f имя(список параметров) {тип_vl имя_vl}
```

где тип\_f – тип функции; имя функции;

тип\_vl – тип переменной; имя\_vl – имя переменной;

Правила представления и использования местной переменной любой функции схожи с правилами представления и использования местной переменной, представленной в теле главной функции main(). Переменная, представленная в теле функции main() является местной.

В общем, все, что было сказано про представленную переменную в функции main(): тип, имя и методы использования и т.д. можно использовать для местной переменной любой другой функции.

Следующий пример использует функцию fact(); для вычисления факториала числа.

```
# include<stdio.h>
# include<conio.h>
int fact(int R){
```

```

int i, k=1;
for(i=1;i<=R;i++){k=i;}
return(k);}

void main(void){
int n; clrscr();
printf(“Ввести число\n”);
scanf(“%d”,&n);
printf(“%d!=%d\n”,n,fact(n));
getch();}

```

В предыдущем примере функция fact() использует 2 местные переменные i и k целочисленного типа. Переменная k вводится со значение 1, даже в момент своего объявления. Именно в этой переменной будет сохранено значение факториала во время выполнения функции fact().

Проанализируем этот пример: В начале выполнения программы пользователь вводит цифру целочисленного типа, из которой будет вычислен факториал. Число, введенное пользователем, сохраняется сначала в переменной n, потом, одновременно с вызовом функции fact(), передаётся как параметр, будучи представленным условному параметру R из функции. В начале выполнения функции местной переменной k, в которой будет храниться значение факториала, присваивается начальное значение 1, для избежания случайных значений. Именно через местную переменную k функция возвращает финальное значение факториала в главную программу. Дальше цикл for будет повторён n раз, для того чтобы вычислить значение факториала n!, умножая каждый раз новое значение параметра цикла i на предыдущее значение k. В конце финальное

значение k будет возвращено в программу с помощью оператора return();

Во время представления местной переменной для функции существует вероятность, что имя местной переменной, объявленной в функции, может быть схожим с именем местной переменной из другой функции. Как было сказано, местная переменная известна только функции, в которой была объявлена. Итак, если 2 функции используют одно и то же имя для местных переменных, это не приводит к разногласию. Компилятор C++ рассматривает имя каждой переменной, как местное для соответствующей функции.

Следующий пример использует функцию suma() для складывания 2 целых чисел. Эта функция присваивает результат своего выполнения местной переменной x. Но функция main() использует в качестве параметра, передаваемого функцией suma(), также и местную переменную x. Из вышесказанного следует, что обе переменные будут рассмотрены, как местные, и не будут возникать разногласия.

```
# include<stdio.h>
# include<conio.h>
int сумма(int a, int b){
int x;
x=a+b;
return(x);}
void main(void){
int xy;
printf(“Выбери 2 цифры\n”);
scanf(“%d%d”,&x,&y);
printf(“%d+%d=%d”,x,y,suma(x,y));
```

```
getch();}
```

Следующий пример использует функцию `suma()` для вычисления суммы элементов целочисленного типа. В качестве параметров используется массив и его величина.

```
#include <stdio.h>
#include<conio.h>
int suma(int y[10], int m){
int i, suma=0;
for(i=0;i<m;i++){
suma+=y[i];}
return(сумма);}
void main(void){
int w, n, i, x[10]; clrscr();
printf(“Ввести длину масива n<10\n”,i);
scanf(“%d”, &n);
for(i=0;i<n;i++);
printf(“Ввести элемент x[%d]\n”,i);
scanf(“%d”, &x[i]);
n=сумма(x,n);
printf(“Сумма масива=%d\n”,w);
getch();}
```

## 10.5. Глобальные переменные.

Глобальной переменной называется переменная, имя и значение которой известны в течение всей программы. О существовании глобальной переменной в программе C++ знает любая функция из этой программы. Для создания глобальной переменной используется её

объявление в начале программы вне любой функции. Любая функция, которая будет выполняться после такого представления, может использовать эту глобальную переменную.

Представление глобальной переменной имеет следующий синтаксис:

```
#include <stdio.h>
```

```
тип vg имя_vg;
```

```
void main(void)
```

```
{...}
```

где `тип_vg` – тип глобальной переменной, `имя_vg` – имя глобальной переменной.

Будучи представленной, значение глобальной переменной не только известно любой функции программы, но и может быть изменено любой из функций, представленной в программе.

Следующий пример использует переменную с именем `цифра`. Будучи доступной любой из 2 функций, представленных в программе, значение глобальной переменной `цифра` будет изменено по очереди в обеих функциях.

```
#include <stdio.h>
```

```
int цифра=100;
```

```
void f1(void){
```

```
printf(“цифра=%d\n”, цифра);
```

```
цифра*=2;}
```

```
void f2(void){
```

```
printf(“цифра=%d\n”, цифра);
```

```
цифра+=2;}
```

```
void main(void){
```

```
printf(“цифра=%d\n”, цифра); //100
```

```
цифра++;  
f1(); //101,102  
f2(); //202,204  
printf(“цифра=%d\n”, цифра); //204  
getch();}
```

Несмотря на то, что глобальные переменные в программе дают новые возможности, желательно избегать их использования из-за того, что любая функция программы может изменить это значение глобальной переменной. Очень трудно следить за всеми функциями, которые могли бы изменить это значение, и это ведёт к трудному контролю над выполнением программы. Для решения этой проблемы можно объявить переменную в теле функции `main()` и потом передать ее другим функциям в качестве параметра. В этом случае в штабель будет перемещена временная копия этой переменной, а начальное значение (оригинал) останется неизменным.

#### 10.6. Конфликт между местными и глобальными переменными.

В случае, когда программа должна использовать глобальную переменную, могут возникнуть конфликтные ситуации между именами глобальной и местной переменной. В этом случае язык C++ отдает предпочтение местной переменной. А именно, если существуют глобальные переменные с тем же именем, что и местная переменная, компилятор считает, что любой вызов переменной с таким именем является вызовом местной переменной. Но существуют ситуации, когда существует потребность обращения к глобальной переменной, которая

находится в конфликте с глобальной переменной. В этом случае можно использовать глобальный оператор (::).

Следующий пример использует глобальную переменную - имя. Кроме того, функция afisare(); использует местную переменную - имя. Используя глобальный оператор (::) доступа, функция вызывает глобальную переменную имя без конфликта с местной переменной имя:

```
#include <stdio.h>
int num=505;
void объявление(void){
int имя=37;
printf(“Местная переменная имя=%d\n”,имя);
printf(“=%d\n”,::имя)
void main(void){ объявление( );}
```

## 11. Указатели.

Программа в языке программирования C/C++ содержит в многих случаях множество переменных вместе с их описанием. Пусть переменная x описана в программе посредством представление формы int x;. В этом случае компилятор бронирует место в 2 байта в памяти компьютера. В свою очередь, ячейка памяти, куда будет вписано значение переменной x, имеет свой адрес. Пример: 21650. Когда переменная x принимает значение, это значение обязательно вписывается в ячейку памяти, бронированную для переменной x. Так как в нашем случае переменная x хранит свое значение по адресу

21650, ячейке с этим адресом будет содержать присвоенное значение переменной x.

В таком случае, к любой переменной из программы можно обратиться, используя 2 способа:

- Ø используя имя переменной. Используя имя переменной, мы обращаемся к значению переменной, которое хранится в памяти;

- Ø используя адрес памяти, где хранится значение переменной x (используя адрес, мы обращаемся к ячейке памяти, где хранится значение переменной).

Язык программирования C/C++ содержит новые технологии работы с значением переменной и адресом памяти, где хранится это значение. С этой целью используются указатели. Указатель представляет собой переменную, которая содержит адрес другой переменной. Синтаксис объявления указателя следующий:

`tip* pume;` где `tip` – тип данных переменной, к которой может относиться указатель (т.е тип данных, адрес памяти которой относится к указателю); звездочка „\*” указывает, что индикатор, который следует после него - указатель.

Имя – идентификатор, который означает имя переменной указателя.

Примеры представления указателей:

```
int *x; //указатель целочисленных данных;
```

```
float *p; //указатель временных данных
```

```
char *z; //указатель символьных данных
```

```
int *y[3]; //массив указателей целочисленного типа данных
```

```
void *k; //указатель предмета типа данных, который  
необязательно определять
```

```
char *s[5]; //массив указателей данных символьного типа
```

`char(*s)[5];` //указатель данных символьного типа из 5 элементов. Одновременно с тем, что указатель был приставлен в качестве отзыва данных целочисленного типа, он не может обратиться к переменной типа `float`, а причина - это объем отдельно забронированной памяти для целочисленных и временных переменных. Существует указатели и для элементов без типа – `void`.

Можем присвоить одному `pointer-void`-у значение другого `pointer non void`, без необходимости операции определения типа переменной.

Пример:

```
int *a; void *b;
b=a; //верно
a=b; //неверно, отсутствует вариация типа.
```

Пусть будет `int x=5`; в этом случае для переменной `x` было забронирован объем памяти в 2 байта, который имеет свой адрес. Адрес памяти, где хранится значение переменной `x`, можно получить с помощью оператора получения адреса “&”. Результат операции получения адреса - это адрес нахождения памяти, что была выделена для соответствующей переменной.

Пример:

Представим, что `x` вписана в память по адресу 21650, тогда `&x` будет равно 21650. Важно, что `&x` - константа указательного типа и свое значение не меняет во время выполнения программы. Пример:

```
#include <stdio.h>
void main(void){
int x=5; float r=1.7;
int *q; float *w;
printf(“%f находится по адресу%d\n”,r,w);
```

```
printf(“%d находится по адресу%d\n”,x,q); }
```

Из примера легко заметить, что адрес ячейки памяти представлен значением целого типа. В то же время, это значение не может больше изменяться в программе, и выражение `&x=55;` будет неверным.

Анализируя все эти определения, возникает вопрос: « С какой целью используются указатели, если значение переменной и значение адреса можно хранить в простых переменных?».

Преимущество использования указателей состоит в том, что к нему можно обращаться 2 способами: `q` и `*q`. Звездочка “\*” в этом случае указывает, что вызывается содержимое в ячейке памяти, адрес которой и есть значение указателя. А именно, значение переменной `x` целого типа равно 5, значение указателя `q=21650`, а `*q` равно целочисленной величине 5, вписанной по адресу в памяти 21650.

В таком случае:

1. переменная `q` может получать значения только в форме адреса `q=&x`, и присвоение в виде `q=21650` неправильно, т.к. здесь происходит попытка присвоения определенного целочисленного значения указателю, а не адресу;

2. переменная `*q` может получать значения целочисленного типа. Пример: `*q=6;` это присвоение расшифровывается так: перенести значение 6 в ячейку памяти по адресу, указанному переменной `q`. Так как переменная `q` указывает клетку с адресом 21650, значение переменной, хранящей значение в ячейке памяти с этим адресом, будет равно 6. Пример:

