Министерство образования Российской Федерации
РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ГИДРОМЕТЕОРОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

А.Д. Шишкин

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ Си

Учебное пособие

Рекомендовано к изданию
Редакционно-издательским советом РГГМУ

Санкт-Петербург
2003

УДК 519.682

Шишкин А.Д.. Программирование на языке Си. Учебное пособие. СПб.: Изд. РГГМУ, 2003.– 104 с.

Рецензент: А.И. Яшин, д-р техн. наук, проф. РЭТУ (ЛЭТИ).

Учебное пособие написано по разделу дисциплины «Методы и средства программирования». Даны основы программирования на языке Си.

Предназначено для подготовки морских инженеров по специальности 141000 – морские информационные системы и оборудование, 075600 – информационная безопасность телекоммуникационных систем, а так же для студентов других специальностей РГГМУ, желающих изучить язык Си.

ВВЕДЕНИЕ

Возможности ЭВМ как технической основы обработки информации связаны с используемым программным обеспечением (программами). Программы предназначены для машинной реализации различного рода задач. При программировании широкое применение имеют термины *задача* и *приложение*. *Задача* (problem, task) – проблема, подлежащая решению. *Приложение* (application) – программная реализация алгоритма решения задачи.

Написание программы предусматривает выполнение определенного числа действий, которые с большей или меньшей детализацией можно разделить на следующие важнейшие этапы:

- постановка задачи;
- выбор метода (алгоритма) решения задачи;
- написание программы на языке программирования Си;
- ввод исходного текста программы с помощью текстового редактора, текст оформляется в виде файла (модуля) с расширением .c или .cpp;
- компиляция модуля (или нескольких модулей вместе); на этом этапе получаем объектный файл, т.е. файл с расширением .obj;
- отладка синтаксиса программы;
- объединение откомпилированных модулей в программу (это часто называют компоновкой или линковкой программы); на этом этапе к программе присоединяют необходимые стандартные библиотеки и мы получаем выполняемый файл с расширением .exe;
- запуск программы на выполнение;
- отладка программы (тестирование программы и устранение ошибок);
- окончательное оформление программы.

При постановке задачи решаются (или уточняются) следующие проблемы:

- цель и назначение задачи, ее место и связи с другими задачами;
- условия и ограничения решения задачи;
- содержание функций обработки входной информации;
- требования к периодичности решения задачи;
- состав, форма и точность представления выходной информации.

Входная информация в задаче определяется как данные, поступающие на вход задачи и используемые для ее решения. Входные данные – это первичные данные документов ручного заполнения, информация, хранимая в файлах, базы данных, выходные результаты решения других задач и т.д.

Выходная информация может быть представлена в виде документов (листинга), файлов данных, сигналов управления выходными устройствами.

Алгоритм – система точно сформулированных правил, определяющая процесс преобразования допустимых исходных данных (входной информации) в желаемый результат (выходную информацию) за конечное число операций.

Алгоритм решения задачи имеет ряд обязательных свойств:

- дискретность – разбиение процесса обработки информации на более простые этапы (операции), выполнение которых ЭВМ или человеком не вызывает затруднений;
- определенность алгоритма – однозначность выполнения каждого определенного этапа преобразования информации;
- выполнимость – конечное число операций решения задачи;
- массовость – пригодность алгоритма для решения определенного класса задач.

В алгоритме отражается логика и способ формирования результатов решения с указанием расчетных формул, логических условий, соотношений для контроля достоверности выходных условий.

Алгоритм решения задачи и его программная реализация тесно взаимосвязаны. Чем детальнее описан алгоритм, тем проще его программная реализация.

Программа – результат интеллектуального труда, для которого характерно творчество. В любой программе присутствует индивидуальность ее разработчика. Вместе с тем программирование предполагает и рутинные работы, которые могут и должны иметь строгий регламент выполнения и соответствовать принятым стандартам.

1. КЛЮЧЕВЫЕ СЛОВА И СИНТАКСИС ЯЗЫКА

1.1. Основные элементы программирования

Большинство программ предназначено для решения прикладных задач. В программах это выполняется путем обработки информации. Для решения задачи программа должна выполнить следующие действия:

- ввести информацию в программу;
- запомнить введенную информацию;
- задать команды для обработки информации;
- передать информацию из программы пользователю,

Команды в программе должны быть организованы таким образом, чтобы:

- некоторые из них выполнялись только при истинном значении заданного условия (или набора условий);
- другие команды выполнялись многократно;
- отдельные команды разделены на части, и выполняемые из разных участков программы.

Итак, из выше описанных действий можно выделить семь основных элементов программирования: *ввод, типы данных, операции, вывод, условное выполнение, циклы и подпрограммы (функции для языка Си).*

В большинстве языков программирования имеются все указанные элементы. Во многих языках, включая Си, имеются и другие средства. Однако, наиболее быстрый способ изучения нового языка основан на понимании того, как в данном языке реализованы перечисленные основные элементы программирования.

1.2. Синтаксис языка Си

Есть шесть классов лексем: идентификаторы, ключевые слова, константы, строки, операторы и разделители.

Идентификаторы: последовательность букв и цифр произвольной длины. Первый символ-буква, например, NAME1, name_1, Name_1, name_1...

Ключевые слова: Они зарезервированы для языка и не могут использоваться другим образом. Список ключевых слов языка представлен в табл.1.1

Таблица 1.1

КЛЮЧЕВЫЕ СЛОВА ЯЗЫКА

Типы данных		Операторы	
Eng	Рус	Eng	Рус
Char	Символ	Asm	
Class	Класс	Break	Прерывание
Double	Двойной	Case	Выбор
Enum	Перечень	Continue	Продолжение
Float	Плавающий	Default	По молчанию
Int	Целый	Delete	Удалять
Long	Длинное целое	Do	Делать
Short	Короткое целое	Else	Иначе
Struct	Структура	For	Для
Union	Объединение	Go to	Идти (переход)
Unsigned	Беззнаковый	If	Если
Void	Пустой	New	Новый
Return	Возврат	Switch	Прерывать
Extern	Внешняя	Inline	В линию
Register	Регистр	Const	Постоянный
Overload	Переопределение	Typedef	Тип переменной
Auto	Автоматически	While	Пока
Static	Статический	Public	Открытый

Константы: целые, символьные, с плавающей точкой, строки.

Целая – последовательность цифр. *Константа* – восьмеричная, если начинается с 0, в противном случае – десятичная. Цифры 8 и 9 не являются восьмеричными цифрами. Последовательность цифр 0X или 0x воспринимается как шестнадцатеричное целое число. В шестнадцатеричные цифры входят буквы от A до F(f), имеющие значения 10 – 15.

Примеры: 0x53=83; 0xF=15; 0xA = 10.

Константа, значение которой превышает наибольшее машинное целое со знаком, считается длинной (long), в остальных случаях константы считаются целыми (int).

Длинные константы: десятичная, восьмеричная или шестнадцатеричная константа, за которой непосредственно стоит буква L (l), считается длинной константой.

Примеры: 121 – десятичная,

0123L – восьмеричная,

0xaaaal – шестнадцатеричная.

Перевод чисел из одной системы счисления в другую приведен в табл. 1.2.

Для представления шестнадцатеричного числа в двоичном коде надо заменить двоичной записью каждую цифру этого числа. Например, числа 0xAB01 и 0x53, представленные в двоичном виде, переводятся в десятичные по известной формуле перевода [1].

Таблица 1.2

ТАБЛИЦА ПЕРЕВОДА ЧИСЕЛ

Десятичное число	Шестнадцатеричное число	Двоичная запись числа	Восьмеричная запись числа	Шестнадцатерич-ная запись числа
0	0	0000	0	0x0
1	1	0001	01	0x1
2	2	0010	02	0x2
3	3	0011	03	0x3
4	4	0100	04	0x4
5	5	0101	05	0x5
6	6	0110	06	0x6
7	7	0111	07	0x7
8	8	1000	10	0x8
9	9	1001	11	0x9
10	A	1010	12	0xA
11	B	1011	13	0xB
12	C	1100	14	0xC
13	D	1101	15	0xD
14	E	1110	16	0xE
15	F	1111	17	0xF

A B 0 1

1010 1011 0000 0001 –> 1010101100000001=

$1 * 2^{15} + 0 * 2^{14} + 1 * 2^{13} + 0 * 2^{12} + 1 * 2^{11} + 0 * 2^{10} + 1 * 2^{9} + 1 * 2^{8} + \ldots 1 * 2^{0} = 43777;$

$0x53 = 01010011 = 0 * 2^{7} + 1 * 2^{6} + 0 * 2^{5} + 1 * 2^{4} + 0 * 2^{3} + 0 * 2^{2} + 1 * 2^{1} + 1 * 2^{0} = 83;$

Символьная константа состоит из символа, заключенного в апострофы (например, 'x'). Значением символьной константы считается численное значение константы в машинном наборе (алфавите). Символьные константы считаются данными типа *char*. Использование неграфических символов, однозначная кавычка ' и обратная косая \ производится в соответствии с табл.1.3

Таблица 1.3

ИСПОЛЬЗОВАНИЕ СПЕЦИАЛЬНЫХ СИМВОЛОВ

Название	Обозначение
Символ новой строки	\n
Горизонтальная табуляция	\t
Вертикальная табуляция	\v
Возврат на шаг	\b
Возврат каретки	\v
Перевод формата	\f
Обратная кавычка	\\
Апостроф	\'
Набор битов 0ddd	\ddd
Набор битов 0xddd	\xddd
Пустой символ	\0

Последовательность \ddd состоит из обратной косой, за которой следуют 1, 2, 3 – восьмеричные цифры, задающие значение требуемого символа. \xddd та же последовательность, где 1, 2, 3 – шестнадцатеричные цифры.

Константы с плавающей точкой (с E) используются для представления больших чисел в компактной форме.

Примеры: 345. = 345

3.14159

2.1E5 = 210000

.123E3 = 123

$4037e^{-5} = .04037$

Строки. Строка – это последовательность символов, заключенная в двойные кавычки "...". Строка имеет тип char, а класс памяти static. Она инициализируется заданными символами. Все строки, даже если они занесены одинаково, различны. Компилятор располагает в конце каждой строки нулевой (пустой) байт \0 с тем, чтобы сканирующая строку программа могла найти ее конец. В строке перед символом двойной кавычки должна стоять обязательно \. Кроме того, могут использоваться те же обозначения, что были описаны для символьных констант.

Примеры:

"Это строковая константа"

"A"

"Это одна *\

строка"

Имена и типы. Имя обозначает объект, функцию, тип, значение или метку. Имя может использоваться только внутри части текста программы, называемой его областью видимости. Имя имеет тип, определяющий его использование.

Объект – это область памяти. Объект имеет класс памяти, определяющий время его жизни.

Область видимости. Существует четыре ее вида: локальная, файл, программа и класс.

Локальная область- имя, описанное в блоке, локально в этом блоке и может использоваться только в нем. Исключения составляют метки, которые могут использоваться в любом месте функции, в которой они описаны.

Файл – имя, описанное вне любого блока. *Программа* – имя, описанное в файле, может использоваться в любом другом файле.

Класс – имя члена класса, локально для его класса и может использоваться только в функции-члене этого класса.

1.3. Стандартные математические функции

В языке Си для математических вычислений используются стандартные математические функции.

double cos(double x); – косинус;
double sin(double x); – синус;
double tan(x) – тангенс;
double log(double x); – логарифм натуральный;
double sqrt(double x); – корень квадратный;
double floor(double x); – ближайшее меньшее целое;
double ceil(double x); – ближайшее большее целое;
int abc(int i); – модуль целого числа;
double acos(double x); – арккосинус;
double fabs(duble x); – модуль числа с плавающей точкой;
double asin(duble x); – арксинус;
double atan(duble x); – арктангенс;
srand (seed) int seed; – инициализация генератора случайных чисел (ГСЧ) rand( ) и
int rand( ); – ГСЧ;
long int time(p), longint p – время в секундах, отсчитываемое от 1.01.1970 г. (0.00 по Гринвичу).
delay(t); – задержка во времени на t микросекунд;

double pow (double x, double y) и
long double pow (long double x, long double y) – возвращает значение, равное x^y.
double exp(double x) и
long double exp (long double x) – возвращает значение exp(x).