```
#include<stdio.h>
```

```
main void(main){
```

```
int x=5;
```

```
int *q;  
q=&x;  
printf("x=%d\n,x"); //x=5;  
printf("x=%d\n,x"); //x=6;
```

В завершении этого примера на экран будут выведены 2 выражения:  $x=5$  и  $x=6$ . Первое значение, полученное переменной  $x$ , является значением 5, присвоенным в момент данного введения. Второе значение, полученное переменной  $x$ , будет 6, присвоенное ячейкой памяти, на которую указывает указатель  $q$ .

Конечно же, переменная  $x$  могла бы получить и новые значения с помощью простого присвоения,  $x=6$ ; но полноценный эффект от использования указателя может быть замечен в их передаче в качестве параметров функции, при создании файла и т.д.

Необходимо заметить, что в некоторых случаях невведенный указатель представляет опасность в программе. Пример: необходимо передать данные по адресу, который содержит указатель; в случае, если это неправильно введено, можно стереть главную информацию, необходимую для функционирования операционной системы или других программ, т.к. невведенный указатель может указать на любой другой адрес в памяти, включительно, и на адрес, где хранятся главные данные операционной системы. Пример ввода указателя:

```
int *q = &x.
```

С целью эффективного использования памяти язык C/C++ позволяет программистам освободить выделенную память, если это необходимо. Для этого используется `free()`, прототип которой находится в библиотеке `alloc.h`.

Пример:

```
#include <stdio.h>
#include <alloc.h>
main void(main){
int x=5, *q=&x;
*q=6;
printf("x=%d\n",x); //x=6;
free(q);
printf("x=%d\n",x); /x=-16;
printf("Адрес x=%d\n",&x); //-12.
```

В результате выполнения этой программы будут получены сообщения: x=6, x =-16, адрес x=-12. Первый ответ будет тот же, если програма будет выполнена на другом компьютере. Последние 2 ответа будут разными в зависимости от компьютера и от объема его памяти. После использования функции free(q), значение, которое хранилось в ячейке памяти, на которую указывает указатель q, будет потеряно, а именно значение x будет потеряно.

### 11.1. Указатели и функции.

Из предыдущих тем нам известно, что во время работы с любой функцией при её вызове можно передать столько параметров, сколько понадобится программе. В то же время с помощью оператора “return” можно вернуть только одно значение из функции. Ситуация, когда функция должна вычислить и вернуть несколько значений программе, это не всегда удачно. В качестве альтернативы можно использовать

глобальную переменную, но в этом случае часто теряется управление над выполнением программы, и усложняется поиск ошибок.

В случае, когда существуют потребность вернуть из функции больше значений, очень сложно использовать указатели. Без использования указателей мы должны передать параметры функции в соответствии со значениями. Это значит, что значение действительного параметра присваивается формальному параметру. Эти параметры занимают разные зоны памяти, и значит, изменение значения формального параметра функции не приводит к изменению значения действительного параметра.

В случае передачи параметра по адресу с помощью указателя, не создаются копии значений, которые дублируют параметр. В этом случае создается вторая переменная, которая указывает на ту же ячейку памяти. В таком случае, если значение формального параметра функции изменено указателем, значение действительного параметра тоже будет изменено.

Пример:

```
#include <stdio.h>

int schimb(int w, int *z)
(*z)*=2; w*=2;
return(w);}

void main(void){
int x=5, y=8,a;
printf("x=%d, y=%d",x,y);
a=schimb(x,&y);
printf("x=%d, y=%d\n",a,y);}
```

В этом примере даны 2 переменные  $x=5$ ,  $y=8$ . Функция `schimb()` меняет их значения, умножая на 2, но если значение  $x$  будет изменено в функции и возвращено программе с помощью оператора “return”, тогда значение  $y$  будет изменено с помощью указателя. Вызвав функцию `schimb()`, мы передаём в качестве параметра не значение переменной  $y$ , а адрес ячейки памяти, где она хранит свое значение. В таком случае любое изменение значения указателя `*z` функции ведет к изменению значения переменной  $y$  из программы.

В другом случае, в работе с указателями используются указатель для целого ряда символов. Последовательность символов представляет массив символьного типа. Когда программа передает массив функции, компилятор передает адрес первого элемента массива. В результате, допустимо, чтобы функция использовала указатель для последовательности символов.

Следующий пример высчитывает, сколько раз в данном выражении встречается символ «а».

```
#include <stdio.h>
int litera(char *s){
int k=0;
while(*3!='\0')
{ printf(“Символ %с по адресу %d”,*3,9);
if(*s=='a') k++;
s++;}
return(k);}
void main(void)
int p; char x[25];
puts(“Выбери строку”);
```

```

gets(x);
p=litera(x);
printf(“Строка содержит %d символы а ”, p);}

```

Здесь при вызове функции litera() в качестве параметра передается не адрес последовательности символов x, а именно ряд. Потому что в передачи массива в качестве действительного параметра, передается не весь массив, а только адрес первого элемента массива. Значит, указатель s содержит адреса элементов массива, а \*s – их значение. В таком случае, при первом выполнении цикла while переменная \*s будет иметь значение равное первому символу из последовательности x, переданной функции. После выполнения первой части функция увеличивает значение переменной s на 1. Это значит, что значение адреса первого символа из строки будет инкрементировано, получая в результате адрес второго символа из строки. Цикл окончится, когда будет найден нулевой символ из строки.

То же самое можно сделать и с массивом целого типа.

```

#include <stdio.h>
#include <conio.h>
int masiv(int *s,int *z)
{int k=0, i;
for(i=0;i<z;i++){
printf(“Элемент %d=%d находится по адресу %d”,i,*s,s);
if(*s==)k++;
s++;}
return(k);}
void main(void){
int p,n,i,x[25]; clrscr();

```

```

printf("Избери величину массива x\n");
for(i<0;i<n,i++)
scanf("%d",&x[i]);
p=masiv(x,n);
printf("Массив содержит %d ноль",p);
getch();}

```

В этом случае высчитывается количество нулей в массиве. Величина массива передаётся функции от действительного параметра *n* к формальному параметру *z*. А для передачи массива *x* в функции используется присвоение значения первого элемента *x[0]* переменной *\*s*.

Используя указатели, можно создать функции, задачей которых было бы избирание значения элементов одномерного массива. Эти функции стоит использовать в программах, где обрабатывается несколько массивов.

В следующем примере демонстрируется этот факт обработки 2 массивов: *x* и *y*.

```

#include <stdio.h>
#include <conio.h>
int masiv(int *k){
int i,z;
printf("Избери величину массива <50\n");
scanf("%d",&z);
for(i=0;i<n;i++) {printf("Избери элемент %d\n",i);
scanf("%d",k);k++;}
return(z);}
void main(void){

```

```

int n1,n2,i,x[50],y[50]; clrscr();
printf("Вводим массив x\n"); n1=masiv(x);
printf("Вводим массив y\n"); n2=masiv(y);
printf("Вывод массива x:\n");
for(i=0; i<n;i++) printf ("x[%d]=%d\n",i,x[i]);
printf("Вывод массива y:\n");
for(i=0; i<n;i++) printf ("y[%d]=%d\n",i,y[i]);
getch();}

```

В этом примере элементы массива и его величина введены в функцию, а главная программа выводит результаты выполнения функции. В качестве параметра функция `masiv()` использует указатель целого типа `*k`, которому передается из программы адрес первого элемента обрабатываемого массива. Внутри цикла значение адреса из указателя исключается, в этом случае, получая адрес следующего элемента обрабатываемого массива. Функция возвращает в программу значение величины текущего массива, которое присваивается переменным `n1` для массива `x` и `n2` для массива `y`.

Имея такой алгоритм, очень легко создать обрабатываемые программы, которые обрабатывают большее число массивов, и даже двумерные массивы. Но в случае двумерных массивов надо заметить, что значения элементов массива будут храниться в ячейках памяти, адреса которых линейно изменяются для элементов массива по строкам, а потом по столбцам, именно слева на право и сверху вниз.

Пример:

```

#include <stdio.h>
#include <conio.h>
int masiv(int(*k)[50]){

```

```

int i,j,z;
printf("Избери величину массива\n");
scanf("%d",&z);
for(i=0;i<z;i++){
for(j=0;j<z;j++){
printf("Избери элемент [%d][%d]\n",i,j);
scanf("%d",&(k[i]+j));}}
return(z);}

void main(void) { clrscr();
int n1,n2,i,j,x[50],y[50][50];
printf("ВВОДИМ МАССИВ X\n");
n1=masiv(x);
printf("ВВОДИМ МАССИВ Y\n");
n2=masiv(y);
printf("массив x: \n");
for(i=0;i<n1;i++){
for(j=0;j<n1;j++){
printf("%d",x[i][j]);}
printf("\n");}
printf("массив y: \n");
for(i=0;i<n2;i++){
for(j=0;j<n2;j++){
printf("%d",x[i][j]);}
printf("\n");}
getch();}

```

Здесь в качестве параметра функция используется указатель (\*k)[50] одномерного массива из 50 элементов. Значит, каждое изменение

адреса через `k++` ведет к положению на первый элемент следующей строки из двумерного массива. А каждый указатель `(k[i]+j)` в цикле будет указывать на элемент столбца `j` из строки `I` двумерного массива.

## 12. Файл в C/C++

Во время выполнения программ во многих случаях существует потребность вывода результатов. Но простые возможности вывода данных на экран очень малы. Даже и использование временной остановки выполнения программы с целью передачи пользователю возможности прочитать всю выведенную информацию – не решает до конца проблему. Если выведенный результат исчезает из пределов экрана, его повторное введение будет возможно после повторного выполнения всей программы. Более того, присвоенные значения переменных программы хранятся только в течение выполнения программы. Одновременно с завершением выполнения программы вся внесенная информация теряется.

Для хранения информации, однажды внесённой с целью её использования вновь, необходимо записать информации на диске в специальных структурных данных, которые называются файлами. Использование файлов позволяет хранить любую информацию любого типа на длительное время, перевод данных от одного информационного носителя к другому, ввод и вывод данных. Вся информация, в чем состоит работа с файлами в C/C++, хранится в `stdio.h`. В начале обработки файла в C/C++ необходимо вписание этой библиотеки в программу с помощью директивы `#include <stdio.h>`,

которая позволяет вписание информации в файл и ее чтение . Во время входа данных в программу из файла с диска сначала идёт их копирование в оперативной памяти, а информация из файла, которая находится на неподвижном диске остаётся неизменной в течение выполнения программы. При выходе данных из программы на диск в файл записываются данные, что сохранялись до того момента в оперативной памяти.

Запись/чтение символов из файла выполняется с помощью указателя файла. Во время записи или чтения информации из файла компилятор использует промежуточный уровень связи между программой и жестким диском, где хранится файл. Этот уровень представляет зону памяти, называемую зона тампон, которая обладает предназначением временного хранения информации с целью ее вписания или чтения затем из файла.

Для послания или чтения информации из зоны тампон компилятор использует специальную структуру, называемую структура-файл. В этой структуре сохранена информация, необходимая компьютеру для выполнения вписывания или чтения данных из файла, включая адрес памяти, куда перемещен файл. Синтаксис объявления указателя файла следующий:

`FILE *file_pointer;` где ключевое слово `FILE` указывает компилятору, что объявляемая переменная является указателем файла. а `file_pointer` – имя указателя. В случае, когда программа предполагает работу с несколькими файлами, необходимо объявление нескольких указателей файлов. Как например:

```
FILE *f1,*f2,*f3;
```

Где \*f1, \*f2, \*f3 имена указателей различных файлов. Обработка файлов в языке C\C++ предполагает выполнение следующих шагов.

1) Открытие файла. (Для того, чтобы было возможно чтение или вписание информации из файла, он должен быть открытым).

2) Обработка файла (операции чтения и ввода).

3) Закрытие файла. Для того, чтобы введенная информация была сохранена в файле, он должен быть закрыт.

### 12.1. Открытие файлов.

Открытие файла выполняется при помощи функции `fopen()`, которая обладает следующим синтаксисом:

```
Pointer=fopen(“имя_f”, “вид “);
```

, где `pointer` – имя указателя файла, `имя_f` - имя действительного файла неподвижного диска, `вид` – вид доступа к файлу.

Результат выполнения этой функции – присвоение адреса структурного файла , указывающего на файл. Первый параметр имени файла, как правило, обладает следующей структурой: `имя.ext`, где `ext` – растяжение файла, состоящего их 3-х символов. В качестве 2-го параметра функция принимает способ доступа к файлу, т.е. информация об операциях, которые могут быть выполнены с помощью файла.

Существуют 3 главных способа открытия файла:

1) Способ “w” допускает открытие файла с целью вписания в него информации или вывод информации на принтер. Если указанный файл не существует, он будет создан. Если файл уже

существует, вся информация, существующая на нем , будет уничтожена.

2) Способ “r” указывает компилятору, что файл будет открыт с целью чтения с него информации. Если файл не существует, в момент открытия будет выявлена ошибка в выполнении.

3) Способ “a” допускает открытие файла с целью заполнения, т.е. вписания информации в его финале. В случае, когда файл не существует, он будет создан заново. Если вводимый файл существует, вводимая информация будет перемещена в конец файла без уничтожения информации в файле.

Пример: для создание нового файла с именем info.txt будет использован следующий синтаксис:

```
FILE *файл;
```

```
файл=fopen(“info.txt”, “w”);
```

В случае, когда необходимо чтение нескольких данных из файла, будет использован способ доступа “r”:

```
FILE*файл;
```

```
файл=fopen(“info.txt”, “r”);
```

Для передачи информации с файла на бумагу с помощью принера, имя файла будет PRN, а способ доступа - “w”:

```
FILE *файл
```

```
файл=fopen(“PRN”, “w”);
```

Необходимо отметить, что имя файла и символ, который определяет способ доступа к файлу, определены двойными кавычками. Это обусловлено тем, что параметры функции fopen() передаются в качестве последовательности символов. Используя

эти возможности, можно набрать с клавиатуры имя желаемого пользователем файла:

```
char name [12];  
FILE *f;  
printf (“Изберите имя файла\n”);  
gets(name);  
f=fopen (name, “w”);
```

Во время работы с файлами в C/C++ используется специальный указатель, в котором сохраняется информация текущего положения чтения с файла. Во время чтения данных из файла указатель определяет следующую позицию данных, которая должна быть считана с диска. Если файл открыт впервые способом доступа “r”, указатель переходит на первый символ из файла. Во время выполнения следующей операции чтения, указатель переходит в начало следующей части данных. Величина шага чтения с файла зависит от кол-ва считываемой с файла информации. Если через шаг считывается только один символ, тогда указатель переходит к следующему символу, если считывается структура, указатель перейдет на следующую структуру. Во время того, когда вся информация была прочитана, указатель попадает на специальный код, называемый финалом файла (eof). Попытка чтения с файла после его финала приводит к ошибке. Если файл открыт способом доступа “w”, указатель также переходит в начало файла. В это случае первые вписанные в файл данные будут перемещены в его начало. Во время закрытия файла, в его финале будет вписан символ окончания файла(eof). Если во время открытия в режиме доступа “w” файл существует,

вся уже существующая в нем информация будет уничтожена, и на ней будет записана новая информация. Любые предшествующие данные, которые могут оставаться нестертыми, будут перемещены в файл посредством кода окончания файла (eof). Так доступ к ним будет закрыт. В этом случае вся информация с открытого файла режима доступа “w” будет уничтожена, так же, как и случае, когда файл будет открыт без вписания в него некоторых данных.

Если файл открывается режимом доступа “a”, указатель переходит в символ окончания файла (eof). Любая информация, записанная таким образом в файл, будет перемещена после уже существующих данных, а после ввода в финале файла добавляется код финала файла(eof). В некоторых случаях возникает ситуация, когда операционная система не может открыть указанный файл в функции fopen(). Это обусловлено полным отсутствием жесткого диска или тем, что указанный файл просто не существует. Возможна такая ситуация, когда необходим вывод данных на принтер, а он не включен, или отсутствует бумага. В случае попытки использования файла, который не может быть открыт, программа будет остановлена в результате допущенной ошибки. Для предотвращения аварийной остановки программы, может быть проверено состояние открытия файла с помощью условного оператора if, который останавливает программу в случае ошибки. Здесь будет использована особенность операционной системы, которая возвращает нулевое значение, в случае, когда возникает ошибка, и файл не может быть открыт. В этом случае нулевой код возвращен вместо адреса структуре файла, и программа

остановится. Условие для предотвращения аварийного выхода из программы, будет обладать следующим синтаксисом:

```
if ( (файл =fopen(“info.txt”,”w”))==НОЛЬ )  
puts (“Файл не может быть открытым “);  
exit();
```

После того, как вся информация вписана в файл или считана с него, необходимо закрыть файл, т.е. прервать связь между файлом и программой. Закрытие файла происходит с помощью функции `fclose()`, которая обладает следующим синтаксисом:

`Fclose(f_pointer);` где `f_pointer` – имя указателя файла. Вместе с закрытием файла получаем гарантию, что вся информация с зоны тампон, действительно была вписана в файл. Если программа оканчивается до закрытия файла, возможна ситуация, когда часть информации, которая не была вписана на диск, остается в зоне тампон и в результате теряется. Кроме этого, если файл не закрыт правильно, в его финале не будет вписан в необходимом виде код окончания файла(`eof`), и следующее открытие файла будет ошибочным. Итак, операционная система потеряет доступ к файлу. Кроме того, закрытие файла освобождает указатель, и после этого он может использоваться для доступа к другому файлу или для выполнения др. операций файла. Наприм.: пусть необходимо создать файл, вписать в него информацию, потом прочесть ее с файла. Необходимо отметить, что после вписания данных, файл необходимо закрыть и только после этого открыть его для чтения, имея так соответствующий доступ к данным файла.

```
#include<stdio.h>
```

```

void main (void) {
FILE * файл ;
if((файл =fopen(“info.txt”,”w”))==НОЛЬ ) {
puts(“Файл не может быть открыт\n”);
exit(); }
// Сюда будут перемещены операторы вписания информации в
файл
fclose(файл );
if(( файл =fopen(“info.txt”,”r”))==НОЛЬ ) {
puts(“ Файл не может быть открыт\n”);
exit();}
// Сюда будут перемещены операторы чтения информации с
файла
fclose(файл );}

```

Здесь файл открывается внутри условия if, т.е во время ввода контролируется возвращенное значение операционной системы открытия файла и, если возвращено нулевое значение, инициируется соответственное сообщение, и программа останавливается. Необходимо отметить, что некоторые компиляторы позволяют ввод данных в файл через очищение зоны тампон с помощью функции flush(). Эта функция позволяет без закрытия файла вписать всю информацию из зоны тамноп в файл, потом эта зона освобождается от каких –то данных.

## 12.2. Функции ввода / чтения с файла.

Язык C++ содержит больше возможностей передачи данных в файл и чтения из файла, в зависимости от использованной функции.

- Функция `fputc()`, `putc()` используются в случае записи одного символа в файл или вывода его на принтер.

- Функция `fgetc()`, `getc()` используется для чтения одного символа с файла.

- Для того, чтобы вписать последовательность символов в файл или вывести на принтер используется функция `fputs()`

- Для того, чтобы прочитать последовательность символов с файла, используется функция `fgets()`

- Функция `fprintf()` используется в случае формативного выхода символов, последовательности символов и цифр на диске или на принтере

- Функция `fscanf()` используется для чтения формативных символов, последовательности символов и цифр файла

- Ввод одной структуры в файл возможен, используя функцию `fwrite()`

- Чтение одной структуры из файла выполняется с помощью функции `fread()`

### 12.2.1. Ввод / чтение символов.

Ввод / чтение символов с файла - это главная форма работы с файлами. Хотя это не очень распространено, эти задачи хорошо знают главные принципы в работе с файлами.

Один символ может быть записан в файл с помощью `fputs()`, используя следующий синтаксис:

`fputs(v,fp)`: где `v` – переменная символьного типа (`char`) а `fp` – имя указателя файла.

Следующий пример выполняет запись нескольких символов в файл до того, как будет нажата `Enter`:

```
#include <stdio.h>
#include <conio.h>
void main(void){
FILE *f;
Char lit; clrscr();
F=fopen("info.txt","w")
printf("Избери несколько символов \n");
do{
lit=getch();
putch(lit);
fputs(lit,f);} while(lit!='\r');
fclose(f);}
```

Здесь файл открыт режимом доступа "`w`". И если во время открытия файла с именем "`info.txt`" его не существует, он будет создан. В цикле `do` выполняется чтение символов от клавиатуры с помощью функции `getch()` и их запись в файл с помощью `fputs()`. В этом случае можно использовать функцию `fputs()` с теми же параметрами. Цикл `do` будет продолжаться до тех пор, пока мы не нажмём кнопку `ENTER` и после этого файл будет закрыт.

Для чтения символов из файла используется функции `gets()` и `fgets()`.

Синтаксис функции `gets()` и `fgets()` следующий:

`ch_var=getcs(fp);` где `ch_var` – переменная а `fp` – указатель файла.

Следующий пример доказывает, как может быть прочитана информация из файла, созданого в предыдыдущем примере:

```
#include <stdio.h>
#include <conio.h>
void main(void){
FILE *fl; char lit;
clrscr( );
fl=fopen(“info.txt”,”r”);
printf(“Прочитанная информация :\ “);
while(lit=fgetch(fl))!=EOF)
printf(“%c”,lit);
fclose(fl); getch( );}
```

Файл открыт способом “r”, можно читать информацию. Цикл While, в котором читаются символы из файла до знака конец в файле EOF, который вписан в конце каждого файла в момент закрытия.

Функция чтения `fgetch(fl)` использует другой указатель в одном и том же файле – `fl`. Это значит, что для записи и чтения информации из файла используется одно и то же имя; запись или чтение рекомендуется выполнять с помощью разных указателей.

### 12.2.2 Запись/чтение последовательности.

В случае, когда необходимо ввести в файл множество символов, т.е. последовательность, используется функция `fputs(fl)`; которая имеет следующий синтаксис:

`fputs(s_var,fp);` где `s_var` – переменная а `fp` – указатель файла.

Функция `fputs( )` выполняет запись последовательности в файл или записывает на бумагу с помощью принтера без указания знака - конец линии. Для того, чтобы каждая записанная последовательность в файл начиналась с новой строки, необходимо использование знака конца линии.

Следующий пример записывает в файл несколько фамилий:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(void){
FILE *k; char фам [15];
clrscr( );
printf( “избери фамилию \n”);
gets(фам);
k=fopen(“фамилия.txt”,”w”);
while(strlen(фам)>0){
fputs(фам,k); fputs(“\n”,n);
printf(“избери следующую фамилию/n”);
gets(фам);}
fclose(k);}
```

Здесь цикл `While` будет повторяться до тех пор, пока не будет избрана последовательность с нулевой длиной.