1.4. Стандартные библиотечные функции

Все стандартные функции имеют прототип в соответствующем заголовочном файле. В соответствии со стандартом языка ANSI в языке Си пятнадцать следующих заголовочных файлов присутствуют обязательно (табл.1.4):

Таблица 1.4

ТИПЫ ЗАГОЛОВОЧНЫХ ФАЙЛОВ

Заголовочный файл	Назначение
assert.h	Диагностика программы
ctype.h	Преобразование и проверка символов
errno.h	Проверка ошибок
floaf.h	Работа с числами с плавающей запятой
limits.h	Определение размеров целочисленных типов
locale.h	Поддержка интернациональной среды
math.h	Математические библиотеки
sefjmp.h	Возможности нелокальных переходов
signal.h	Отработка сигналов
stdarg.h	Поддержка функций с неопределенным числом аргументов.
stddef.h	Разное
stdio.h	Библиотека стандартов ввода/вывода
stdlib.h	Библиотека общего назначения
string.h	Функции работы со стройками символов
time.h	Функции работы с датами и временем
dos.h	Подключение Dos

Наиболее часто встречающиеся функции языка Си приведены в табл.1.5

Таблица 1.5

ФУНКЦИИ ЯЗЫКА Си

Функция	Перевод	Назначение
printf ( )	Принтф	Вывод на экран некоторой информации
cprintf ( )	Спринтф	Форматированный вывод на экран
main ( )	Майн	Определяет имя функции
scanf ( )	Сканф	Ввод с клавиатуры
getch ( )	Гетч	Ожидает, пока не будет введен с клавиатуры какой-либо символ
gets ( )	Гете	Читает символы с клавиатуры до тех пор, пока не будет нажата клавиша «Enter»
strcpy (S1,S2)	Стрикопи	Копирование содержимого строки S2 в строку S1
strcat (S1,S2)		Присоединяет строку S2 к строке S1 и помещает ее в массив строки S1. Строка S2 не меняется
strcmp (S1,S2)		Сравнивает строки S1 и S2. Результат равен 0, если S1 = S2; положительное решение, если S1 > S2; отрицательное значение, если S1 < S2
strlen (S)		Возвращает длину строки S символ \O в конце не учитывается
puts ( )	Патс	Выводит строку символов в stdout
putchar ( )	Патча	Выводит символ в stdout
cputs		Выводит строку на экран
putch ( )	Патч	Выводит символы на экран

1.5. Сводка операций языка Си

Все операции языка Си разбиты на категории (табл. 1.6). Каждая операция имеет свой приоритет, который убывает с ростом категории. Все операции одной категории имеют одинаковый приоритет. Унарные операции (категория 2), условная (категория 14) и присваивания (категория 15) ассоциируются (выполняются) справа налево. Все остальные операции ассоциируются слева направо.

Как видно из приведенных таблиц, язык Си богат на операции. Знак операции – это символ или комбинация символов, которые сообщают компилятору о необходимости произвести определенные арифметические, логические или другие действия.

Для каждой операции определено количество операндов и определенный порядок выполнения:

- один операнд – унарная операция, например унарный минус (–x), изменяющая знак;
- два операнда – бинарная операция, например, операция сложения (x + y) или вычитания (x – y);
- три операнда – операция условия ?:, такая операция только одна.

Таблица 1.6

СВОДКА ОПЕРАЦИЙ ЯЗЫКА Си

Категория	Операция	Название или действие
1. Наивысшего приоритета	( )	Вызов функции
	[ ]	Индексирование
	–>	Косвенное обращение к члену класса
	: :	Прямое обращение к члену класса
2. Унарные	.	Прямое обращение к члену класса
	!	Логическое отрицание
	~	Дополнение до единицы
	+	Унарный плюс
	–	Унарный минус
	+ +	Преинкремент или постинкремент
	– –	Предекремент или постдекремент
	&	Адрес
	*	Обращение
	size of	Размер
	new	Создание динамического объекта
	delete	Удаление
3. Мультипликативные	*	Умножение
	/	Деление
	%	Деление по модулю
4. Косвенное обращение	. *	Прямое косвенное обращение через указатель
	–> *	Косвенное обращение через указатель
5. Аддитивные	+	Бинарный плюс
	–	Бинарный минус
6. Сдвига	<<	Сдвиг влево
	>>	Сдвиг вправо
7. Отношения	<	Меньше
	<=	Меньше или равно
	>	Больше
	>=	Больше или равно
8. Равенства	= =	Равно
	!=	Не равно
9.	&	Побитовое И
10.	^	Побитовое исключающее ИЛИ
11.	|	Побитовое включающее ИЛИ
12.	&&	Логическое И
13.	||	Логическое ИЛИ
14.	? :	Условия

Каждая операция может иметь только определенные типы операндов. Например, операция побитового сдвига определена только для целочисленных операндов. Более подробно об операциях будет дано в следующих разделах.

2. БАЗОВЫЕ СРЕДСТВА ЯЗЫКА Си

2.1. Типы данных

Составление программы на языке Си предполагает выполнение следующих основных этапов (здесь предполагается, что задача поставлена, формализована и выбран алгоритм её решения):

1. Ввод и размещение в памяти ЭВМ исходных данных.
2. Задание последовательности операций над исходными данными в соответствии с выбранным алгоритмом решения задачи.
3. Вывод результатов решения задачи.

Ввод данных осуществляется с клавиатуры, с диска, либо с портов ввода /вывода.

Размещение в памяти ЭВМ требует указания типа переменных. По структуре данные разделяют на простые и составные (сложные).

Простые типы данных. Эти типы (табл.2.1) являются базовыми типами данных языка Си. На их основе формируются более сложные типы.

Таблица 2.1

ТИПЫ ПРОСТЫХ ДАННЫХ

Имя базового типа	Спецификация		Объём занимаемой памяти, байт
Целые	signed char	Знаковый символьный	1
	signed int	Знаковый целый	2
	signed short int	Знаковый короткий целый	2
	signed long int	Знаковый длинный целый	4
	unsigned char	Беззнаковый символьный	1
	unsigned int	Беззнаковый целый	2
	unsigned short int	Беззнаковый короткий целый	2
	unsigned long int	Беззнаковый длинный целый	4
Плавающие	Float	Плавающий	4
	Double	Плавающий 2-й точности	8
	long float	Длинный плавающий	8
	long double	Длинный плавающий 2-й точности	10
Прочие	Void	Пустой	
	Enum	Перечислимый	

При задании типов данных, т.е. при описании данных, если спецификация не используется, то компилятор предполагает тип int. Если не используется спецификация signed или unsigned, то предполагается знаковый тип.

Тип переменной определяет максимально – возможное число, которое может быть помещено в памяти ЭВМ. Самое маленькое отрицательное число, которое можно записать в 8 – ми разрядной сетке – (–128), в 16 – разрядной сетке (–32768).

Поэтому, например, если вычислить n! при n=8 мы получили бы число 40320. Оно значительно превосходит максимально возможное положительное число (32767), которое можно записать в 16-й разрядной сетке, что привело бы к переполнению разрядной сетки ЭВМ. Поэтому надо осторожно подходить к выбору типа int или же брать тип float, которое записывается в форме $N = m \cdot E \pm P$, где m-мантисса, E=10, P – целочисленный порядок, в пределах от –39 до +38.

В табл. 2.2. приведены значения чисел, которые могут быть записаны в выше перечисленных типах.

Таблица 2.2

ЧИСЛОВЫЕ ЗНАЧЕНИЯ ТИПОВ

Тип переменной	Количество бит	Диапазон чисел
shortint (знаковый)	8 бит (левый бит отведён под знак)	$-128 \leq a \leq 127$
int	16 (знаковый)	$-32768 \leq a \leq 32767$
longint	32 (знаковый)	$-2147483648 \leq a \leq 2147483647$

Составные (сложные) типы данных. К составным типам данных относятся:

- массивы – данные регулярной структуры;
- структуры – логически связанные данные разных типов.

Данные типы будут рассмотрены позднее подробно.

Особое место занимают данные типа *указатель.* Значением указателя является адрес расположения в памяти (или адрес памяти) простой переменной, массива, структуры либо функции. В языке Си аппарат указателей используется наиболее интенсивно.

Пример записи данных в программе.

```
int a,b,c;
float x,y;
char ch;
double e;
unsigned u;
и т.д.
```

2.2. Операции над данными

Операции над данными задаются с помощью операторов:

- присвоения;
- передачи управления по условию;
- организации циклов.

Эти операторы являются средствами организации линейных, разветвляющихся и циклических алгоритмов. Любая программа, кроме самой простой, состоит из вышеперечисленных базовых структур алгоритмов, определяемых соответствующими операторами.

Знаки операций (арифметических, отношения, логических, битовых) используются для объединения констант и переменных в соответствующие выражения.

Операции над данными предполагают наличие объектов некоторого типа и использование знаков операций.

Наиболее распространенной является операция *присваивания* "=". Она предназначена для изменения значений переменных, в том числе и вычислений «по формуле»

Например,

x =362;

k = k + 2;

m = c = 1;

Базовая форма

<имя>=<выражение>

Разновидность операции присваивания

<имя>=<имя> <знак операции> <выражение>

В отличие от других языков программирования, в Си применяется также компактная форма операции присваивания. В компактной форме последнюю запись можно представить так:

Примеры:

A = a + b то же, что a + = b,

A = a*b то же, что a * = b,

A = a*(3*b +10) то же, что a*=3*b+10 ,

i=i+1 тоже, что и i++.

Арифметические операции.

Различают унарные и бинарные операции.

Бинарными операциями являются:

+ сложение;

– вычитание;
* умножение;
/ деление;
% деление по модулю.
Унарные операции:
– унарный минус;
операции единичного приращения:
++ положительного (увеличения на единицу – инкремент),
— отрицательного (уменьшения на единицу – декремент).

Различают *апостериорное приращение*, например:
c = a+b++, что при пошаговом выполнении будет означать: c1=a+b; c2 = a+(b+1); c3=a+(b+2); и т. д.,
и *априорное приращение,* например c = a+++b, что при пошаговом выполнении будет означать: c1 = a+(b+1); c2 = a+(b+2)…

i++ и —i – это полноправные выражения.

Старшинство арифметических операций следующее:
++, —
– (унарный минус)
*, /, %
+, –

Операции, одинаковые по старшинству, выполняются в порядке слева направо.

Чтобы изменить порядок операций используют круглые скобки.

Операции над битами (с двоичными разрядами).

Бинарные:
Сдвиг влево a=b<<c;
Сдвиг вправо a=b>>c;
Операция “И” a=b&c;
Операция “ИЛИ” a=b|c;
Операция исключающее “ИЛИ” a=b^c;
Унарная
“НЕ” a=~b.

Помимо перечисленных операций, в языке Си используются операции отношения и логические операции. В языке Си нет данных логического типа. Поэтому принято соглашение, что если в результате логической операции получено значение не равное нулю (!=0), то результат трактуется как ”истина”. В противном случае – как “ложь”.

Примеры операций отношения:

a<b;a>b; a>=b; a<=b; a= =b; a != b,

где = = знак «равно»;

!= знак «не равно».

< меньше,

> больше,

>= больше или равно,

<= меньше или равно.

Примеры логических операций:

a&&b – операция логическое “И”,

a | | b – операция логическое “ИЛИ”,

! a – операция логическое “НЕ”.

2.3. Операции вывода данных

Ввод и вывод данных в языке Си осуществляется не с помощью встроенных операторов, как в других языках, а с помощью специальных программных модулей, называемых функциями, содержащимися в файлах -прототипах.

Вывод результатов счёта. Вывод результатов счёта осуществляется на стандартный терминал (stdout), на диск, на принтер, либо в порт ввода/вывода.

Следует сказать, что программа на языке Си состоит из функций, или как минимум из одной функции, называемой main(). Эта функция является главной функцией и любая программа начинает выполняться с её главного оператора.

Функции вывода данных. Основные функции вывода данных приведены в табл. 2.3

Таблица 2.3

ФУНКЦИИ ВЫВОДА ДАННЫХ

Имя функции	Описание	Файл, содержащий прототип
printf()	Производит форматированный вывод данных в stdout	stdio.h
puts()	Выводит строку символов в stdout	stdio.h
putchar()	Вводит символ в stdout	stdio.h
cprintf()	Осуществляет форматированный вывод на экран	conio.h
cputs()	Выводит строку на экран	conio.h
putch()	Выводит символ на экран	conio.h

Функции из файла-заголовка conio.h работают только на компьютере IBM PC. Их можно рассматривать, как дополнение к стандартным функциям. Их так же называют консольными.

Отличие стандартных функций от консольных заключается в том, что последние не преобразуют символы перевода строки \n в последовательность символов перевода строки \n и возврата каретки \r. Поэтому программист сам должен позаботиться об этом (см. ниже).

Обобщённая запись функции printf() следующая:

printf("строка форматов", обьект, обьект, ...);

Пример:

```
int y; // объявление целочисленной переменной
int x=5; //объявление и инициализация переменной
...
y=x+20; // операция присваивания
printf("получено число %d \n", y); //вывод числа
printf("получено число %d \n", x+20); //вывод значения выражения
printf("получено число %d %d \n", x,y); //вывод двух объектов
```

Здесь %d – спецификация поля представления целого числа;

x, y – объекты вывода (сами числа);

x + 20 – выражение, значение которого выводится;

// – признак строки- комментария.

В качестве объекта вывода используются константы, переменные, выражения, указатели функций.

В функции printf() используются следующие спецификации полей представления данных (табл. 2.4)

Таблица 2.4

СПЕЦИФИКАЦИИ ПОЛЕЙ ДАННЫХ

Формат (Спецификатор)	Типы вводимой информации
%d	Десятичное целое число
%i	Десятичное целое число со знаком
%c	Символ
%s	Строка символов
%f	Число с плавающей точкой
%u	Десятичное целое число без знака
%ld	Длинное целое
%p	Целое указателя
%o	Восьмеричное целое без знака
%x	Шестнадцатеричное целое без знака

Первые шесть форматов используются наиболее часто. Размер поля вывода задается либо по умолчанию, либо явно цифрой между знаком % и соответствующей буквой, например:

%–10s; %6d; %8.3f; %ld.

Здесь, так называемые, модификаторы имеют следующий смысл:

– печать с крайней левой позиции поля;

10 – задает максимальное число печатаемых символов строки;

6 – задает максимальное число печатаемых цифр;

8.3 – задает общую ширину поля (8) и число символов (3) после десятичной точки.

В строке форматов должно быть столько же спецификаторов полей вывода, сколько указано объектов. Спецификатор обязательно должен соответствовать типу объекта.

Пример распространенной ошибки:

```
int x;
float y;
 . . .
printf ("Получено x = %f y = %f \n", x , y);
```

Объекты int и float это объекты разных типов и должны иметь разные спецификаторы.

Пара символов \n, \r и т.п. называется управляющей последовательностью.

Назначение этих последовательностей следующее:

\n – переход на следующую строку;

\r – возврат каретки;

\t – табуляция;

\a – звонок;

\b – возврат на один шаг;

\f – перевод формата (страницы);

\ \ – вывод знака \ ;

\ ' – вывод знака ';

\ ''– вывод знака ''.

2.4. Функции вывода puts( ) и cputs()

Данные функции выводят строку символов на экран. Строка символов может быть определена в программе в виде константы путем заключения последовательности символов в двойные кавычки.

Например:

"Система Borland C"

Функция puts() выводит строку символов, а затем выполняет переход к следующей строке поля экрана.

Пример простой программы:

```
# include <stdio.h> / / Подключение функций ввода/вывода
# include <conio.h >
void main (void) / / Главная функция
{
printf("Система Borland C\n") ; / / Печать строки
puts("Система Borland C"); / / Печать строки
cputs("Система Borland C\n\r"); / / Печать строки
}
```

2.5. Задание окна вывода

Большинство программ связано с окнами (windows), а не со всем экраном. Окно – это прямоугольная область экрана, которую программа использует для выдачи сообщения.

Borland C++ позволяет устанавливать размер и местоположение окон на экране. Окном может быть весь экран или его маленькая область. По умолчанию областью вывода является весь экран. Если требуется установить окно меньшего размера нужно использовать функцию window(), а для очистки окна – функцию clrscr().

Приведем пример простой программы и рассмотрим ее более подробно:

Пример:

```
#include <conio.h>
int main (void)
{
clrscr( ); /* очистка текстового окна*/
window(10,10,60,20); /* задание нового окна */
cputs ("Это текст \r\n"); /* вывод текста */
getch( ); /* ждать нажатия клавиши */
return 0;
}
```

Рассмотрим каждую строку:

1. #include <conio.h> – директива препроцессора (текстового процессора), требующая подключить файл conio.h к нашей программе. Любая директива препроцессора начинается с символа # (номер).

2. int main (void) – здесь int – тип значения, возвращаемого функцией main, т.е. целое число;

main – имя функции;

(void) – означает отсутствие параметров у функции.

3. { – открывающая скобка – означает начало описания функции.

4. /* это комментарий */ – игнорируется компилятором.

5. clrscr ( ) – функция, очищающая текущее текстовое окно. Первоначально текстовое окно занимает все поле экрана.

6. window(10,10,60,20),

window(x1,y1,x2,y2) – функция, определяющая координаты текущего текстового окна. Окно задается координатами левого верхнего угла (x1,y1) и правого нижнего угла (x2,y2). В обычном режиме (C80) эти координаты задаются пределами: 1<= x <= 80; 1<= y <= 25.

7. cputs() – выводит строку текста, заключенную в двойные кавычки. \r\n –управляющие последовательности (см. табл.1.3).

8. getch( ) – функция, ожидающая нажатия любой клавиши; используется для временной остановки программы.

9. return 0 – оператор, с помощью которого функция возвращает в точку вызова некоторое значение.

10.} – закрывающая скобка функции main ().

Число открывающих и закрывающих скобок в программе должно быть всегда одинаковым.

2.6. Операции над адресами

Любое данное размещается в памяти ЭВМ и, следовательно, имеет некоторый адрес, обращаясь по которому можно извлечь данное из памяти. В языке Си определены две основные операции над адресами:

&– определение адреса операнда;

* – обращение по адресу.

Например, если bip – переменная типа int, то &bip – адрес переменной, по которому она расположена в памяти.

Если char *prt –указатель на данное типа char, то *prt – это само данное типа char (символ).

Пример операции над адресами:

```
# include <conio.h>
void main (void)
{
int bip;
char *prt;
bip = 2+3;
prt = ''Язык Turbo C\n''
cprintf (''bip = %d &bip = %p\n\r'', bip, &bip);
cprintf (''*prt = %c prt = %p\n\r'', *prt, prt);
}
```

В результате выполнения этой программы в первой строке будет выведено значение переменной bip, равное 5 и адрес, по которому это значение размещено в памяти. Во второй строке будет выведен символ "Я", т.е. первый символ и адрес, по которому этот символ расположен в памяти.

Для того, чтобы более четко уяснить механизм адресации к строке символов, рассмотрим этот вопрос более подробно.

Определение местоположения строки символов и адресация к ней осуществляется с помощью указателя.

Символьные массивы (строки). Вспомним, что в языке Си переменная типа указатель объявляется следующим образом:

int *a; char *b; и т.п.

После таких объявлений *a* и *в* трактуются как переменные – указатели на соответствующий тип, а **a* и **b* – как сами переменные данного типа.

Переменные типа указатель содержат адрес операнда, на который они ссылаются:

Пример:

```
# include <conio.h>
void main (void)
{
char *st;
st = "Язык программирования C\n"
cputs (st);
}
```

В результате обработки строки программы:

st =”Язык программирования С\n”

компилятор создает в тексте объектного кода машинный эквивалент этой строки, а переменной – указателю *st* присвоит адрес первого символа строки, т.е. адрес символа “Я”. Машинный код строки завершается специальным символом – терминатором или нулевым символом '\O'.

Команда cputs(st) воспроизведет строку символов на экране дисплея, начиная с того, который адресуется переменной st, вплоть до символа – терминатора.

Помимо определения строки символов в форме указателей существует и другая форма задания строки: в форме *массива символов*.

Пример:

```
# include <conio.h>
# include <string.h> /* файл заголовка функций, оперирующих со строками*/
void main (void)
{
char mas [24];/* описание массива из 24 символов */
strcpy (mas,''Язык программирования С "); /*копирование строки в массив */
cputs (mas);
}
```

По команде char mas [24] компилятор зарезервирует область памяти для 23 символов строки и еще один байт для размещения символа- терминатора '\O'.

Переменная mas несет информацию об адресе первого символа (нулевого элемента массива, т.к. массивы в Си нумеруются с 0: mas [0], mas[1],.. и т.д.).

Функция cputs() выведет на экран посимвольно все элементы массива. При этом каждый раз будет осуществляться проверка: не является ли очередной символ символом-терминатором.

Вот почему при размещении строки в массиве всегда необходимо отводить как минимум на один элемент больше для '\O', или определяйте необходимый размер в памяти по умолчанию:

```
char mas[]=''Строка символов"; /* Размер массива не задан */.
```

Отметим, что функция cputs() и strcpy() работают с формальными параметрами типа указатель. Их прототипы имеют следующий вид:

int cputs (char string);*

*char strcpy (char*dest, char* sours).*

Это значит, что при вызове этих функций им должны в качестве фактических параметров передаваться адреса. Но в нашей программе нигде не встречается символ &. В чем же дело?

В том, что при описании массива его имя трактуется как указатель на его первый элемент, т.е.

mas эквивалентно & mas[0]

2.7. Ввод данных в языке Си

Основные функции, осуществляющие ввод данных представлены в табл.2.5.

Таблица 2.5

ФУНКЦИИ ВВОДА

Имя Функции	Описание	Файл, содержащий прототип
Scanf()	Выполняет форматный ввод из потока stdin	stdio.h
Gets()	Получает строку символов из потока stdin	stdio.h
Getchar()	Вводит символ из потока stdin	stdio.h
Cscanf()	Выполняет форматный ввод с клавиатуры	conio.h
Cgets()	Считывает строку символов с клавиатуры	conio.h
Getch()	Вводит символ с клавиатуры без эхо-печати	conio.h

Рассмотрим краткое описание этих функций.

Функции scanf() и cscanf() используются для интерактивного ввода. Обобщенная форма записи этих функций следующая:

(c)scanf ("строка форматов", адрес, адрес,...);

В строке форматов применяются те же спецификаторы, которые были рассмотрены ранее для функции printf(): %d, %f, %c, %s (табл.2.4).

В отличие от функций printf() функция (c)scanf() требует указания в списке ввода не данных, а их адресов.

Пример:

```
/* ввод двух целых чисел в ячейки памяти "a" и "b"*/
(c)scanf ("%d %d", &a, &b);
```

где &a &b- адреса операндов "a" и "b".

Пример:

```
/* ввод строки символов, представленных массивом */
#include <conio.h>
void main (void)
{
clrscr ( );
char im [10];
cprintf ("Введите имя:" );
cscanf ("%s", im) ;
cprintf ("\n\r Ваше имя : %s \n\r", im);
}
```

Вспомним, что имя массива указывает на его первый элемент, поэтому перед im не ставится символ &.

Входной поток разбивается на отдельные поля с помощью специальных знаков: пробелов, символов табуляции.

Внимание: Для ввода строки, содержащей пробелы, функцию (c)scanf () не используют. Для этой цели предназначена функция (c)gets().

Функции gets() и cgets() читают строку символов, оканчивающуюся символом перевода строки. Символ перевода строки заменяется на символ -терминатор '\0'.

Пример:

```
# include <stdio.h>
#include <conio.h>
void main (void)
{
clrscr ();
char string [40];
printf ("Введите строку : ");
gets (string);
printf ("Строка = %s \n", string);
getch ( );
}
```

Функции getchar() и getch(). Функция getchar() предназначена для ввода одиночного символа. Возвращает считанный символ, преобразованный в целое число. Так как буфер stdin имеет размер в одну строку, то функция ничего не возвращает, пока не нажата клавиша "Enter".

Пример:

```
# include <stdio.h>
int main (void)
{
char c;
while ((c=getchar())!='\n')
printf ("%c", c);
return 0;
}
```

2.8. Поразрядные (побитовые) операции

Их можно производить с любыми целыми числами, переменными и константами. Их нельзя использовать с переменными типа float, double, long double.Поразрядными операциями являются:

& AND (и),
| OR (или),
^ XOR,
~ NOT (не),
<< сдвиг влево,
>> сдвиг вправо.

Операции AND, OR, NOT, XOR полностью соответствуют аналогичным логическим операциям. Поразрядные операции позволяют, в частности, обеспечить доступ к любому биту информации. Часто эти операции используют в драйверах устройств, программах, связанных с принтером, модемом и другими устройствами.

При выполнении поразрядной операции над двумя переменными, например типа char, операция производится над каждой парой соответствующих разрядов. Отличие поразрядных операций от логических и операций отношения состоит в том, что логическая операция и операция отношения всегда дают в результате 0 или 1. Для поразрядных операций это не так.

Приведем следующий пример:

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

ch = ch & 127;

Пусть ch = 'A'

'A' в двоичном коде: 11000001;

127 в двоичном коде: 01111111

'A' & 127: 01000001.

Если мы хотим установить для старшего разряда 1, удобна операция OR (или)

ch = ch | 128

'A' 11000001

127 10000000

'A' | 128 11000001.

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

Операция сдвига. Форма представления:

value >> число позиций; value<< число позиций.

Пример. Двоичное представление числа x = 9: 00001001.

Тогда x = 9<<3: 01001000,

x = 9>>3 : 00000001.

Пример. Пусть unsigned char x = 255 в двоичном виде 11111111.

Значения выражения сдвига будут иметь вид:

X = x<<3: 11111000

X = x>>3: 00011111

X = x>>5: 00000111.

3. УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА Си

3.1. Организация ветвящихся процессов: оператор if

Программа – это описание средствами языка программирования алгоритма решения задачи. Любой алгоритм можно разделить на следования, ветвления и циклы. В простейшем случае имеют место только следования.

Если имеет место проверка условий и выбор одного из возможных направлений продолжения вычислений, то имеет место ветвление (Рис. 3.1).

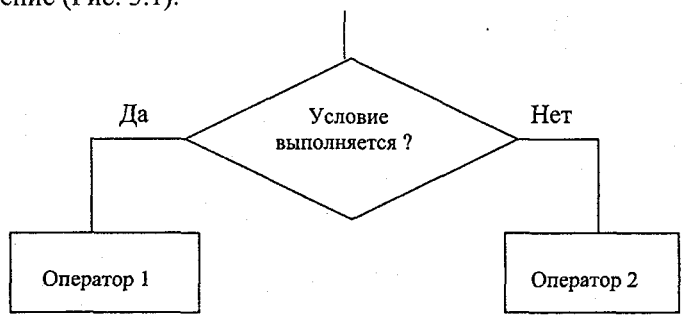


Рис. 3.1

Ветвления в программах осуществляются с помощью оператора условия **if.** Этот оператор используется в двух формах:

а) в полной форме:

if (выражение) (if – если, else – иначе)

оператор_1;

else

оператор_2;

Правило выполнения: если выражение в скобках (здесь некоторое условие) не равно нулю, т.е. "истина", то выполняется оператор_1. В противном случае выполняется оператор_2.