Функция `fputs( )` запишет в файл последовательности с новой строки, благодаря записи `fputs(“\n”,w);`

В конце своей обработки файл `фамилия.txt` будет обязательно закрыт `fclose(k);`; надо заметить, что для вывода на бумагу выбранных фамилий

с помощью принтера необходимо указать имя файла “prn” в следующем виде:

`K=fopen(“prn”,”w”);` Для правильного вывода на экран необходимо использовать длину последовательности из 81 символов с целью, чтобы последовательность была избрана в пределах ширины экрана перед нажатием кнопки ENTER.

Чтение последовательности символов из файла выполняется функцией `fgets( )`, которая имеет следующий синтаксис:

`fgets(s_var,l,fp);` где `s_var` – переменная, `l` – переменная или константа целого типа, которая указывает возможное максимальное количество символов в последовательности, `fp` – указатель файла.

Следующий пример позволяет прочитать фамилии из файла и фамилию.txt, созданные в предыдущем примере.

```
#include <stdio.h>
#include <conio.h>
void main(void){
FILE *r; char name [15];
clrscr( );
r=fopen(“фамилия.txt”,”r”);
printf(“прочитанная информация из файла:\n”);
while(fgets(name 15, r)!=ноль)
printf(“%S”,name);
fclose(r); getch( );}
```

Здесь цикл `while` будет повторяться до тех пор, пока не будет указан код «конец файла». В случае, когда читаем информацию из файла, как последовательность символов, для указания конца файла используется нулевое значение, а код EOF используется при чтении символов.

Функция `fgets()` прочитает последовательность до кода „новая строка”, если его длина не больше “1-l”, указанной в параметрах функции. Обратите внимание, что функция `printf(“%S”,name);` не использует код “\n” для перехода к новой строке, потому что каждая последовательность файла содержит код “\n” – новая линия записана в файл в предыдущем примере с помощью функции `fputs(“n”,n);`

### 12.2.3. Вход/выход с форматом.

Функция для обработки символов и последовательностей имеет возможность записать и прочитать из файла только информацию. В случае, когда необходимо записать в файл данные, которые содержат числовые значения, используется функция `fprintf()`, которая имеет следующий синтаксис:

`fprintf(fp,format, data);` где `fp` – указатель файла, в котором выполняется запись, `format` – последовательность контроля формата записанных данных и `data` – список переменных или значений, которые должны быть записаны в файл. Пример:

```
fprintf(f,“%d”,cost);
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(void){
FILE *f; clrscr( );
char имя [20], ответ =’у’;
float(77) цена; int единицы;
f=fopen(“_____ .txt”,”w”);
```

```

while(ответ=='y'){
printf("избери название продукта\n");
scanf("% s",имя)
printf("избери цену продукта% s\n", имя);
scanf("% f",& цена);
printf("избери количество продуктов% s\n", имя);
scanf(, "% d",& единица);
printf(f, % s % f % d\n", имя, цена, единица);
ответ=getch( );}
fclose(f);}

```

В результате, в файл будет записана информация о нескольких продуктах, например:

```

дискета 4500000    100
мышь    140000000    3
монитор 2000      1

```

Обратите внимание на знак “\n” – конец строки, в конце последовательности контроля формата из функции fprintf(). Благодаря этому знаку, информация о каждом продукте записана в файле с новой строки.

Чтение информации с форматом из файла выполняется функцией fscanf(), которая имеет те же самые возможности, как и функция scanf( ), и использует следующий синтаксис:

```

scanf( fp, format, data): которая схожа с синтаксисом функции fprintf(
).
```

В следующем примере выполняется чтение информации из файла “produs.txt” про продукты из складов в файл предыдущего примера:

```
# include <stdio.h>
```

```

#include <conio.h>
void main(void){
clrscr( );
FILE *a; char имя [20];
float цена; int един;
a=fopen(“продукт.txt”,”r”);
while(fscanf(a, % s% f % d”, & имя, & цена, & един)!=EOF){
printf(“название: % s\n”, имя);
printf(“цена: % f\n”, цена);
printf(“количество: % d\n”, един);}
fclose(a); getch( );}

```

Здесь чтение происходит одновременно с проверкой условия в продолжении цикла while. Цикл while будет выполняться до того момента, пока не будет нажат код окончания файла, которым в этом случае будет EOF.

Данные из файла будут прочитаны в переменных имя, цена, единица, а потом выведены на экран.

#### 12.2.4. Файлы и Структуры.

Для того, чтобы записать переменную структурного типа в файл, используется функция fwrite( ), которая имеет следующий синтаксис:

fwrite(& struct\_var, struct\_size, n, fp); где:

& struct\_var – имя переменной с оператором адрес, который говорит адрес стартовой ячейки в памяти, где расположена структура.

struct\_var – размер структуры. Чтобы определить размер структуры используется функция sizeof(s); где s – имя переменной структурного типа.

n – целое число, которое определяет количество структур, которые будут записаны в файл с одной попыткой, здесь рекомендуется использовать значение 1. Значение больше 1 используется в случае, когда в файл записывается массив структур с одной попытки.

fp – имя указателя файла.

Пример: fwrite(&a, size(a), 1, f);

В следующем примере создан массив структур с информацией о группе студентов, который содержит имена студентов, год рождения и средний бал. В начале массив структур заполняется информацией, потом одна структура за другой с помощью цикла for и функцией fwrite( ) будут записаны в файл. Имя файла будет названо пользователем и будет храниться в переменной “filename” символьного типа.

```
#include <stdio.h>
#include <conio.h>
структ студ {
char имя [15];
int год; float средний бал;};
void main(void){ clrscr( );
структ студ x[10]; int, i, n;
File *f; char filename [12];
float m;
printf(“избери число студентов\n”);
scanf(“% d”, &n);
```

```

for(i=0; i<n; i++){
printf(“избери имя студента\n”);
scanf(“% s”, x[i] имя);
printf(“избери год рождения\n”);
scanf(“% d”, x[i] год);
printf(“избери средний бал\n”);
scanf(“% f”, &m); x[i] средний бал – m;}
printf(“избери имя файла\n”);
scanf(“% s”, filename);
f=fopen(“filename”,”w”);
for (i=0; i<n; i++) fwrite(& x[i], sizeof(x[i], 1, f);
fclose(f); getch( );}

```

В результате будет создан файл с именем, которое было присвоено переменной filename, и в него будут записаны структуры с информацией о студентах. Если мы откроем файл с помощью обыкновенного редактора текстов, то заметим в нём непонятное содержание.

На самом деле, информация, которая находится в этом файле, непонятна ЭВМ и может быть прочитанной с помощью функции fread(), у которой тот же синтаксис, что и у функции fwrite().

```
fread(& struct_var, struct_size, n, fp);
```

В следующем примере будет выполняться чтение всех записей из файла, созданного в предыдущем примере, и их вывод на экран. Надо заметить, что функция fread() в результате выполнения возвращает значение, которое соответствует количеству структур, удачно прочитанных из файла. В нашем случае структуры читаются по одной из файла, значит, функция вернёт значение 1, если будет удачна. В

случае создания или определения конца файла функция вернёт нулевое значение.

```
#include <stdio.h>
#include <conio.h>
структ студ {
char имя[15];
int год; float средний бал ;};
void main(void ) { clrscr( );
структ студ у[10];
FILE *к; char fn[12]; int i=0;
printf(“избери имя файла\n”);
scanf(“%s”, fn);
к=fopen(“fn”,”r”);
printf(“прочитанная информация из файла:\n”);
while(fread(&у[i], sizead(у[i]), 1, к)==1){
printf(“имя студента: %s\n”,[i]. имя);
printf(“год рождения: %s\n”, у[i]. имя);
printf(“средний бал:%f\n”, у[i]. средний бал); i++;}
fclose(к); getch( );}
```

Следующая таблица содержит описание всех возможностей ввода и чтения данных относительно файлов, включительно и значения, возвращенные в случае ошибочного чтения.

Таблица 9.

Тип данных	Функция выхода	Функция входа	Возвращёное значение при чтении
СИМВОЛЫ	Putc( ); fputc(	Getch( );	EOF

	);	fgetch( );	
Строка	Fputs( );	Fgets( );	НОЛЬ
Данные с форматом	Fprintf( );	Fscanf( );	EOF
структуры	Fwrite( );	Freadf( );	О

Приложение 1. Функции входа - выхода в С/С++ . Функции выхода в С.

Функции выхода в любом языке программирования имеют назначение передавать данные из памяти компьютера в другие места назначения выхода: экран, принтер, файл на диске и т.д. Во время выхода выполняется копия данных, которые будут переданы к выходу, а их оригинал сохраняется в памяти компьютера. В языке программирования С существует много функций выхода, и их использование зависит от типа данных, которые будут выбраны, и от метода их представления. Самый простой синтаксис существует у функций выбора с символами и их последовательностями.

Заметка:

Все функции выхода в языке С поддерживаются и языком с++.

Функция puts( ).

Функция puts( ) имеет назначение вывести на экран строку. Синтаксис этой функции следующий: puts(p); где P – параметр функции, который может быть:

- 1) буква
- 2) константа последовательного типа.
- 3) переменная последовательного типа.

Определение: буквой называется конкретная последовательность символов, которые входят в инструкции языка вместо имени константы или переменной. В случае использования буквы в качестве параметра для функции puts() строка, которая должна быть выведена на экран, будет входить в круглые скобки между кавычками. Пример: puts (“hello world!”); Использование константы последовательного типа в качестве параметра функции puts() пользуется следующими правилами:

```
#define MESAJ “hello world” void main(void)
puts(MESAJ);}
```

Фраза “hello world” была присвоена константе MESAJ , которая потом используется в качестве параметра функции puts() без кавычек.