б) в сокращенной форме:

if (выражение)

оператор_1;

Если выражение в скобках (...) не равно нулю, то выполняется оператор_1, в противном случае управление передается следующему оператору программы.

Операторы_1 и _2 могут быть как простыми, так и составными. В последнем случае группа операторов должна заключаться в фигурные скобки:

```
{
оператор_1;
оператор_2;
.....
оператор_n;
}.
```

Каждый оператор внутри скобок должен заканчиваться ";".

Пример 3.1:

```
# include <conio.h>
void main ( void)
{
clrscr ();
float a, b, rez ;
printf ("Введите значение a и b: ");
scanf ("%f %f ", &a, &b);
if (b= =0)
printf ("Отношение a / b не определено \n ");
else
{
rez = a / b;
printf ("Отношение a / b равно %6. 3f\n", rez);
}
getch( );
}
```

Еще один фрагмент:

```
. . . . . .
if ( (ch = getch( ) )= ='g')
puts ("Конец работы \n");
else
puts ("Работа продолжается \n");
```

Выражение после слова if может состоять из нескольких операций присваивания, разделяемых запятыми. Значение всего выражения будет определяться последним присваиванием. Это справедливо для любого составного выражения. Например:

```
if (c=ch,ch=getch( ) ).
```

3.2. Вложение конструкции оператора if

При использовании вложенных конструкций действует правило: конструктор else всегда соответствует ближайшему слева конструктору if, не имеющему части else.

Пример 3.2:

```
if (x >1)
if (y= =2)
z =5;
else
z = 6;
```

Здесь часть else относится ко второму оператору if.

Пример3.3:

Программа функции sqn(x). Она вычисляет знак введенного числа x, т.е. sgn(x) принимает значение 1, если x > 0, значение –1, если x < 0, и значение 0, если x=0.

```
#include <stdio.h>
main()
{
int sqn;
float x;
printf ("Введите число");
scanf ("% f ", &x);
if (x>0)
{ sqn =1; printf ("число %f положительное sqn = %d \n ",x, sqn);}
if (x= =0)
 { sqn=0; printf ("число %f равно нулю sqn= %d \n", x, sqn);}
if (x<0) { sqn = –1; printf ("число %f отрицательное sqn= %d \n",
x, sqn);}
 }
```

Здесь мы не использовали оператор else.

Часто встречается необходимость использовать конструкцию **if – else – if:**

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

В этой форме условия проверяются сверху вниз. Как только какое-либо из условий принимает значение “истинно”, выполнится оператор, следующий за этим условием., а вся остальная конструкция будет проигнорирована.

Запишем фрагмент программы примера 3.3 с использованием оператора else.

```
.....
if (x>0) { sqn = 1; printf (“число %f положительное \n”, x); }
else if (x<0)
{ sqn= –1; printf (“число %f отрицательное \n”, x); }
else { sqn =0; printf (“число %f равно нулю \n”, x); }
```

3.3. Операторы организации цикла

Если в программе имеет место периодическое повторение некоторой последовательности действий, то говорят о наличии цикла.

Циклические вычисления в языке Си реализуются операторами **for ... , while ... , do ... while...** . Операторы цикла for – со счетчиком , while- с предусловием; do...while- с постусловием.

Оператор for имеет следующую конструкцию:

for (выражение_1 ; выражение_2 ; выражение_3)оператор;

где: выражение_1 устанавливает начальное значение параметра цикла;

выражение_2 определяет условие продолжения цикла;

выражение_3 задает правило модификации параметра цикла.

Каждое из этих трех выражений может быть групповым или может отсутствовать, в том числе и одновременно, но разделители ‘;’ обязательно должны быть. Если отсутствует выражение_2 , то оно считается истинным по умолчанию.

Конструкциии бесконечных циклов:

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

Тем не мене для таких циклов также может быть организован выход. Для этого используют оператор break, который будет рассмотрен несколько позже.

Пример 3.4:

Вычислим сумму первых n целых положительных чисел.

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

```
void main()
{
int n; // количество суммируемых чисел
int sum; // сумма
int i ; // счетчик циклов
printf("Вычисление суммы положительных чисел \n");
printf("введите количество суммируемых чисел \n");
scanf("%i", &n);
sum = 0;
for (i =1; i <= n; i++)
sum + = i;
printf(" Сумма первых % i целых положительных чисел \n");
printf(" равна % i", sum);
printf(" \n Для завершения нажмите <Enter>");
getch();
}
```

Наиболее универсальным является оператор цикла **while** (пока). Оператор while имеет следующую форму:

while (выражение)
оператор;

Пока выражение в скобках (. . .) не равно нулю, повторяется выполнение оператора (простого или составного).

Пример 3.5:

```
/*Определение длины строки */
#include <stdio.h>
void main (void)
{
 int dlina = 0 ;
 puts (" Введите строку , затем нажмите <Enter>") ;
 while (getchar () != '\n')
    dlina ++ ;
    printf ("\n Длина строки равна %d символам ", dlina) ;
}
```

Особенностью цикла while является то, что сначала проверяется значение выражения. Если оно равно нулю с самого начала, то цикл не выполнится ни разу, а управление будет передано следующему оператору. Это так называемый цикл с предусловием.

В Си используется также оператор цикла с постусловием **do ... while** .Форма его записи следующая:

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

Сначала выполняется тело цикла (оператор), а затем вычисляется значение выражения. Если оно равно нулю (истинно), то тело цикла выполняется снова. Этот процесс повторяется до тех пор, пока значение выражения не станет равным нулю. После чего управление будет передано следующему оператору программы.

Разница между циклами, while и do... while заключается в том, что при использовании конструкции do... while цикл выполнится хотя бы один раз.

Замечание. Операторы цикла for, while, do...while могут завершаться досрочно при выполнении в их теле операторов break , go to и return.

Вложенные циклы. Когда один цикл находится внутри другого, то говорят, что это вложенные циклы. Вложенные циклы часто используются при заполнении таблиц, в матричных вычислениях, обработке массивов.

*Оператор продолжения **continue.*** Оператор продолжения передает управление на следующую итерацию в операторах for, while , do ... while . Он может присутствовать только в теле этих операторов. Остающиеся в теле цикла операторы при этом не выполняются. В операторе for следующая операция начинается с вычисления выражения приращения, а затем происходит вычисление условного выражения. В операторах while и do ... while следующая операция начинается с вычисления условных выражений.

Пример 3.6:
Программа печатает натуральные числа, кратные 7.

```
#include <stdio.h>
main( )
{
int i;
for (i = 1; i <1000; i++)
{
for (i % 7) continue ; // Вложенный
```

```
printf ("%8d" , i); // цикл
}
}
```

3.4. Оператор передачи управления go to

В языке Си используется также оператор безусловного перехода **go to,** хотя без него можно обойтись в любой ситуации. Для использования оператора go to надо ввести понятие **метки (label).** Синтаксис:

go to <метка>;

...

<метка>: <оператор>;

Действие: Оператор go to передает управление непосредственно оператору, помеченному меткой. Область действия метки ограничивается функцией, в которой она определена, поэтому нельзя передать управление оператору, находящемуся в другой функции. Имя метки должно быть уникальным.

При последовательном выполнении операторов метка игнорируется. Она имеет смысл только для оператора go to.

По метке можно передать управление любому оператору функции, например, войти внутрь цикла или в конструкцию if. Это само по себе довольно опасно. Поэтому применяются метки только в тех случаях, когда это действительно оправдано.

Одно из полных применений оператора go to – это выход из влажных циклов.

```
for (... ) { whlie (...) { for (...) { . . . go to exit;
. . . . . .
} } } exit: printf ("Быстрый выход из вложенных циклов");
```

3.5. Оператор передачи управления (оператор-переключатель) switch

Этот оператор обеспечивает выбор возможного направления хода вычислений. Он используется тогда, когда в программе необходимо произвести выбор одного из нескольких вариантов.

Общая синтаксическая форма записи:

switch (выражение)

{

case константа_1: оператор(ы) ;

case константа_2: оператор(ы) ;

...

default: операторы (ы) ;

}

Правило выполнения: Сначала вычисляется значение выражения. Это значение сравнивается с константами во всех вариантах case. Управление передается оператору или группе операторов, константа которого совпадает со значением выражения. Если ни одна из констант не совпала со значением выражения, то выполняются операторы, связанные с меткой default. Если же конструкция default отсутствует, то ни один оператор не будет выполнен, а управление передастся следующему оператору программы.

Примечание: default – не обязательно последний; default – не обязательная конструкция.

Если будет выполнятся какой-либо оператор, то будут выполнятся операторы всех последующих вариантов (case и default) до тех пор, пока не встретится оператор **break**.

Область действия меток case – тело оператора switch. Никакие две константы не могут иметь одинакового значения.

Константы могут иметь только типы int, char, enum. Объект сравнения с меткой может быть константой, переменной, указателем функции, выражением упомянутых типов.

Меток может быть несколько, например:

case '1': case '3': case '5': оператор;

case '2': case '4': case '6': оператор;

Пример 3.7:

Использования оператора switch.

```
# include <math.h> // Подключение математической библиотеки
# include <stdio.h>
void main (void)
{
float x;
puts("Введите значение x: ");
scanf ("%f", &x);
puts ("Введите первую букву имени функции: \n"
"S- Sin \n"
"C-Cos \n"
"A-Atan ");
```

```
flushall () ;
switch (getchar( ) )
{
case 'S' : case's' :
printf ("Синус X= %f \n", sin(x) ) ; break;
case 'C': case 'c':
printf ("Косинус X= %f \n", cos(x) ); break;
case 'A': case'a':
printf("Арктангенс X= %f \n", atan(x)); break;
default:
puts ("Ошибка \a\n");
}
}
```

Замечание: Одна из распространенных ошибок состоит в том, что часто забывают разделять альтернативные операторы в теле переключателя операторами break.

3.6. Оператор разрыва break

Оператор разрыва *break* прерывает выполнение операторов for, while, do...while и switch. Он может присутствовать только в теле этих операторов. Появление этого оператора в другом месте программы рассматривается как ошибка.

Синтаксис:

break;

Действие:

Если оператор разрыва находится внутри вложенных операторов, то прерывается только непосредственно охватывающий его оператор. Если же требуется прерывание более чем одного уровня вложенности, то следует использовать операторы возврата return и перехода go to.

Замечание: С помощью оператора go to нельзя передать управление на конструкции case и default в теле переключателя.

Пример 3.8.

Использование оператора break.

```
# include <stdio.h>
// Применение ключа switch с использованием оператора break.
main( )
{
```

```
char ch;
printf("Введите заглавную букву русского алфавита");
ch = getchar();
if (ch >= 'А' && ch <= 'Я')
switch(ch)
{
case 'А':
printf("Алексеев \n");
case 'Б':
printf("Булгаков \n");
case 'В':
printf(" Волошин \n");
break;
case 'Г':
printf("Гоголь \n");
break;
default:
printf("Достоевский, Зощенко и др. \n);
break;
}
else
printf("Надо было ввести заглавную букву \n");
}
```

3.7. Операция условия ?:

Операция *условие* – единственная операция языка Си, имеющая три операнда.

Синтаксическая форма:

Выражение_1? выражение_2: выражение_3;

Правило выполнения:

Если выражение_1 истинно (т.е. ! = 0), то результатом операции является значение выражения_2, в противном случае – значение выражения_3.

Примеры использования:

1. Нахождения max из двух чисел:

```
max = (a>b)? a: b;
```

2. Нахождение абсолютной величины числа x:

```
abs = (x >0)? x: –x;
```

3.8. Препроцессор языка Си и директивы условной компиляции

Препроцессор языка Си – это программа, выполняющая обработку входных данных для другой программы. Препроцессор языка Си просматривает программу до компилятора, заменяет аббревиатуры в тексте программы на соответствующие директивы, отыскивает и подключает необходимые файлы, может влиять на условия компиляции. Директивы препроцессора не являются в действительности частью языка Си. Препроцессор включает в себя следующие директивы (табл. 3.1)

Таблица 3.1

ДИРЕКТИВЫ ПРЕПРОЦЕССОРА ЯЗЫКА Си

Определение	Назначение
#define	Определение макроса
#undef	Отмена определения макроса
#include	Включение объекта-заголовка
#if	Компиляция, если выражение истинно
#ifdef	Компиляция, если макрос определен
#ifndef	Компиляция, если макрос не определен
#else	Компиляция, если выражение в if ложно
#elif	Составная директива else/if
#endif	Окончание группы компиляции по условию
#line	Замена новым именем строки или имени исходного файла
#error	Формирование ошибок трансляции
#pragma	Действие определяется реализацией
#	Null- директива

Как видно из табл.3.1, все директивы начинаются с символа #.

Директива #define вводит макроопределение или макрос. Общая форма директивы следующая:

 # define ИМЯ_МАКРОСА последовательность_символов

Последовательность символов называют еще строкой замещения. Когда препроцессор находит в исходном файле имя_макроса (просто макрос), он заменяет его на последовательность_символов.

Можно отменить определение макроса директивой # undef:

 # undef имя_макроса

Данная строка удаляет любую ранее введенную строку замещения. Определение макроса теряется и имя_макроса становится неопределенным.

К примеру можно определить MAX как величину 100:

#define MAX 100

Это значение будет подставляться каждый раз вместо макроса MAX в исходном файле, Можно также использовать макрос вместо строковой константы:

#define NAME " Turbo C++"

Если последовательность символов в директиве не помещается в одной строке, то можно поставить в конце строки \ и продолжить последовательность на другой строке. Среди программистов принято соглашение, что для имен макросов используются прописные буквы, так как их легко находить в программе. Также все директивы #define лучше помещать в начало программы.

Директива *#define* имеет еще одну важную особенность: макрос может иметь аргументы. Каждый раз, когда происходит замена, аргументы заменяются на те, которые встречаются в программе.

Пример: #define MIN(a, b) ((9a)<(b)) ? (a) : (b)

.....

printf("Минимум из x и y % d", MIN(x ,y));

printf("Минимум из a и b % d", MIN(n ,m));

.....

Когда программа будет компилироваться, в выражение, определенное MIN (a, b) будут подставлены соответственно x и y или m и n. Аргументы a и b заключены в круглые скобки, так как вместо них может подставляться некоторое выражение, а не просто идентификатор.

Например, printf("Минимум % d", MIN(x*x, x));

Директива # error имеет вид:

 # error сообщение_об_ошибке

Эта программа прекращает компиляцию программы и выдает сообщение об ошибке.

Директивы условной компиляции. К данным директивам относятся:

#if, #else, #elif, #endif.

Данные директивы производят выборочную компиляцию программы. Если выражение, следующее за #if, истинно, то коды, заключенные между #if и #endif, будут компилироваться. В противном случае они при компиляции будут пропущены. Выражение, следующее за #if, проверяется во время компиляции, поэтому оно может содержать только константы и макросы, которые прежде определены. Переменные здесь не могут использоваться.

Директива *#else* используется так же, как и else в языке Си.

Пример 3.9:
Использование условной компиляции.

```
# include <stdio.h>
# define MAX 100
main(void)
{
# if MAX>99
printf(" MAX равно %d \n", MAX);
# endif
return 0;
}
```

Директива #elif используется для организации вложенной условной компиляции. Форма использования ее следующая:

```
#if <выражение>
последовательность операторов
#elif <выражение 1>
последовательность операторов
#elif <выражение 2>
последовательность операторов
.....
# endif
```

Другой метод условной компиляции состоит в использовании директив *#ifdef* и *#ifndef* Основная форма использования этих директив следующая:

#ifdef ИМЯ_МАКРОСА
последовательность операторов
 # endif
и соответственно

#ifndef ИМЯ_МАКРОСА
последовательность операторов
 # endif

Если макрос определен, то при использовании # ifdef компилируется соответствующая последовательность до операторов # endif. Если же макрос не определен или был отменен директивой #undef, то соответствующая последовательность операторов игнорируется компилятором.

Директива #ifndef действует противоположным образом. О других директивах более подробно можно прочитать в [2].

4. СЛОЖНЫЕ ТИПЫ ДАННЫХ

4.1. Объявление и инициализация массивов

Ранее мы ввели типы данных, которые называются базовыми или встроенными. На основе этих типов язык Си позволяет строить другие более сложные типы данных и структуры данных.

Массивом называется набор данных одного и того же типа, собранных под одним именем. Каждый элемент массива определяется именем массива и порядковым номером элемента, который называется индексом. Индекс всегда является целым числом.

Основная форма объявления массива в программе:

Тип < имя массива > [размер 1] [размер 2]....[размер n];

Чаще всего используются одномерные массивы:

Тип <имя массива> [размер];

Тип – базовый тип элементов (int, float, char).

Размер – количество элементов одномерного массива.

В языке Си индекс всегда начинается с 0. Первый элемент – массива всегда имеет индекс 0. Например, если мы объявили массив int a [100], это значит массив содержит 100 элементов – от a [0] до a [99]. Для одномерного массива легко подсчитать, сколько байт в памяти будет занимать этот массив.

N Килобайт = < размер базового типа > * < количество элементов >.

Язык Си допускает двумерные массивы. Их можно назвать как: массив одномерных массивов. Двумерный массив int a[3][4] можно представить в виде таблички:

	Номер столбца – второй индекс (j)			
Номер строки – первый индекс (i)	A [0] [0]	A [0] [1]	A [0] [2]	A [0] [3]
	A [1] [0]	A [1] [1]	A [1] [2]	A [1] [3]
	A [2] [0]	A [2] [1]	A [2] [2]	A [2] [3]

В памяти ЭВМ массив располагается непрерывно по строкам, т. е. a [0] [0], a [0] [1], ... a [0] [3], a [1] [0], a [1] [1], ...a [2] [3].

Следует помнить, что память для всех массивов, которые определены как глобальные, отводится во время компиляции, и сохраняется всё время, пока работает программа. Часто двумерные массивы используются для работы с таблицами, содержащими текст.

T	U	R	B	O		B	A	S	I	C	\0					
T	U	R	B	O		C	+	+	\0							

Инициализацию массивов можно производить разными способами.

Первый способ.

float arr [6] = {1.1, 2.2, 3.3, 4.0, 5.0, 6}; // одномерный массив

int a [3] [5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

Второй способ.

a[0] [0] = 1; a [0] [1] = 2; a [0] [2] = 3; a [0] [3] = 4

a [0] [4] = 5; a [0] [5] = 6; a [0] [6] = 7; a [0] [7] = 8 и т. д.

Многомерные массивы, в том числе и двумерные, можно инициализировать, рассматривая их как массив массивов.

Инициализации:

int a [3] [5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}; и

int a [3] [5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};

эквиваленты. Количество инициализаторов не обязательно должно совпадать с количеством элементов массива. Если инициализаторов меньше, то оставшиеся элементы массива не определены.

В тоже время инициализации:

int a [3] [5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 11}; и

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

различны.

Соответствующие массивы будут заполнены следующим образом:

В первом случае:

1	2	3	4	5
6	7	8	9	10
11				

Во втором случае:

1	2	3		
4	5	6	7	8
9	10	11		

В пустых клетках значения не определяются.

Символьные массивы могут инициализироваться как обычные массивы

char str [15] = {'B', 'o', 'r', 'l', 'a', 'n', 'd', ' ', 'C', '+', '+'};
а могут как строка:

char str [15] = "Borland C + +";

Можно указывать массив без указания размера:

int mas [ ] = {1, 2, 3, 4, 5, 1, 2};

char str [ ] = "Это объявление и инициализация символов" ;

компилятор сам определит количество элементов массива. Символьные массивы в языке Си занимают особое место. Во многих языках есть специальный тип – строка символов. В языке Си этого нет, и работа со строками реализована как работа с одномерным массивом.

Строка – это одномерный массив типа *char,* заканчивающийся *нулевым байтом. Нулевой байт* – это байт, каждый бит которого равен нулю.

Например: char str [11];

Предполагается, что строка содержит 10 символов, а последний байт зарезервирован под нулевой байт.

Есть два способа ввести строку с клавиатуры:

1) С помощью функции scanf() со спецификатором %s. Она вводит символы до первого пробельного символа.

2) С помощью библиотечной функции gefs(), которая вводит строки с пробелами.

При объявлении многомерных массивов с неизвестным количеством элементов можно не указывать размер только в самых левых квадратных скобках, например:

```
int arr[][3] = {1, 2, 3,
5, 6, 7,
8, 9, 0};
```

Пример 4.1.:

Все числа из каждой строки массива a умножить на числа соответствующего столбца массива b и результат сложения записать в массив.

```
/ * Перемножение матриц */
# include <stdio.h>
# include <conio.h>
void main (void)
{
```

```
/* Инициализация двумерных массивов */
int a [3] [4] = {
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12}
};
int b [4] [2] = {
{1, 2},
{3, 4},
{5, 6},
{7, 8}
};
int c [3][2], i, j, k; // массив результатов и параметры цикла
clrscr();
for ( i = 0; i = 3; i + +) // проход по вертикали (строкам)
{
for ( j = 0; j < 3; j + +) // проход по горизонтали (рядам)
{
c [i] [j] = 0; // обнуление
for (k = 0; k<4; k + +)
c [i] [j] + = a [i] [k] * b [k] [j];
}
}
for (j = 0; j < 3; j + +)
{
for (j = 0; j < 2; j + +)
printf ("%6d", c[i][j];  printf ("\ n");
}
}
// Результат прогона: Ручное вычисление
// 50 60      c [0, 0] = 1*1 + 2*3 + 3*5 + 4*7 = 50
// 114 140    c [0, 1] = 1*2 + 2*4 + 3*6 + 4*8 = 60
// 178 220    c [1, 0] = 5*1 + 6*3 + 7*5 + 8*7 =114
              c [1, 1] = 5*2 + 6*4 + 7*6 + 8*8 =140
              c [2, 0] = 9*1 + 10*3 + 11*2 + 12*7 = 178
              c [2, 1] = 9*2 + 10*4 + 11*6 + 12*8 = 220
```

4.2. Указатели

Понимание и правильное использование указателей очень важно для создания очень хороших программ на языке Си. Указатели необходимы для успешного использования функций и распределения памяти. Кроме того, многие конструкции языка Borland C + + требуют применения указателей. Однако с указателями следует обращаться очень осторожно. При использовании неинициализированного указателя может быть зависание компьютера и ошибки.

Объявление указателей. Указатель – это переменная, которая содержит адрес объекта, т.е. адрес ячейки памяти. Вообще то это просто целое число. Но нельзя трактовать указатель как переменную или константу целого типа. Если переменная будет указателем, то она должна быть соответствующим образом объявлена.

Указатель объявляется следующим образом:

*тип *<имя переменной>;*

Например:

```
char *ch ;
int * temp, i, *j ;
float * pf, f ;
```

Здесь объявлены указатели ch, temp, j, pf, переменная i – целого типа и переменная f типа float.

Операции над указателями. С указателями связаны две специальные операции & и *. Обе эти операции являются унарными, т. е. они имеют один операнд, перед которыми они ставятся.

Операция & соответствует операции «взять адрес».

Операция * соответствует словам «значение, расположенное по указанному адресу».

Особенность языка Си состоит в том, что знак * соответствует двум операциям, не имеющим друг к другу никакого отношения: арифметическое умножение и операции «взять значение». В тоже время спутать их в контексте нельзя, т.к. одна операция унарная содержит один операнд, а вторая – умножение, бинарная (содержит два операнда). Унарные операции & и * имеют наивысший приоритет, наравне с унарным минусом.

В указателе очень важным является базовый тип, т.к. он определяет, сколько байтов занимает переменная указатель. Если int – 2

байта, char – один байт и т.д. Простейшие действия с указателями продемонстрируем на следующей программе.

Пример 4.2.

```
# include <stdio.h>
main ()
{
float x = 10.1, y ;
float * pf ;
pf = &x ;
y = *pf ;
printf("x = %f y= %f", x, y); // Результат: x=10.1; y=10.1; pf=FFF6
*pf ++;
printf ("x = %f y = %f", x, y); // Результат: x=10.1; y=10.1 ; pf=FFF2
y =1+ *pf * y;
printf ("x = %f y = %f", x, y); // Результат: x=10.1; y=1 ; pf=FFF6
return 0;
}
```

К указателям можно применить операцию присваивания. Указатели одного и того же типа могут использоваться в операции присваивания, как любые другие переменные.

Пример 4.3.

```
# include <stdio.h>
main ()
{
int x = 0 ;
int *p, *g ;
p = &x ;
g = p ;
printf("%p", p); /* печать содержимого p */
printf("%p", g ); /* печать содержимого g */
printf("%d %d", x., *g); /*печать величины x и величины по адресу g */
}
```

В этом примере приведена еще одна спецификация формата функции printf() %p – печать адреса памяти в шестнадцатеричной системе счисления.

В языке Си указателю допустимо присвоить любой адрес памяти. Однако, если объявлен указатель на целое int *pi; а по адресу, который присвоен данному указателю, находится переменная типа float, то при компиляции будет выдано сообщение об ошибке в строке

p = &x;

Эту ошибку можно исправить, преобразовав указатель на int к типу указателя на float явным преобразованием типа:

p = (int*)&x;

но при этом теряется информация о том, на какой тип указывал исходный указатель.

Пример неправильной программы:

```
main ()
{
float x =10.1, y;
int *p;
p = &x /* потом заменим на p=(int*)&x */
y = *p;
printf ("x = %f y = &f \ n", x, y);
}
```

В результате работы этой программы не будет получен тот ответ, который ожидался. Переменной y не будет присвоено значение переменной x, т.к. будут обрабатываться не 4 байта, как положено для переменной типа float, а только 2 байта, т.к. базовый тип указателя – int.

Как и над другими типами переменных, над указателями можно производить арифметические операции сложения и вычитания, а также операции (++) и (—). Указатели можно сравнивать. Применимы шесть операций:

<, >, < =, > =, =, = =, ! =

Пример 4.4

```
main()
{
int *p;
```

```
int x;
p = &x;
printf ("%p%p", p, ++p);
printf ("%p", ++p);
}
```

После выполнения этой программы, мы увидим, что при операции ++p значение указателя изменится не на 1, а на 2. И это правильно, т.к. значение указателя должно указывать не на следующий адрес памяти, а на адрес следующего целого. А целое, как мы помним, занимает 2 байта. Если бы базовой тип указателя был не int, а double, то были бы напечатаны адреса, отличающиеся на 8. Именно столько байт занимает переменная типа double.