Третий метод смотрите ниже:

```
void main(void) {char MESAJ []=”hello world”};
puts (MESAJ);}
```

Использование константы и переменных другого типа в качестве параметра функции puts() приведет к ошибке компилятора. Единственная разница между использованием констант, переменных и использованием букв в том, что буква находится в кавычках, а константы и переменные используются без кавычек. Множество компиляторов выполняют переход в новую строку после выполнения функции puts(). Это значит, что после вывода данных на экран, каретка автоматически переходит в новую строку, но это правило не выполняется всеми компиляторами. В этом случае, для перехода в новую строку нужно использовать знак “\n” – переход в новую строку. В этом случае функция puts() будет иметь следующий синтаксис: puts(“hello world!\n”);

Прототип функции puts( ) описан в библиотеке stdio.h

`#include <stdio.h>` для программы.

Функция `putchar( )`.

Имеет назначение выводить единственный символ на экран. Как и функции `puts()` в качестве параметра используется буква, константа или переменная символьного типа. Пример:

Буква в качестве параметра:

```
Putchar('c');
```

константа:

```
#define lit 'c' void main(void){ putchar(lit);}
```

Переменная символьного типа в качестве параметра:

```
void main(void){ char lit; lit='c' putchar(lit);}
```

С помощью функции `putchar()` можно вывести на экран только один символ. В случае инструкции `putchar('da');` будет ошибка.

Главное в использовании символов и строк состоит в том, что строка пишется в кавычках, а символы в апострофах. Множество компиляторов не выполняют переход каретки в новую строку после выполнения функции `putchar()`. Для того, чтобы перейти к новой строке, используется знак “`\n`”; некоторые компиляторы для вывода символов используют функцию `putch()`, у которой синтаксис тот же, что и у функции `putchar()`, описанной в библиотеке `stdio.h`.

Для программы используют `#include<stdio.h>`

Функция `putchar()` также описывается в библиотеке `stdio.h`.

Прототип функции `putch()` описывается в библиотеке `conio.h`, и её использование будет `#include<conio.h>`.

## Функция printf()

Функции `putch()` и `puts()` используются очень часто, но их возможности не велики. Эти функции не выводят на экран цифровое значение, и могут обрабатывать только один параметр. В языке C существует функция `printf()`. Она позволяет выводить на экран данные любого типа и может обрабатывать несколько параметров. С помощью этой функции можно определить форматирование данных, выведенных на дисплей. В самом простом случае эта функция может быть использована вместо функции `puts()` для вывода на экран последовательности символов:

```
#define MESAJ "hello world!";  
void main(void); printf(MESAJ); printf("добро пожаловать"); }
```

Как и в случае функции `puts()`, функция `printf()` может иметь в качестве параметра букву, константу или переменную символьного типа.

Для вывода на экран числовых значений и для форматирования разных типов данных, список параметров функции `printf()` делится на 2 части:

```
Printf("последовательности с форматом", список данных);
```

Первый параметр называется последовательностью управления или последовательностью формата. Этот параметр пишется в кавычках и указывает компилятору, из какого места последовательности должны быть показаны данные.

Последовательность с форматом может содержать любой текст вместе с ярлыками, называемыми указателями формата, которые определяют тип данных и их местонахождение. Любой указатель формата начинается со знака (%), после него следует символ, который

указывает тип данных. Существуют разные указатели : %d – целое число, %u – целое число без знака, %f - целое число типа float или double, %e – действительное значение в начальной форме , %g – реальное число, выведенное в формате %f или %e в зависимости от того, какая форма записи короче; %c – символ, %s – последовательность символов.

Таким образом первая часть функции printf может быть написана в следующей форме: printf(“%d”,...). Знак процент (%) говорит компилятору, что после него следует указатель формата, а для его вывода на экран символа (%) необходимо написать его 2 раза в следующей форме: printf(“%%”).

Буква ‘d’ указывает компилятору, что на экран будет выведено значение целого типа, а значение число будет написано в десятичной системе.

Вторая часть из списка параметров - это список данных, который содержит буквы, имя переменной или константы, значения которых необходимо вывести на экран.

Список данных разделяется от последовательности с форматом через запятую. Все элементы из списка данных также разделяются запятой. Когда компилятор обрабатывает эту функцию, он заменяет указатели формата значениями из списка данных. Пример:

Printf(“%d”, 5); Во время выполнения этой функции значение 5 будет вставлено вместо указателя формата (%d). Последовательность с форматом содержит не только указатель формата, но и простой текст, который может содержать указатели формата. Пример:

Printf(“яас %d”,5); В результате выполнения этого примера на экран будет выведено “5 часов!”, тот же эффект получится при

использовании функции puts(“5 часов”); но для комбинирования текста со значениями, константами и числовыми переменными необходимо использовать функцию printf( ). Пример:

```
void main(void){  
int час; час=5: printf(“час%d”,час);}
```

Этот пример использует в качестве параметра переменную целого типа час, которая находится в списке данных. Список данных может содержать несколько параметров. Пример:

```
void main(void){  
int часб мин; час=5; мин=25;  
printf(“час%d *n %d минуты”,час, мин);}
```

В ответе будет: “5 часов , 25 минут”. В случае использования нескольких параметров в списке данных, а также в последовательности с форматом, параметры из списка данных должны соответствовать по количеству, позиции и типу указателям из последовательности с формата. В нашем случае первый символ %d относится к переменной час, а второй – к переменной минуты.

В функции printf() можно использовать не только несколько параметров того же типа, но и параметры разных типов. Пример:

```
void main(void){  
int колич; float цена; цена=5.21; колич=3;  
printf(“цена %f лей для %d kg.”, цена количество);}
```

В ответе будет: “цена 5.210000 лей за 3 kg.”. Здесь, и в правду, параметры из списка данных соответствуют по количеству, позиции и типу указателям формата из последовательности с формата.

Функция `printf( )` не переводит автоматически каретку в новую строку для вывода данных на экран. Для перехода в новую строку используется знак “\n”, как и в функции `puts( )`. Пример:

```
printf(“цена %f лей \n для %d kg.”,5.21, 3);
```

После вывода на экран будет:

Цена 5.210000 для 3 kg.

Как знак ‘\n’ здесь могут использовать знаки: ‘\a’, ‘\b,’ ‘\t’, ‘\n’, ‘\r’, т.д.

Без использования указателя ширины полей цифры будут выведены в стандартном формате для определенного типа данных. Пример, реальная цифра будет выведена с 6 позициями после занятой. В предыдущем примере цена 5.210000 выведена с 6 позициями после цифры, этот формат можно изменить. Используя указатель ширины полей, можно определить количество позиций, на которых будет выведёно значение любого типа данных. Пример:

```
printf(“цена %2 f лей \n для %d kg.”, 5.21, 3);
```

Здесь реальное число будет выведено на экран на 2 позиции после запятой: 5.21.

```
printf(“цена %6.21f лей \n для % d kg/”, 5.21, 5_;
```

Получим \_\_\_\_5.21, где ‘\_\_\_’ - место. Указатель ширины полей для реальных чисел имеет следующую форму ‘%k, rf’, где k – окончательное число позиции (вместе с запятой), где будет выведена реальная цифра, r – число позиции после запятой. Если k будет больше чем r+2, тогда перед числом будут свободные места в количестве k-r+2. Под цифрой 2 здесь подразумевается 2 позиции из реального числа: позиция от целого числа до запятой, и позицией также является запятая. Если k будет меньше чем r+2, указатель ширины полей просто

будет игнорирован. Указатель ширины полей в форме %b.f представит цифру 5.21 как 5.2100. Для целых чисел, символов и последовательностей символов указатель ширины полей имеет форму %kd, %ks, %ks, где k – окончательное число позиций, где будет выведено значение. В то же время, если k будет больше реальной длины выведенного значения, в начале этого значения будут оставлены пропуски в количестве k-p, где p – число позиций, на которых реально находится выведенное значение. Если k будет меньше реальной длины значения, указатель ширины полей просто будет игнорирован. Пример:

```
printf(„час %3d”, 5); // в ответе “ час_ _5”;
```

```
printf(“моё имя %25”, Андрей); //в ответе “ моё имя Андрей”;
```

```
printf(“буква %2c; ‘A’); //в ответе “буква_A”;
```

где символ ‘\_\_\_\_\_’ означает место.

### Ввод информации в C++.

Все функции выхода, рассмотренные выше, действительны для языка C/C++. Но язык C++ содержит стандартный вид выхода “cont”, который вместе с оператором оставления, состоящим из 2 знаков (<<) меньше, позволяет вывод букв, значений констант и переменных без использования указателей формата.

Для использования cont необходимо вписать в текст программы C++ файл <iostream.h> с помощью директивы #include <iostream.h>, из-за того, что он содержит описание процессов входа и выхода в C++. Структура инструкции, которую использует cont следующая: "cont <<список\_данных;" , где знак ‘<<’ – оператор, который указывает компилятору необходимость вывода на экран списка данных, которые следуют после него. В качестве выведенной информации с помощью

cont могут служить буквы, имя константы и переменной любого типа. Используя один и тот же процесс выхода, можно вывести на экран несколько аргументов с условием, что они будут определены между собой оператором оставления. Пример:

```
# include <conio.h>
# include <iostream.h>
void main(void){ clrscr( );
const колия=3;
int час, мин, float сумма;
сумма=5.21; час=9; мин=20;
cont <<"hello world!\n";
cont <<"час"<<9;
cont <<"\n час"<<час<<"n"<<мин<<"минуты\n";
cont <<"цена"<<сумма<<"лей для"<<колич<<"kg\n";
getch( );}
```

Так же, как и функция printf(), cont не переводит каретку в новую строку. Для этого используется знак '\n', как показано в примере выше.

### Функции входа в С.

Процесс входа данных предполагает ввод информации для нормальной работы программы. Введенная информация располагается в переменных, это значит, что значения, полученные от пользователя, присваиваются значениям переменных, которые хранятся в памяти. Введение данных можно выполнить, откуда угодно: с клавиатуры, памяти, CD-ROM и т.д.

Вход данных - это процесс, который определяет дальнейшую работу программы. Исправление ввода данных сказывается и на исправлении

результатов, полученных с использованием данных. Введенные данные могут присваиваться как значения только переменным, но не константам, значения которых не меняются в течение выполнения программы. Если переменная, в которой будет записано значение от клавиатуры, уже имеет значения, тогда новое значение заменит старое.

Далее будут изучены возможности ввода от клавиатуры.

Функция `getchar()`

Функция `getchar` даёт возможность ввода от клавиатуры единственного символа. Большинство компиляторов не отличают значения типа `int` и типа `char` при использовании функции `getchar()`. Если переменная будет представлена, как переменная типа `int`, функция `getchar()` получит её в качестве значения ввода, и компилятор не выдаст ошибки. Даже значение символа, набранного от клавиатуры, будет изменено в целое число, равным коду этого символа в таблице символов ASCII.

Синтаксис использования этой функции следующий:

`var=getchar( );` где `var` – имя переменной, которая будет присвоена набранному символу с клавиатуры. Здесь функция `getchar()` вызвана, используя другой синтаксис в сравнении с функциями `puts()`, `putch()`, `printf()`. Эта запись обозначает: присвоить переменной с именем `var` полученное значение в результате выполнения функции `getchar()`. Функция `getchar()` не использует аргументов, из-за этого круглые скобки после имени функции остаются пустыми. После того, как пользователь избирает клавишу от клавиатуры, `getchar()` выводит введенный знак на экран. В этом случае не обязательно нажатие кнопки `ENTER` после выполнения функции, из-за того, что `getchar()` дает

возможность ввести только один знак, после чего программа переходит к дальнейшему выполнению.

Введенное значение присваивается переменной сразу же после того, как был избран любой символ. Во время использования функции `getchar()` в текст программы должен быть включен файл `<stdio.h>`, который содержит прототип этой функции. Некоторые компиляторы C и C++ используют функцию `getch()`, похожую на функцию `getchar()`. Описание прототипа функции `getch()` находится в файле `<conio.h>`, который должен быть включен в текст программы: `# include <conio.h>`.

Пример:

```
#include <stdio.h>
#include <conio.h>
void main(void){ char lit lit1; lit=getch( ); lit1=getch( ); putchar(lit);
putchar('\n'); putchar(lit);}
```

Во время выполнения программы компилятор не имеет указаний для остановки выполнения этого после выполнения всей программы. Это необходимо для анализа результатов.

Функции `getch()` и `getchar()` дают возможность получить этот эффект, используя следующий синтаксис:

```
getch() или getcha(). Пример:
#include <stdio.h>
#include <conio.h>
void main(void) { char lit, lit1; lit =getchar( );
putchar(lit); putchar('\n'); putchar(lit);
getch( );}
```

В этом случае, дойдя до конца программы и встретив функцию `getch()`, компилятор остановит выполнение программы до того момента, пока не будет нажата кнопка ENTER.

Заметка: Язык C/C++ содержит функцию `gets()` назначена для ввода от клавиатуры любой строки. Синтаксис и принцип работы этой функции будут изучены в теме "Последовательности символов".

Функция `scanf()`.

Функция ввода `scanf()` даёт возможность ввода в компьютер данных любого типа. Функция сканирует клавиатуру, определяет избранные клавиши, затем передает информацию с помощью указателя формата, который находится в её составе. Как и в случае функции `printf()`, `scanf()` может иметь больше аргументов, создавая возможность для ввода значений целого типа, символьного и последовательности символов одновременно, т.е в одной функции. Список параметров функции `scanf()` состоит из 2 частей, как и в функции `printf()`. Первая часть - это последовательность с форматом, а вторая часть- последовательность со списком данных. Последовательность с форматом содержит указатели формата, названные конверсиями символов, которые определяют способ, в котором должны будут выполнены входящие данные. Список данных содержит список переменных, в которых будут храниться введенные значения. Синтаксис функции `scanf()` следующий:

```
Scanf(“последовательность с форматом”, список данных);
```

Пример:

```
# include <stdio.h>
```

```
void main(void) {int a; float b;
```

```
printf(“избери значение a целое и b реальное\b”);
```

```
scanf(“%d%f”, &a, &b);
```

```
printf(“a=%d b=%f\n”, a,b); getchar( );}
```

Проанализируем функцию `scanf(“%d%f”, &a, &b);`

Используя функцию `scanf()`, во время ввода данных в списке данных необходимо указать адрес переменной, которой будет присвоено значение. Адрес памяти, забронированный переменной во время представления, можно узнать, используя оператор адреса '&'. Когда функция `scanf()` встречает формат переменной `a`, она его определяет как формат любого целого числа и сканирует клавиатуру, ожидая избрания целого числа, которое потом вписывает в адрес памяти, забронированного переменной `a`, занимая 2 байта. Цифра, избранная на клавиатуре, является значением этой переменной. Также происходит и с переменной реального типа `b`. После обработки всех переменных из списка параметров функция `scanf()` заканчивает работу в случае нажатия кнопки ENTER.

Процесс ввода в C++.

Компиляторы языка C++ поддерживают функции ввода `gets()`, `getchar()`, `scanf()`, про которые мы говорили выше. Кроме этих функции язык C++ содержит возможность ввода данных.

Стандартный вид `cin` с 2 знаками (`>>`) "больше", которые называются избранными операторами, даёт возможность входу данных любого типа с клавиатуры и имеет следующий синтаксис:

`cin>>var;` где `var` – имя переменной, которой будет присвоено значение, прочитанное с клавиатуры. Пример:

```
# include <stdio.h>
```

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

```
void main(void){int a; float b;  
printf("избери значения a целое и b реальное\n");  
cin>>a>>b;  
cout<<"a="<<a<<"b="<<b; getchar( );}
```

В этом примере с помощью ввода `cin` были сканированы с клавиатуры и присвоены значения переменным `a` и `b`. В этом случае не надо называть адрес памяти, где будет вписано сканированное значение, как в случае функции `scanf()`, и указывается только имя переменной. Кроме того, стандартная функция ввода автоматически определяет тип введенного значения, и не надо использовать указатель формата. Если в работе используем больше переменных, то надо их разделять с помощью запятой.

## Приложение 2. Математические функции

Как было сказано, стиль программирования языка C характеризуется большим числом функций, не очень объемных (обработка данных в этих функциях не зависит от других частей программы). Это дает возможность легко ввести любую поправку в некоторые функции, не задевая другие. В большинстве это необходимо в случае функций, созданных пользователем для деления задачи на несколько более простых задач. Но язык C содержит и множество стандартных функций, которые облегчают программирование. Эти функции предлагают множество решений, и каждый программист может составить собственную библиотеку функций, которые заменят коллекцию стандартных функций языка.

Язык программирования C содержит в своём арсенале инструкций очень мало инструментов для того, чтобы находится в первых местах в

вершине языков программирования высшего уровня. Большое преимущество обработки сложных математических выражений в языке С обязано библиотекам стандартных функций, которые облегчают решение некоторых очень сложных задач. Некоторые математические функции языка С встречаются чаще всего в программировании и будут изучены в этом параграфе.

Заметить, что для использования этих формул в программе в С или С++ необходимо ввести файл `math.h` с помощью директивы `#include<math.h>`

### 1. Функция `abs(x)`

Прототип: `int abs(int x); double fabs(double x ); long int labs(long int x);`

Эффект: Возвращает абсолютное значение числа `x`.

Пример:

```
# include <math.h>
# include <iostream.h>
# include <conio.h>
void main(void){ clrscr( );
int x, y;
cout<<"изберите значение x\n"
cin>>x;y=abs(x);
cout<<"содуль"<<x<<"="<<y; getch( );}
```

### 2. Функция `cos(x)`

Прототип: `double cos(double x); long double cosl(long double x);`

Эффект: Возвращает значение косинуса числа `x` [`cos(x)`];

Пример:

```

#include<math.h>
#include <iostream.h>
#include <conio.h>
void main(void){ clrscr( );
double x, y;
cout<<"изберите значение x\n";
cin>>x; y=cos(x);
cout<<"cosinus"<<x<<"="<<y; getch( );}

```

### 3. Функция $\sin(x)$

Прототип: `double sin(double x); long double sinl(long double x);`

Эффект: возвращает значение синуса числа  $x$  [ $\sin(x)$ ];

Пример:

```

#include<math.h>
#include<iostream.h>
#include<conio.h>
void main(void){ clrscr( ) ;
double x, y;
cout<<"изберите значение x\n";
cin>>x; y=sin(x);
cout<<"sinus"<<x<<"="<<y; getch( );}

```

### 4. Функция $\tan(x)$

Прототип: `double tan(double x); long double tanl(long double x);`

Эффект: возвращает значение тангенса числа  $x$  [ $\tan(x)$ ];

Пример:

```

#include<math.h>

```

```

#include<iostream.h>
#include<conio.h>
void main(void){ clrscr( );
double x, y;
cont<<"изберите значение x\n";
cin>>x; y=tan(x);
cont<<"tan"<<x<<"="<<y; getch( );}

```

### 5. Функция $\text{acos}(x)$

Прототип:  $\text{double acos}(\text{double } x)$ ;  $\text{long double acosl}(\text{long double } x)$ ;

Эффект: возвращает значение арккосинуса числа  $x[\text{arccos}(x)]$ ;

Примет:

```

#include<math.h>
#include<iostream.h>
#include<conio.h>
void main(void){ clrscr( );
double x, y;
cont<<"изберите значение x\n";
cin>>x; y=acos(x);
cont<<"arccosinusul"<<x<<"="<<y; getch( );}

```

### 6. Функция $\text{asin}(x)$

Прототип:  $\text{double asin}(\text{double } x)$ ;  $\text{long double asinl}(\text{long double } x)$ ;

Эффект: возвращает значение арксинуса числа  $x[\text{arcsin}(x)]$ ;

Пример:

```

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

```

```

#include <conio.h>
void main(void){clrscr( );
double x, y;
cont<<"изберите значение x\n";
cin>>x; y=asin(x);
cont<<"arcsinusul"<<x<<"="<<y; getch( );}

```

## 7. Функция atan(x)

Прототип: double atan(double x); long double atanl(long double x);

Эффект: возвращает значение арктангенса числа  $x$  [arctg(x)];

Пример:

```

#include <math.h>
#include <iostream.h>
#include <conio.h>
void main(void){clrscr( );
double x, y;
cont<<"изберите значение x\n";
cin>>x; y=atan(x);
cont<<"arctangent"<<x<<"="<<y; getch( );}

```

## 8. Функция log(x)

Прототип: double log(double x); long double logl( long double x);

Эффект: возвращает натуральный логорифм  $x$ ;

Пример:

```

#include<iostream.h>
#include<conio.h>
void main(void){clrscr( );

```

```
double x, y;  
cont<<"изберите значение x\n";  
cin>>x; y=log(x);  
cont<<"натуральный логорифм"<<x<<"="<<y; getch( );}
```

## 9. Функция log10(x)

Прототип: double log10(double x); long double log10l(long double x);

Эффект: возвращает десятичный логорифм числа x;

Пример:

```
#include <iostream.h>  
#include <conio.h>  
void main(void){ clrscr( );  
double x, y;  
cont<<"изберите значение x\n";  
cin>>x; y=log10(x);  
cont<<"десятичный логорифм"<<x<<"="<<y; getch( );}
```

## 10. Функция exp(x)

Прототип: double exp(double x); long double expl(long double x);

Эффект: возвращает значение  $e^x$ , где  $e=2.7$ , константа.