К указателям можно прибавлять некоторое целое или вычитать. Пусть указатель p имеет значение 2000 и указывает на целое, тогда в результате выполнения оператора p=p+3 его значение будет 2006. Если указатель p1=2000 был бы указателем на float, то после применения оператора p1= p1+10 значение p1 было бы 2040.

Общая формула для вычисления указателя по формуле p= p+ n будет иметь вид:
<p>=<p>+n*<количество байт памяти базового типа указателя>.

Сравнение p < g указывает, что адрес, находящийся в p, меньше адреса, находящегося в g.

4.3. Массивы и указатели на языке Си

Массивы описывают регулярную структуру данных одного типа.

Одномерные массивы:

```
int temp [365];
char arr [10];
char *point[10];
```

Двумерные массивы:

```
int array[4] [10];
char arr [3] [7];
```

Число в [ ] указывает количество элементов массива, поэтому:

temp [365] – массив из 365 элементов.

Доступ к каждому элементу осуществляется по его индексу (номеру), т.е. temp[0], temp[1],…,temp[364]– последний элемент. Элементы массива нумеруются начиная с 0.

Можно также использовать многомерные массивы, например: int arr [k] [l] ...[n];

Однако следует помнить, что для хранения элементов таких массивов требуется значительный объем памяти.

Рассмотрим, как происходит размещение элементов массива в памяти ЭВМ. Как правило, элементы массива занимают последовательные ячейки памяти. При этом элементы размещаются таким образом, что самый последний индекс возрастает быстрее. Это в случае двумерного массива означает, что он будет записываться построчно: строка за строкой. Поскольку указатели указывают адрес ячейки, то между массивами и указателями существует тесная связь. Вспомним, что имя массива – это указатель на его первый элемент. По существу массив можно рассматривать как индексированный указатель. Доступ к элементам массива осуществляется по номеру индекса. При этом приращение индекса на единицу вызывает перемещение указателя на число байт, соответствующее объекту данного типа: для целых чисел на 2 байта, для действительных – на 4 байта и т.д.

Объявления int mas[] и int *mas идентичны по действию: оба объявляют mas указателем.

Индекс массива действует аналогично стрелки часов, показывающей по очереди на каждый следующий элемент массива.

Пример 4.5:

```
int mas[10];
int *ptr;
ptr = mas; // присваивает адрес указателю
// следующие операции дадут один и тот же результат:
mas[2] = 20;
*(ptr + 2) = 20;
// следующая операция прибавит 2 к первому элементу:
*ptr + 2;
```

Указатели и многомерные массивы. Рассмотрим двумерный массив и действия с указателями.

```
int mas[4][2];
int *ptr;
ptr = mas;
```

ptr сейчас указывает на первый столбец первой строки, т.е.

ptr = = mas = = &mas [0] [0];

Увеличим указатель:

ptr+1 = = &mas [0] [1];

ptr+2 = = &mas [1] [0];

ptr+3 = = &mas [1] [1] и т.д.

Двумерный массив можно представить как массив массивов. В нашем случае мы имеем четырех элементный массив, состоящий из двух элементов. Примечательно, что этот четырех элементный массив можно представить в виде одномерного mas[0],..., mas[3]. При этом имя массива по-прежнему является указателем на его первый элемент, т.е. mas[0]= =&mas[0] [0] . На что же будут указывать mas[i]? В этом случае mas [i] указывает на i-тую строку, т.е. на первый элемент i - й строки. Таким образом

mas [0] == &mas [0] [0];

mas [1] == &mas [1] [0];

mas[2] == &mas [2] [0];

mas[3] == &mas [3] [0];

Массивы и указатели – различия.

Имя массива – является указателем – константой. Описания:

char string_1[20]= «Язык Си» и

char *string_2 = «Язык Си»;

размещают в памяти соответствующую строку символов. Различие состоит в том, что указатель string_1 является константой, а указатель string_2 – переменной. Это различие проявляется в случае использования операции единичного приращения ++. Эту операцию можно применять только к переменным. Поэтому string_2++ – допустимая конструкция, а string_1++ – запрещенная. Однако и в том и в другом случае можно использовать операции сложения с указателем, т.е.

string_1 + i ;

string_2 + i ; – допустимые конструкции.

При задании массива символов можно указывать размер явно, например:

char mas_1 [10] = “Яблоко”;

Или определить массив по умолчанию

Char mas_2 [ ] = “Груша”.

Отличие заключается в том, что во втором случае будет выделено ровно столько памяти, сколько необходимо.

Массивы и указатели символьных строк

Часто бывает необходимо иметь массив символьных строк. При этом возможно задать два типа описаний:

1) char string_1[10][20]; //10 строк по 20 символов

2) char* string_2[10]; // массив из 10 указателей на строки символов. Какие здесь различия? Различие заключается в том, что в первом случае задается "прямоугольный" массив, в котором каждая строка имеет фиксированную длину, а во втором определяется "рваный" массив, где длина каждой строки занимает столько байт , сколько необходимо.

Например:

char string_1[3] [7] ={"Яблоко", "Слива" , "Груша"};

char *string_[3]={"яблоко", "слива", "груша"};

Заполнение массивов будет следуюим:

string_1: string_2:

Яблоко\0 яблоко\0

Слива\0\0 слива\0

Груша\0\0 груша\0

5. ФУНКЦИИ В ЯЗЫКЕ Си

5.1. Типовая структура программы на языке Си

В отличие от других языков программирования высокого уровня в языке Си нет деления на процедуры, подпрограммы и функции. В Си программа строится только из функций.

Функция – это независимая совокупность объявлений и операторов, предназначенная для выполнения определенной задачи. Каждая функция должна иметь имя, которое используется для вызова функции. Имя главной функции main(), которая должна присутствовать в каждой программе, зарезервировано. Функция main() не обязательно должна быть первой, однако с нее всегда начинается выполнение Си-программы. Приведем типовой пример записи программы

```
# include < > /* директивы препроцессора */
# define NAME значение // макроопределение или макрос
/* объявление функций */
int func_1 (список параметров) ;
void func_2 (список параметров);
...
float func_n (список параметров);

/* описание глобальных переменных */
float a,b,c char ch;

/* определение функций */
int func_1 (список параметров)
{
int i,j ; /*описание локальных переменных */
тело функции;
}
void func_2 (список параметров)
{
тело функции;
}
float func_3 (список параметров)
```

```
{
тело функции;
}
void main(void)
{
описание главной функции
}
```

С пользованием функций в языке Си связаны три понятия: объявление функции, определение функции и вызов функции.

Объявление или прототип функции задает ее имя, типы и число формальных параметров, тип значения, возвращаемого функцией, и класс памяти. В объявлении формальные параметры могут иметь имена, однако это не является необходимым. Объявление оканчивается символом ';' (точка с запятой).

Примеры объявлений:

int func_1(char, int, float); //объявление о функции func_1()*

char func_2(int a, int b, char* ch); //объявление о функции func_2()*

Из примеров видно, что сначала указывается тип значения, возвращаемого функцией (по умолчанию тип int), затем следует имя функции, после чего в круглых скобках указываются типы формальных параметров, разделяемых запятыми. В том случае, если формальные параметры отсутствуют, либо функция не возвращает никакого значения, используется служебное слово void:

```
void func_3(int, int); //отсутствует возвращаемое значение
void func_4(void); // отсутствуют формальные параметры и возвращаемое значение
```

Имена функций должны быть уникальными.

Список формальных параметров может оканчиваться запятой и многоточием (,...):

```
int func_5(int* ,int,...);
```

Это означает, что число аргументов функций переменно, но не менее числа аргументов, заданных перед последней запятой. Контроль типов не указанных переменных возлагается на программиста.

Определение функции задает ее заголовок, объявления локальных объектов (констант, переменных) и операторы, которые определяют действие функции. Тело функции заключается в фигурные скобки.

Заголовок функции синтаксически имеет такой же формат, как и прототип функции, с той лишь разницей, что, кроме типов, указываются имена формальных параметров и отсутствует символ ';' в конце заголовка.

Пример определения функции:

```
char* func(int param_1, int param_2, char* param_3)
//тело функции
{
int i, j, k; //объявления локальных объектов
char buff [80];
...
оператор;
оператор;
...
return buff; //возвращаемое значение
}
```

Вызов функции определяет действия, выполняемые функцией. При вызове функции ей могут быть переданы данные посредством аргументов функции, называемых фактическими параметрами. Если функция возвращает значение, то это и есть основной результат выполнения функции, который при выполнении программы подставляется в точку ее вызова. Если функция не возвращает никакого значения ее действие может состоять из выполнения операций, не связанных с обработкой данных.

Существует два способа вызова функции:

1. имя_функции (список фактических параметров);

2. (* указатель_на_функцию) (список фактических параметров), здесь круглые скобки обязательны.

Указатель_на_функцию – это переменная, содержащая адрес функции, Адрес функции может быть присвоен указателю оператором:

указатель_на_функцию = имя_функции ;

Примеры вызовов:

```
func (a,b ) ;
func (&a, &b ) ;
func () ;
( *func) (&a, &b ) ; и т.д.
```

Вызов функции может являться частью выражения, стоящего справа от операции присваивания, например:

Y = func_1 (a, b) + func_2 (c, d) ;

Операция присваивания «сохраняет» значение, возвращаемое функцией. Если же просто написать func_1 (a, b), то значение не будет сохранено.

Операция func_1 (a, b) правомерна, если функция не возвращает значения, т.е. имеет тип void.

Функция, не имеющая параметров и не возвращающая значения должна быть описана как void func(void); ее вызов будет иметь вид func ();

5.2. Оператор return

Этот оператор имеет два варианта применения:

1. Если некоторая функция возвращает некоторый результат (значение) в точку вызова, то в тексте функции должен обязательно быть оператор вида

return (выражение);

(круглые скобки здесь не обязательны).

Пример 5.1: Функция нахождения большего из двух чисел.

```
int max (int a, int b)
{
if (a > b)
return (a);
else
return (b);
}
```

Можно еще короче:

```
int max (int a, int b)
{
if (a > b) return (a);
return (b);
}
```

А можно использовать оператор “?”:

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

Если функция не возвращает никакого значения, она может также заканчиваться оператором return.

Отсутствие этого оператора не вызывает никаких неприятностей, т.к. при достижения закрывающей скобки '}' компилятор автоматически генерирует код возврата.

2. Если требуется возврат в точку вызова до исполнения всех операторов тела функции в результате выполнения какого – либо условия. Например,

```
{
…
if (ch = ='\0')
return 0;
…
}
```

5.3. Передача параметров в функцию

Часто требуется, чтобы одна функция изменяла значения переменных, относящихся к другой функции. Предположим, что у нас имеются две переменные *x* и *y*, и мы хотим, чтобы они обменялись своими значениями. Схематически такой обмен представлен на рис 5.1.

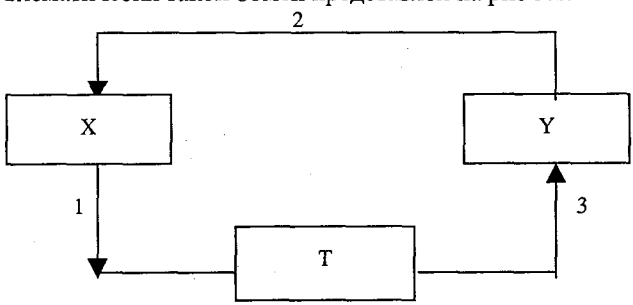


Рис.5.1

Для обмена двумя значениями вводится временная переменная temp, согласованная по типу с переменными *x* и *y*.

Последовательность операций здесь следующая:

```
temp = x;
x = y;
y = temp;
```

Рассмотрим программу, реализующую обмен конкретными значениями.

Пример 5.2.

```
//Программа обмен
#include <stdio.h>
void obmen( int, int); // Прототип функции
void main ( void)
{
int x = 5, int y = 20;
printf (" x = %d y = %d", x, y);
obmen ( x, y); // Вызов функции
printf ("После обмена : x = %d y = %d", x, y);
}
// Описание функци
void obmen( int a, int b);
{
int temp;
temp = a;
a = b;
b = temp;
}
```

На входе программы *a* и *b* получили значения *x* и *y*, но на выход результат не передавался и значения внешних переменных *x* и *y* не изменились.

Результат работы программы будет выглядеть следующим образом:

x = 5 y = 20.

После обмена значениями: x = 5 y = 20, т.е. значения переменных не поменялись местами. Это произошло потому, что по правилам языка Си параметры функции являются параметрами – значениями. Это означает, что в момент обращения к функции, ей в качестве фактических параметров передаются не сами переменные, а их копии в виде значений, которые они имеют в данный момент.

Изменение значений локальных переменных *a* и *b* никак не сказывается на значениях *x* и *y*. Это совершенно различные переменные.

Каким же образом все – таки поменять значения переменных x и y в программе? Для этого следует использовать указатели. Почему? Потому что указатель – это адрес переменной, следовательно, используя указатель можно осуществлять доступ к самой переменной. Значение указателя и в вызывающей программе и в функции, которой это значение передано, указывает на одну и ту же переменную.

Как будет выглядеть правильная программа? Запишем ее.