Пример:

```
#include <iostream.h>  
#include <conio.h>  
void main(void){ clrscr( );  
double x, y;  
cont<<"изберите значение x\n";  
cin>>x; y=exp(x);
```

```
cont<<"его экспонента"<<x<<"="<<y; getch( );}
```

## 11. Функция ldexp(a,b)

Прототип: double ldexp(double a, int b); long double ldexpl(long double a, int b);

Эффект: возвращает значение  $2^b * a$ ;

Пример:

```
#include <iostream.h>
#include <conio.h>
void main(void){ clrscr( );
double a,y;
cont<<"изберите значение a\n";
cin>>a; y=ldexp(a,3);
cont<<"2 в 3 степени умножить на"<<a<<"="<<y; getch( );}
```

## 12. Функция frexp(x,y)

Прототип: double rexp(double x, int\*y); long double frexpl(long double x, int\*y);

Эффект: возвращает значение  $x * 2^y$ , вычисляя и значение y;

Пример: Как неизвестная здесь используется только переменная x. Имея  $x=8$ ,  $k=frexp(x,y)$ , вычисляется реальная цифра (k), которую надо умножить на  $2^y$  для того, чтобы получить результат  $x=8$ , и, в то же время, вычислить y(значение степени, в которую надо возвести цифру 2). В нашем случае  $x=8$  и  $k=frexp(x,y)$ ; получим следующий ответ:  $y=41$ ;  $k=0.5$ ; именно  $0,5=8/24$ ж

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

```

#include <stdio.h>
int main(void) {clrscr( );
double k,x; int y;
x=8;
k=frexp(x&y);
printf(“число%f будет получено в ответе\b”,x);
printf(“умножение 2 в степень%d и%f “, y, k); getch( );}

```

### 13. Функция pow(x,y)

Прототип: double pow(double x, double y); long double powl(long double x, long double y);

Эффект: возвращает значение  $x^y$ ;

Пример:

```

#include <iostream.h>
#include <conio.h>
void main(void){clrscr( );
double x, y, k;
cout<<”изберите значение x\n”;
cin>>x;
cout<<”изберите значение y\n”;
cin>>y;
k=pow(x,y);
cout<<x<<” в степень ”<<y<<”=”<<k; getch( );}

```

### 14. Функция pow10(x)

Прототип:

Double pow10(int x); long double pow10l(int x);

Эффект: возвращает значение  $10^x$ ;

Пример;

```
#include <iostream.h>
#include <conio.h>
void main(void){ clrscr( );
double y; int x;
cout<<"изберите значение x\n";
cin>>x; y=pow10(x);
cout<<"10 в степени"<<x<<"="<<y; getch( );}
```

### 15. Функция sqrt(x)

Прототип: double sqrt(double x);

Эффект: возвращает значение  $\sqrt{x}$

Пример:

```
#include<iostream.h>
#include<conio.h>
void main(void){ clrscr( );
double y, x;
cout<<"изберите значение x\n";
cin>>x, y=sqrt(x);
cout<<"квадратный корень из"<<x<<"="<<y; getch( );}
```

### 16. Функция ceil(x)

Прототип: double ceil(double x); long double ceil(long double x);

Эффект: возвращает значение ROUND UP числа x, а потом округляет x в верхней части;

Пример:

```

#include <iostream.h>
#include <conio.h>
void main(void){ clrscr( );
double y, x;
cont<<"изберите значение x\n";
cin>>x; y=ceil(x);
cont<<"округленное значение"<<x<<"="<<y; getch( );}

```

### 17. Функция floor(x)

Прототип: double floor(double x); long double floorl(long double x);

Эффект: возвращает значение ROUND DOWN числа x, а именно округляет x в нижней части;

Пример:

```

#include <iostream.h>
#include <conio.h>
void main(void){ clrscr( );
double y, x;
cont<<"изберите значение x\n";
cin>>x; y=floor(x);
cont<<"округленное значение"<<x<<"="<<y; getch( );}

```

### 18. Функция fmod(x,y)

Прототип: double fmod(double x, double y); long double fmodl(long double x, long double y);

Эффект: возвращает остаток от деления x на y;

Пример:

```

#include <iostream.h>

```

```

#include <conio.h>
void main(void){clrscr( );
double y, x, k;
cont<<"изберите значение x\n";
cin>>x;
cont<<"изберите значение y\n»;
cin>>y;
k=fmod(x,y);
cont<<"остаток от деления"<<x<<"на"<<y<<"="<<k; getch( );}

```

### 19. Функция modf(x,y)

Прототип: double modf(double x, double \*y); long double modfl(long double\*x);

Эффект: возвращает дробную часть реального числа x, сохраняясь в y а значения целой части реального числа x;

Пример:

```

#include <iostream.h>
#include <conio.h>
void main(void){clrscr( );
double y, x, k;
cont<<"изберите значение x\n";
cin>>x;
k=modf(x, &y);
cont<<"цифра"<<x<<"целое число="<<y<<"и дробная часть="<<k;
getch();}

```

## 20. Функция div(x,y)

Прототип: `div_t div(int x, int y); ldiv_t ldiv(long int x, long int y);`

где `div_t` представлено в следующей форме:

```
typedef struct {long int quot; /*целая часть
```

```
long int rem; /*остаток} div_t;
```

Заметка: необходимо ввести библиотеку `<stdlib.h>`.

Эффект: возвращает составленное значение типа структуры `div_t`, которая содержит 2 значения: целую часть и остаток от деления `x` на `y`.

Пример:

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

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

```
#include <stdio.h>
```

```
div_t a;
```

```
void main(void){ clrscr( );
```

```
a=div(16.3);
```

```
printf(“16div3=%d, остаток=%d\n”, a.quot, a.rem); getch( );}
```

## 21. Функция randomize(x)

Проводник генератора случайных чисел.

Прототип: `void randomize(void);`

Эффект: вводит генератор случайных чисел.

Заметка: необходимо ввести библиотеку `<stdlib.h>`;

Пример: используется вместе с функцией `random(x)`.

## 22. Функция random(x).

Генератор случайных чисел.

Прототип: `int random(int x);`

Эффект: возвращает целое значение в интервале от 0 до (x-1);

Заметка: необходимо библиотека <stdlib.h>;

Пример:

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
void main(void){int I, clrscr( );
randomize ( );
for(i=50; i<60; i++)
printf("случайное число %d\n", random(i)); getch( );}
```

### 23. Функция rand( )

Прототип: int rand(void);

Эффект: возвращает целое случайное число в интервале от 0 до RAND\_MAX, где RAND\_MAX зависит от реализации языка.

Заметка: необходима библиотека <stdlib.h> не требует ввода

Пример:

```
#include<stdlib.h>
#include<conio.h>
#include<conio.h>
void main(void){int i; clrscr( );
for(i=0; i<15; i++)
printf("случайное число %d\n", rand( )); getch( );}
```

### Приложение 3. Функции, использованные при работе со строками.

Множество компиляторов C/C++ имеют специальные функции для работы с рядами. Для таких целей можно создать собственные функции, но выгоднее использовать функции из стандартных библиотек. Прототипы этих функций описываются в библиотеке `<string.h>`, и значит, для их использования, в программе должна быть введена эта библиотека, используя синтаксис: `#include<string.h>`. Одна из этих функций.

#### 1. Функция `strcat( )`

Прототип: `char*strcat(char*dest, const char*sursa);`

Эффект: Добавляет;

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char dest[50];
char sursa[5];
dessa="Turbo"; sursa="C++";
strcat(dest, sursa); puts(dest); getch( );}
```

Ответ будет: dest=Turbo C++

#### 2. Функция `strcmp( )`

Прототип: `int strcmp(const char*S1, const char*S2);`

Эффект: Сравнивает две строки;

Язык C не позволяет сравнить две строки в форме: `if(S1==S2);`

Здесь сравнения делается с помощью функции `strcmp()`, которая возвращает нулевое значение, в случае, когда строки одинаковы или одно значение не равно нулю, в случае, если строки не совпадают.

После выполнения функции `strcmp()`, будет возвращена целое значение, которое будет:

меньше 0, если  $S_1 < S_2$ ;

больше 0, если  $S_1 > S_2$ ;

равно 0, если  $S_1 = S_2$ ;

Пример:

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
void main(void){
char S1[30]; char S2[30]; int k;
gets(S1); gets(S2); k=strcmp(S1,S2);
if (k==0) puts(“S1 и S2 совпадают”); else puts(“не совпадают”);
getch( );}
```

### 3. Функция `strcmpi()`

Прототип: `int strcmpi(const char*S1, const char*S2);`

Эффект: Сравнивает 2 строки, несмотря на записную книгу СИМВОЛОВ.

Пример:

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
```

```

void main(void){
char S1[30]; char S2[30]; int k;
gets(S1); gets(S2); k=strcmp(S1,S2);
if(k==0) puts(S1 и S2 совпадают); else puts(“не совпадают”);
getch( );}

```

#### 4. Функция strncmp( )

Прототип: int strncmp(const char\*S<sub>1</sub>, const char\*S<sub>2</sub>, size\_tk);

Эффект: Функция strncmp( ) сравнивает одно число данное символами с 2 переменными последовательности.

Пример:

```

#include <string.h>
#include <stdio.h>
#include <conio.h>
void main(void){
char a[40]=”Turbo”, b[40]=”Pascal”; int k;
k=strncmp(a,b,1); printf(“%d”,k);}

```

Здесь будет сравнен первый символ последовательности a с первым символом из строки b.

После выполнения функции strncmp( ), целое значение будет возвращено:

меньше 0 если a<b;

больше 0 если a>b;

равно 0 если a=b;

Здесь будет ответ k>0;

#### 5. Функция strcmpi( )

Прототип: `int strncmpi(const char*S1, const char*S2, size_tk);`

Эффект: Сравнивает число из символов, начиная с первого из 2 строки не отличая прописные символы от заглавных.

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[40]="Turbo", b[40]="Pascal"; int k;
k=strncmp(a,b,1); printf("%d",k);}
меньше 0 если a<b;
больше 0 если a>b;
ровно 0 если a=b;
Здесь будет ответ k>0;
```

## 6. Функция `strlen( )`

Прототип: `size_t strlen(const char*S);`

Эффект: Определяет длину строки S;

Во многих случаях длина строки не совпадает с длиной массива, в котором находится строка. Значит длина массива больше. Функция `strlen( )` определяет длину строки и возвращает значение типа равному количеству символов, которые находятся в строке.

Пример:

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

```
char name[20]; int k;  
puts("введи имя"); gets(name);  
k=strlen(name); printf("Ваше имя %d символы", K);  
getch( );}
```

## 7. Функция strcpy( )

Прототип: char\*strcpy(char\*S<sub>1</sub>, const char\*S<sub>2</sub>);

Эффект: Копирует строку S<sub>2</sub> в строку S<sub>1</sub>. После выполнения функции strcpy (S<sub>1</sub>, S<sub>2</sub>); строка S<sub>1</sub> потеряет начальное значение и приобретет новое значение из строки S<sub>2</sub>. А S<sub>2</sub> останется неизменна.

Пример:

```
#include<string.h>  
#include<stdio.h>  
#include<conio.h>  
void main(void){  
char name[30]; char fam[30];  
puts("Введите имя"); gets(name);  
puts("Введите фамилию"); gets(fam);  
strcpy(name, fam);  
puts(name); puts(fam); getch( );
```

В завершение этого примера имя совпадает с фамилией.

## 8. Функция strchr( )

Прототип: size\_t strchr(const char\*S<sub>1</sub>, const char\*S<sub>2</sub>);

Эффект: Определяет местонахождение символа из строки S<sub>1</sub>, который первый встречается в строке S<sub>2</sub>. Возвращает значение целого типа равного местонахождением положением этого символа.

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char name[20] fam[20]; int k;
puts("Введите имя"); gets(name);
puts("Введите фамилию"); gest(fam);
k=strcspn(name, fam);
printf("Символ %с из % S первым был найден %S", name[k], name,
fam);
getch( );}
```

Если в данном случае name="Степан" и fam="Иванов", тогда результат функции k=strcspn(name, fam); будет k=4, потому что на четвёртом месте в слове "степан" находится буква "а", которая первой была найдена в "иванов".

## 9. Функция strspn( )

Прототип: size\_t strspn(const char\*S<sub>1</sub>, const char\*S<sub>2</sub>);

Эффект: Определяет местонахождение символа из последовательности S<sub>1</sub>, начиная с того, что S<sub>1</sub> отличается от S<sub>2</sub>, возвращает значение целого типа равного положению этого символа.

Пример:

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

```

char name[20]; fam[20]; int k;
puts("Ведите имя"); gets(name);
puts("Введите фамилию"); gets(fam);
k=strcspn(name, fam);
printf("Начиная с знака %C, последовательность %S отличается от
%S", name[k], name, fam);
getch( );}

```

Вслучае name="Стеан" и fam="Стоянов", k=2, потому что, начиная с символа с номером 2, переменная name отличается от переменной fam.

## 10. Функция strdup( )

Прототип: char\*strdup(const char\*S);

Эффект: Удваивает строку S. В случае удачи функции strdup( ), возвращает как значение указателя адреса в памяти, который содержит удвоенную строку и возвращает нулевое значение в случае ошибки. Функция strdup(S) делает копию строки S, получив пространство через вызов функции malloc( ). После использования удвоенной строки, программист должен освободить память для его размещения.

Пример:

```

#include<alloc.h>
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[ ]="УТМ"; char*b;
b=strdup(a); puts(b); free(b);}

```

Здесь “а” - строка, а “b” указатель адреса в памяти, где будет записана удвоенная строка. При выполнении функции `strdup( )`, значение строки из “а” будет копировано в часть памяти, адрес которой находится в “b”.

Функция `puts( )` выводит на экран удвоенную строку. Функция `free( )` освобождает память занимаемую удвоенной строкой.

### 11. Функция `strlwr( )`

Прототип: `char*strlwr(char*S);`

Эффект: Переводит все заглавные символы их строки *S* в прописные. В качестве параметра функция использует переменную типа строки. В результате выполнения этой функции, если в строке есть заглавные символы, они будут превращены в прописные, а если в строке заглавных символов нет – строка не меняется.

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[10]=”Pascal”;
strlwr(a);
puts(a); getch( );}
```

### 12. Функция `strupr( )`;

Прототип: `char*strupr(char*S);`

Эффект: Превращает все символы из строки в заглавные.

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[10]="Pascal";
strupr(a);
puts(a); getch( );
```

### 13. Функция strncat( )

Прототип: char\*strncat(char\*dest, const char\*sursa, size\_tk);

Эффект: Функция strncat( ) добавляет число равное k символами с начала последов sursa до конца последов dest.

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[20]="Turbo"; b[10]="Pascal";
strncat(a,b,3);puts(a);}
```

В результате выполнения этого примера, первые 3 символа b[10] будут добавлены к последнем a[20]. В результате функция strncat( ) будет строка и будет храниться в переменной a. После выполнения примера, переменная a будет содержать значение "Turbo Pascal".

### 14. Функция strncpy( )

Прототип: char\*strncpy(char\*dest, const char\*sursa, size\_tn);

Эффект: Копирует число, данное символом из одной последовательности в другую.

Пример:

```
#include<strinf.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[40]="Turbo"; b[40]="Basic";
strncpy(a,b,2); puts(a);}
```

Функция `strncpy( )` выписывает число `N` данное символом из последовательности в начале последовательности. Если последовательность будет длинее чем `N` , тогда результат будет иметь начало, равное с копированными символами, а конец начальный. Значение результата будет находиться в последовательности назначения.

#### 15. Функция `strnset( )`

Прототип: `char*strnsct(char*S int ch, size_tn);`

Эффект: Функция `strnsct( )` копирует символ `ср` в первые `n` мест из последовательности `*S`. В случае, когда `n>strlen(s)`; тогда `n=strlen(s)`;

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char a[15]="студент", b=w;
strnset(a,b,3); puts(a);}
```

В выполнении функции `strnset( )`, будет возвращено значения типа последовательности, которая будет выписана в нужную последовательность.

```
a="WWW dent".
```

## 16. Функция `strrev( )`

Прототип: `char*strrev(char*S);`

Эффект: Функция `strrev( )` меняет последовательность символов `S`. После выполнения функции `strrev( )`, первый символ из последовательности будет заменён последним символом, 2 символ с предпоследним, и т.д. кроме нулевого символа.

Пример:

```
#include<string.h>
```

```
#include<stdio.h>
```

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

```
void main(void){char S1[10]="Студент";
```

```
printf("Начальная последовательность-%S\n", S1);
```

```
strrev(S1);
```

```
printf("Финальная последовательность - %S\n", S1);
```

После выполнения программы будет - тнедтс.

## 17. Функция `strstr( )`

Прототип: `char*strstr(const char*S1, const char*S2);`

Эффект: Функция `strstr( )` определяет последовательности символа из `S1`, начиная с которой была последовательность `S2`;

Пример:

```
#include<string.h>
```

```

#include<stdio.h>
#include<conio.h>
void main(void){
char S1[20], S2[20],rez ;
S1="интернационал"; S2="национ";
rez=strstr(S1, S2); printf("Последовательность: %S", rez);}

```

Ответ: "подпоследовательность: националь ". Если последовательность S<sub>2</sub> не была найдена в S<sub>1</sub> функция strstr( ) возвращает значения "ноль".

### 18. Функция strchr( )

Прототип: char\*strchr(const char\*S, intc);

Эффект: сканирует последовательность S в поисках символа с. В случае удачи функция возвращает идентификатор символу из S, который первый был найден и схож с символом с. В случае ошибки (если такого символа не существует в последовательности S) функция возвращает нулевое значение.

Пример:

```

#include<string.h>
#include<conio.h>
#include<stdio.h>
void main(void){
clrscr( );
char S[15];
char*ptr, c='p';
strcpy(S, "Turbo C++");
ptr=strchr(S,c);

```

```

    if(ptr){printf(“Символ%c находится %d в последовательности%s\n”,
с, ptr – S,S);
    puts(ptr);}
    else printf(“Символ не был найден\n”);
    getch( );

```

В завершении этого примера на экран будет выведен результат: символ р имеет местонахождение 2 в последовательности Турбо С++, и потом, благодаря функции puts(ptr); будет выведена последовательность S символа р, к которому указывает ptr: “Turbo С++”.

## 19. Функция strerror( )

Прототип: char\*strerror(int errnum);

Эффект: Определяет ошибку по номеру ошибки и выдаёт идентификатор последовательности с символами, которые описывают ошибку.

Пример:

```

#include<stdio.h>
#include<conio.h>
#include<errno.h>
void main(void){\
clrscr( );
char*numerr;
numerr=strerror(11);
printf(“Ошибка:%S\n”, numerr);
getch( );}

```

В выполнении этого примера функция `strerror(11)`; определяет ошибку, функция `printf` выводит на экран: ошибка: Invalid format.

## 20. Функция `strpbrk()`

Прототип: `char*strpbrk(const char*S1, const char*S2);`

Эффект: Функция ищет в последовательности  $S_1$  первый символ, который существует в последовательности  $S_2$ ;

В случае нахождения функция возвращает идентификатор первому символу из  $S_1$  находившемуся в  $S_2$ .

Пример:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main(void){
clrscr( );
char*S1="Университет";
char*S2="Молдова";
char*ptr;
ptr=strpbrk(S1,S2);
if(ptr) printf("Первый символ из S1 находится в S2: %c\n",*ptr);
else printf("Символ не найден\n");
getch( );}
```

Ответ: будет найден символ "v".

## 21. Функция `strchr()`

Прототип: `char *strchr(const char*s, int c);`

Эффект: Функция ищет последнее появление символа *c* в последовательности *S*. В случае удачи функция возвращает идентификатор последнему символу *S* схожим с символом *c*.

Пример:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main(void){
clrscr( );
char S[15];
char*ptr, c='r';
strcpy(S,"Программирование");
ptr=strchr(S,c);
if(ptr){printf("Символ %c находится: %d\n",c,ptr-S);
puts(ptr);}
else printf("The character was not found\n");
getch( );}
```

В результате выполнения этого примера на экран будет выведен результат. Символ *r* находится на 8 месте благодаря функции `puts(ptr)`; будет выведена последовательность *S* последнего символа *r*, к которому указывает `ptr: "re"`.

## 22. Функция `strset( )`

Прототип: `char*strset(char*s, int ch);`

Эффект: Меняет все символы из последовательности *S* в значение символа *c*. Финальный результат сохранится в последовательности *S*.

Пример:

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
void main(void){
char S[10]="студент";
printf("Начальная последовательность - %S\n",S);
strset(S,'a');
printf("Финальная последовательность - %S\n", S);
getch();}
```

Ответ: Начальная последовательность – студент,финальная последовательность – ааааааа.

## Библиография

1. Negrescu L. Limbajele C si C++ для начинающих – Bucuresti, 1996.
2. Cojocaru O. Turbo C++. – Кишинев, 1994.
3. Gremalschi A. Персональные компьютеры. – Кишинев, 1997.
4. Франка. C++. Учебный курс. – Санкт-Петербург, Издательство ПИТЕР, 2001.
5. Карпов. C++. Специальный справочник. – Санкт-Петербург, Издательство ПИТЕР, 2002.
6. Павловская. C/C++. Программирование на языке высокого уровня. – Санкт-Петербург, Издательство ПИТЕР, 2001.
7. Крупник. Изучаем Си. – Санкт-Петербург, Издательство ПИТЕР, 2002.
8. Balanescu T. Паскаль и Турбо Паскаль. – Bucuresti, 1992.
9. Anghel, Florin Stinga. Паскаль интересует всех. – Bucuresti, 1992.