Пример 5.3.

```
# include <stdio.h>
void obmen (int*, int*);
void main(void)
{
int x = 5, y = 20;
printf (…);
obmen (&x, &y); //адрес x, y = указателю на a и b.
printf (" x = %d y = %d", x, y);
}
void obmen (int*a, int*b)
{
int temp;
temp = *a; //разадресация, т.е. берется значение по адресу
*a = *b;
*b = temp;
}
```

Теперь программа будет работать правильно.

Декларация void obmen (int*, int*)

говорит о том, что в качестве аргументов в функцию obmen () следует передать указатели (адреса). Вот почему при вызове функции в качестве фактических параметров используют &x и &y, т.е.:

obmen (&x, &y).

5.4. Ссылочные переменные

Как мы видели в примере 5.3, при передаче в функцию указателей в теле функции необходимо использовать явно операцию разадресации (* адрес объекта). Это усложняет синтаксис. Чтобы исключить этот недостаток в языке Си были введены ссылки. Ссылка

– это другое имя переменной (объекта). Синтаксис объявления ссылочной переменной имеет следующий вид:

Тип &имя ссылки=имя объекта;

Например:

```
int x = 20;
float y;
int &n = x;
float &m =y;
```

После таких объявлений ссылочные переменные n и m будут определять местоположение в памяти переменных x и y, т.е., если ссылке присваивается значение, то и переменная получит тоже значение.

Из примеров видно, что при объявлении ссылочной переменной она обязательно должна быть проинициализирована, так как в противном случае неясно на какой объект будет производиться ссылка.. При обращении к ссылочным переменным нет необходимости в операции снятия ссылки, т. е. переменные n и m будут обрабатываться как "нормальные" переменные типов int и float.

Запишем теперь программу, изменяющую значения переменных с использованием ссылочных переменных:

Пример: 5.4

```
# include<stdio.h>
void obmen(int &, int &);
void main (void)
{
int x = 5, y = 20;
printf("x = %d y = %d \n", x, y);
obmen(x, y);
printf("x = %d y= %d \n", x, y);
void obmen(int &a, int &b);
{
int temp;
temp = a;
a = b;
b = temp;
}}
```

Из примера 5.4 видно, что синтаксис при использовании ссылок более удобен.

5.5. Рекурсивные вызовы функций

Любая функция может быть вызвана рекурсивно, т. е. она может вызвать саму себя. Классический пример рекурсии – это вычисление факториала числа n! =1*2*3…*n. Пример рекурсивной функции, вычисляющей значение факториала для n> 0:

```
fact (int n) // Название функции
{
int a;
if (n= = 1) return 1;
a = fact (n – 1)*n;
return a;
}
```

5.6. Массивы и функции

Массивы могут использоваться в качестве формальных аргументов функции. В случае одномерных массивов указывается тип, имя и размер массива – параметра. При использовании в качестве формального параметра двумерного массива следует указать его размер по второму индексу, например,

int func(int mas[] [N], int n, int m);

При таком описании функции компилятору сообщается на сколько строк следует разбить передаваемый массив – параметр.

Вызов такой функции должен иметь следующий вид:

var = func(mas, n, m);

т.е. фактическим аргументом при вызове функции является указатель на первый элемент массива.

Приведем пример программы, в которой используется передача в функцию двумерного массива.

Пример:5.5

```
/* Программа работы с матрицей */
#include<stdio.h>
#include<conio.h>
#define N 5 // Задание размера матрицы в виде макроса
void input (int mas[] [N], int, int); /* функция ввода матрицы */
```

```
int making (int mas[] [N], int, int); /* функция определения максимального элемента */
void main (void)
{
int m, n ;
int a[N][N] ;
int max ;
clrscr() ;
printf (" Введите размер исходной матрицы \n ");
printf (" Число строк = ");
scanf("%d", &m);
printf (" Число столбцов = ");
scanf ("%d", &n);
input(a, m, n); // Выэов функции
max = making(a, m, n);
printf("Значение максимального элемента матрицы равно %d \n", max);
getch();
}
/* Функция осуществляет ввод матрицы заданного размера */
void input (int a[] [N], int m, int n)
{
int i, j;
for (i = 0;i < m; i++)
for (j = 0; j <n; j++)
{
printf (" Введите A(%d,%d) элемент матрицы : ", i+1, j+1);
scanf ("%d", &a[i] [j]);
}
}
/* Функция вычисляет максимальный элемент матрицы заданного размера */
int making (int a[] [N], int m, int n)
{
int max = a[0] [0] ;
int i, j ;
for (i=0 ; i<m; i++)
for (j=0 ; j<n ; j++)
```

```
if (max <a[i] [j] )
max = a[i] [j] ;
return max ;
}
```

В случае использования одномерного массива заголовки функций будут иметь следующий вид:

```
void input (int mas[N] , int) ;
int making(int mas[N] , int) ;
```

а в текст программы следует ввести соответствующие изменения.

6. ТИПЫ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

Кроме известных нам типов данных язык Си позволяет создавать еще пять типов данных: структуры (structure), объединения (union), перечислимый тип (enumeration), поля битов (bitfields). Пятая возможность – с помощью оператора typedef создавать новое имя (псевдоним) для уже существующего типа.

6.1. Структура в языке Си

Структура – это совокупность взаимосвязанных элементов одного, либо разных типов. Взаимосвязь определяет порядок размещения элементов структуры. Элементы структуры объединены под одним именем. Поэтому структура трактуется не как множество отдельных элементов, а как единое целое.

Примеры структур: библиографические данные о книге; строка платежной ведомости; данные о некотором товаре; координаты точек на плоскости и т. п.;

Над структурами в языке Си стандартом ANSI определены следующие операции:

1. операции присваивания между структурами, имеющими одно и то же имя типа;
2. операции копирования;
3. передача структур в качестве параметров функциям и возврат их в качестве результата;

Кроме этих операций, к структурным переменным применимы следующие:

4. адресная операция &; (кроме битовых полей);
5. операции доступа к элементам структуры " ." и –> ;
6. инициализация структурных переменных, имеющих класс auto.

В то же время к структурам нельзя применять операции отношения (т.е. нельзя сравнивать структуры).

Элементами структуры могут быть переменные базового типа, массивы, указатели, объединения, другие структуры. Однако элементом структуры не может быть структура того же типа, в которой он содержится. В то же время этот элемент может быть указателем на тип структуры, в которую он входит. Это позволяет создавать связанные списки структур.

Структура задается в программе структурным шаблоном вида:

```
struct book
{
char title [40];
char author [30];
float price;
}
```

Этот шаблон описывает структуру из двух символьных массивов и одной переменной типа float.

Имя book – это имя типа структуры, на которое впоследствии можно ссылаться, как и на имя стандартного типа.

Шаблон только описывает структуру, не вызывая никаких действий компилятора. Для того, чтобы компилятор выделил память для структуры, необходимо описать саму структурную переменную. Это описание может быть задано следующим образом:

```
struct book libr ;
```

где struct book – имя_типа, libr – имя_переменной.

Можно объединить определение структурного шаблона и описание структурной переменной, например:

```
struct book      struct
{ {
char title [];   float x ;
char author [];  float y ;
float price;     float z:
} libr ;         } coord ;
```

Можно также проинициализировать структурную переменную. Имя типа структуры можно опустить, т. е.

```
struct
{
```

.....
.....
.....
} libr ;

Это делается, когда используется одна структурная переменная. Список элементов, заключаемый в { } называется полями структуры. Доступ к отдельному полю осуществляется с помощью следующей записи:

libr . title
/ \
имя переменной имя поля

Указатель на структурную переменную может быть задан как:

*struct book *ps ;*

Доступ к элементу (полю) структуры, описанной через указатель, осуществляется с помощью следующей записи:

ps—>title;
или ps—>price.

Последняя запись эквивалента (*ps). price.

Создание связанного списка структур осуществляется с помощью следующей конструкции:

struct book
{
char title [ ];
char author [ ];
float price;
*struct book *next*; /*указатель на следующую строку структуры */
} libr;

Пример 6.1:

/* Передача в функцию и возврат целой структурной переменной */

```
# include <stdio. h>
# include <string. h>
struct book
 { char title [20];
 char author [15];
```

```
float price;
}; /* внешний шаблон виден всем */
struct book func (struct book); /* прототип функции */
   void main (void)
{
struct book lidr_1={"Киселев А.",
"Избранное",
4.25}, libr_2;
libr_2=func (libr_1);
    printf ("%–20s %–20s цена –%5.2f \ n",
libr_1. title, libr_1. author, libr_1. price);
   printf ("%–20s %–20s цена –%5.2 f \ n",
libr_2. title, libr_2. author, libr_2. price);
}
struct book func ( struct book libr)
    { strcpy (libr. title, "Язык Си");
    strcpy (libr. author, "Семенов К.");
libr. price = 6.36;
return libr; }
```

Переменная типа структуры может быть глобальной, локальной и формальным параметром. Можно, естественно, использовать структуру или ее элемент как любую другую переменную в качестве параметра функции. Например,

```
func1(first. a) ; func2 (&second. b);
```

Заметим, что & ставится перед именем структуры, а не перед именем поля.

Можно в качестве формального параметра передать по значению всю структуру:

Пример 6.2:

```
/* Использование структуры в качестве параметра */
include<stdio.h>
struct stru {
 int x ;
 char y ;};
void f(struct stru param ); /* прототип функции */
main (void)
{
```

```
struct stru arg ;
arg. x =1;
arg. y = '2';
f(arg) ;
return 0;
}
void f(struct stru param)
{
printf ("%d %d \n", param. x, param. y ) ;
}
```

Можно также создать указатель на структуру и передавать аргумент типа структуры по ссылке. Объявить указатель на структуру можно следующим образом:

struct stru *adr_pointer;

struct stru – переменная типа указатель на структуру struct stru.

Если мы передаем структуру по значению, то все элементы структуры заносятся в стек. Если структура простая и содержит мало элементов, то это не так страшно. Если же структура в качестве своего элемента содержит массив, то стек может переполниться.

При передаче по ссылке в стек занесется только адрес структуры. При этом копирования структуры не происходит, а такие появляется возможность изменять содержимое элементов структуры.

```
struct complex {
float x;
float y;} c1, c2;
struct complex *a ; /* объявление указателя */
a = & c1;
```

Указателю *a* присвоится адрес переменной *c1*. Получить значение элемента можно так:

```
(*a). x ;
```

Кроме данного способа получить значение, мы можем использовать так же оператор *a —> x,* который обычно и применяется, т. е. *(*a).x* эквивалентно *a —> x.*

6.2. Объединения

В языке Си определен еще один тип для размещения в памяти нескольких переменных разного типа. Это – объединение. Объявляется объединение так же, как и структура, например:

union u{

int i ;
char ch ;
long int l ;
};

Это объединение не задает какую-либо переменную. Оно задает шаблон объединения.

Можно объявить переменную:

union u alfa, beta ;

Можно было объявить переменные одновременно с заданием шаблона. В отличие от структуры для переменной типа union места в памяти выделяется ровно столько, сколько надо элементу объединения, имеющему наибольший размер в байтах. Так под переменную *alfa* будет выделено четыре байта, под переменную i – 2 байта; под переменную ch – 1 байт; под переменную l – 4 байта;

Синтаксис использования элементов объединения такой же, как и для структур:

u.ch = ‘ 5 ‘;

Для объединения разрешена также операция —>, если мы обращаемся к объединению с помощью указателя.

6.3. Битовые поля

Структура может содержать битовые поля. Целые компоненты типа *signed* и *unsigned* можно объявить битовыми полями шириной от 1 до 16 битов. Ширина битового поля и его необязательный идентификатор задаются следующим образом:

тип <идентификатор>: ширина;

где тип – это char, unsigned char, int или unsigned int .

Если идентификатор битового поля опущен, то число битов, заданное выражением ширина, распределяется в памяти, но поле при этом остается недоступным программе. Если при этом значение ширина равно нулю, то следующее поле будет начинаться со следующего слова памяти.

Пример:

```
struct str {
int i: 3;
unsigned j: 4;
int : 4
int k: 2;
```

} a ;

Для указанных в структуре полей задается следующее распределение памяти:

```
15 14 12 11 10  9  8  7  6  5  4  3  2  1  0
 x  x  x  x  x  x  x  x  x  x  x  x  x  x  x
      <—>  <--------> <--------> <---->
       k      не испол.      j        i
```

Целые поля хранятся в виде дополнения до 2, причем крайний левый бит является старшим. Для битового поля типа int (например, signed) старший бит интерпретируется как знаковый. Например, поле k типа signed int шириной 1 может содержать только значения –1 и 0, так как битовой шаблон 1 будет интерпретироваться как -1.

Пример 6.3:

```
#Include<stdio.h>
void main (void)
{
struct str {
int i:1;
unsigned j:4;
int l:4;
int k:2;
} a;
a.i=1;
printf("%d",a.i);
}
```

Результат работы программы: –1.

6.4. Доступ к отдельному биту

В отличие от других языков программирования язык Си обеспечивает доступ к одному или нескольким битам в байте или слове. Это имеет свои преимущества. Если многие переменные принимают только два значения, (такие переменные называют флагами), то можно использовать 1 бит.

Один из методов, встроенных в язык Си и позволяющих иметь доступ к биту, – это поля битов (bit-fields). В действительности поля битов – это специальный тип структуры, в котором определено, из каких бит состоит каждый элемент. Основная форма объявления такой структуры следующая:

```
struct имя_ структуры
 {
тип имя 1: длина в битах ;
тип имя 2: длина в битах ;
..... ;
тип имя N: длина в битах ;
}
```

В этом объявлении структур тип может быть одним из следующих: int, unsigned, signed. Имя I может быть пропущено, тогда соответственно число бит не используется (пропускается). Длина структуры всегда кратна восьми. Так, если указать:

```
struct onebit
{
unsigned one_bit : 1;
} obj ;
```

то для переменой obj будет выделено 8 бит, но использоваться будет только первый.

6.5. Переименование типов – typedef

Язык Си с помощью оператора typedef позволяет задавать новое имя уже существующим переменным. При этом не создается новый тип данных.

Например:

```
typedef char SIMBOL;
typedef unsigned UNSIGN;
typedef float real;
```

Достаточно часто используется оператор typedef с применением структур:

```
typedef struct st_tag {
char name[30];
int kurs;
char group[3];
int stip;
} STUDENT;
```

Теперь для определения переменной можно использовать

```
struct st_tag avar;
```

а можно использовать

```
STUDENT avar;
```

7. ВЫДЕЛЕНИЕ ПАМЯТИ И УПРАВЛЕНИЕ ЕЮ

7.1. Определение размера выделяемой памяти (операция sizeof)

С помощью этой операции можно определить в байтах размер памяти, которая соответствует идентификатору или типу. Выражение с операцией sizeof имеет следующий формат

sizeof (выражение)

Выражение – это либо идентификатор, либо имя типа, заключенное в круглые скобки:

sizeof (имя_объекта) ;

Идентификатор не может относиться к полю битов или быть именем функции.

Если имя типа описывает массив, то результатом является размер всего массива, а не размер указателя, соответствующий имени массива.

Пример 7.1:

```
# include <stdio. h>
# include <conio. h>
void main (void)
{
float mas[100];
clrscr();
printf( "Размер одного элемента массива %d в байтах\n", sizeof ( mas[20]));
printf( "Размер всего массива %d в байтах\n", sizeof ( mas));
getch();
   }
```

Результат выполнения программы:

Размер одного элемента массива 4 байта.

Размер всего массива 400 байт.

7.2. Динамическое выделение памяти

Язык Си позволяет выделять память динамически, т.е. во время работы программы. Как было показано ранее, по области видимости переменные могут быть глобальными и локальными.

Для глобальных переменных отводится фиксированная часть памяти на все время работы программы. Локальные переменные хранятся в стеке. Между ними находится область свободной памяти

для динамического распределения во время работы программы. Принятое распределение памяти в программах на языке Си показано на рис.7.1.

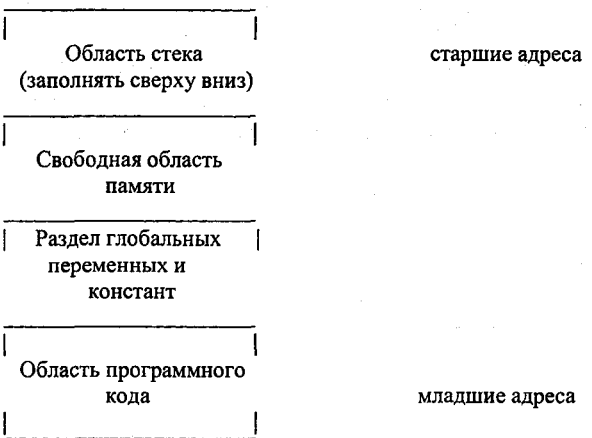


Рис. 7.1

Наиболее важными функциями для динамического распределения памяти являются malloc() и free(). Прототипы этих функций хранятся в заголовочном файле alloc.h

Выделение памяти размером size байт осуществляется функцией malloc() .

Синтаксис использования:

```
unsigned size; / * размер блока памяти в байтах */
void *malloc (size) ;
void free(void *p);
```

Функция malloc() возвращает указатель на первый байт выделенного блока памяти либо NULL, если для выделения памяти указанного размера нет места.

Действие функции free(), обратное действию функции malloc(). Эта функция освобождает ранее выделенную область памяти.

Выделение памяти и ее обнуление осуществляется функцией calloc() .

Синтаксис:

unsigned n_elem , size_elem ; /* число элементов и размер одного элемента в байтах */

void *calloc (n_ elem, size_elem);

Функция возвращает указатель на выделенный блок памяти, либо NULL, если память нельзя выделить.

Из приведенных описаний видно, что обе функции возвращают указатель на любой тип (void), следовательно, при обращении к этим функциям необходимо использовать явную операцию преобразования типов.

Освобождение ранее выделенной памяти осуществляется функцией

```
void free (ptr ):
char *ptr;
ptr = (char*) malloc(...) ;
...
free (ptr);
```

Пример 7.2:

Выделение памяти под массивы

```
# include<alloc.h>
# include<stdio.h>
# include<conio.h>
# include<stdlib.h>
int main (void)
{
int *mas_1;
float *mas_2;
clrscr ( );
mas_1 = (int*)malloc(10*sizeof (int));
mas_2 = (float*)calloc( 1000,sizeof(float));
if (mas_1==NULL || mas_2== NULL )
{
printf ("Не хватает памяти ");
getch ( );
exit (0);
}
printf("Начало mas_1 %p\n",mas_1);
printf("Начало mas_2 %p\n",mas_2);
```

```
getch ( );
free (mas_1);
free (mas_2);
return (1);
}
```

В C++ для выделения и освобождения памяти используется также функции new() и delete ().

Динамическое распределение памяти удобно использовать тогда, когда заранее неизвестно количество используемых переменных. В частности, этот механизм используется для создания массивов с изменяемым количеством элементов.

7.3. Динамические массивы

При работе с массивами память для размещения их элементов может выделяться динамически. Для одномерных массивов доступ к элементам осуществляется как обычно: по номеру индекса элементов.

В случае использования двумерных массивов (матриц) их элементы располагаются в памяти в виде линейного массива. При этом доступ к отдельному элементу осуществляется не по номеру строки и столбца, а также как и для одномерных массивов – по номеру элемента. Таким образом, программист сам должен контролировать правильность доступа к нужному элементу массива.

Чтобы работать с матрицами в естественной форме, целесообразно выделять память с использованием двойных указателей. Рассмотрим фрагмент программы:

```
a = (int**) malloc(m*sizeof (int*));
for (i = 0: i <m; i++)
a[i] = (int*)malloc(n*sizeof (int));
```

В результате выполнения первой строки будет выделена память для размещения *m* указателей на данное типа int, а переменной *a* будет присвоено значение адреса начала выделенной области. В результате выполнения цикла for будет выделена память для размещения *m*n* элементов типа int в соответствии со следующей схемой:

```
1-й этап                                    2-й этап (цикл for)
int **a -> int*a[0] ->a[0] [0] a[0] [1] a[0] [2] … a[0] [n]
int* a[1] -> a[1] [0] a[1] [1] a[1] [2] … a[1] [n]
```

int* a[2] -> a[2] [0] a[2] [1] a[2] [2] … a[2] [n]

…

int* a[m] -> a[m][0] a[m][1] a[m][2] …a[m] [n]

Таким образом, в памяти будет выделена область для размещения элементов матрицы размером m*n, а доступ к отдельному элементу будет осуществляться по индексу строки и столбца.

Пример 7.3: Работа с динамическими массивами

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
int** input (int, int); //Функция ввода матрицы
void output (int**, int, int); // Функция вывода матрицы
void making (int**, int, int);
void main(void)
{
 int m,n;
 int** p;
 clrscr( );
 puts("Введите размер исходной матрицы ");
 printf("число строк = ");
 scant("%d",&m);
 printf("число столбцов = ");
 scanf("%d",&n);
 p = input(m,n);
 making(p,m,n);
 output (p,m,n);
 free(p);
 getch( );
}
int** input(int m, int n)
{
 int i, j ;
int **a;
a=(int**)malloc(m*sizeof(int*));
for(i=0; i<m; i++)
 {
 a[i]=(int*)malloc(n*sizeof(int));
 for(j=0; j<n; j++)
```

```
{
a[i] [j] =0;
}
}
for(i=0; i<m; i++)
for(j=0; j<n; j++)
{
printf("\nВведите A(%1d,%1d) элемент матрицы : ", i+1, j+1);
scanf("%d", &a[i] [j]);
}
return a;
}
/*Функция заменяет элементы с четной суммой индексов на противоположные */
void making(int **a, int m, int n)
{
int i, j;
for (i=0; i<m; i++)
for(j=0; j<n; j++)
if((i=j)%2 = = 0)
    a[i] [j] = -a[i] [j];
}
void output(int **z, int m, int n)
{
int i,j;
printf("\nРезультирующая матрица \n");
for(i=0; i<m; i++)
{
for(j=0; j<n; j++)
printf("%8d", z[i] [j]);
printf("\n");
}
}
```

7.4. Динамические структуры

Каждая структура данных характеризуется, во – первых, взаимосвязью элементов и, во-вторых, набором типовых операций над этой структурой. В случае динамической структуры важно знать:

1) каким образом может расти и сокращаться структура данных;
2) каким образом можно включить в структуру новый элемент и удалить существующий;
3) как можно обратиться к конкретному элементу структуры для выполнения над ним определенной операции.

Доступ по индексу, как это осуществляется в массивах, здесь не удобен, так как любой элемент при изменении размеров структуры может изменить свою позицию. Поэтому доступ к элементам динамической структуры относительный: найти следующий (предыдущий) элемент по отношению к текущему, найти последний элемент и т.д.

Одной из простейших и в то же время типичных динамических структур данных является *очередь*. Проблема очереди возникает всегда, когда имеется некоторый механизм обслуживания, который может выполнять заказы последовательно, один за другим. Если к моменту поступления нового заказа данное устройство свободно, оно немедленно приступает к выполнения заказа. Если же оно занято, то заказ поступает в конец очереди других заказов. Когда устройство освободится, оно приступает к выполнению заказа из начала очереди. Если заказы поступают нерегулярно, очередь то удлиняется, то укорачивается и даже может оказаться пустой (рис 7.2).

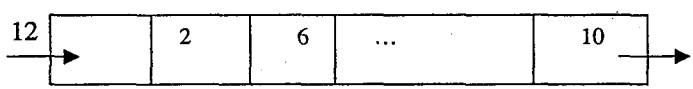


Рис. 7.2

Очередь часто называют системой, организованной и работающей по принципу FIFO (first in – first out). Основными операциями с очередью являются добавление нового элемента в конец и удаление элемента из начала очереди.

Рассмотрим, как может быть реализована работа с очередью на Си с использованием структур, указателей на структуры и функции динамического выделения/освобождения памяти.

В программе, приведённой ниже, работа с очередью ведётся с помощью двух функций:

Функция insert() отводит память под очередной элемент, заносит в него нужную информацию и ставит в конец очереди.

Функция take_off() удаляет из очереди её первый элемент, освобождает память из-под него и перемещает указатель начала очереди на следующий элемент.

В случае попытки удаления элемента из пустой очереди параметр ошибки error получает значение 1.

```
#include <studio.h>
#include <alloc.h>
#define QUEUE struct queue
QUEUE
{ int info; // поле информации элемента очереди
 QUEUE *next; // поле ссылки на следующий элемент очереди
};
void insert (QUEUE **q, int item)
{
QUEUE *tek = *q; // указатель на текущее звено очереди
QUEUE *pred = 0; // pred содержит адрес последнего элемента
QUEUE *new_n;
    while(tek) //просматриваем очередь до конца
{ pred=tek; tek= tek→next;}
 new_n→info=item;
if(pred) //очередь не пуста
{new_n→next=pred→next;
pred→next=new_n;
}
else { q=new_n; (*q) →next=0;}
 }
    int take_off(QUEUE **q, int *error)
    {int value=0; //значение возвращаемого элемента очереди
    QUEUE *old= *q; //указатель на старую голову очереди
    if(*q) //если очередь не пуста
            {value=old→info; q=(*q)→next;
            free(old); *error=0; //элемент удален из очереди
            }
     else *error=1;
     return value;
 }
 void main(void)
{
int error, j;
```

```
QUEUE*q1, q2;
   for(j=12;j<=15;j++)
insert(&q1,j);
    for(j=1;j<=4;j++)
insert(&q2,take_off(&q1,&error));
for(j=1;j<=4;j++)
printf("\n q2:%d", take_off(&q2,&error))  ;
   }
```

Другая часто встречающаяся структура данных – *стек (магазин)*– отличается от очереди тем, что она организована по принципу LIFO (last in – first out). Операции включения и удаления элемента в стеке выполняются только с одного конца, называемого *вершиной* стека. Когда новый элемент помещается в стек, то прежний верхний элемент как бы "проталкивается" вниз и становится временно недоступным. Когда же верхний элемент удаляется с вершины стека, предыдущий элемент "выталкивается" наверх и опять является доступным (рис.7.3).

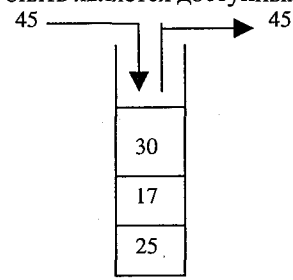


Рис. 7.3

Потребность в организации стека возникает при решении, например, такой задачи. Пусть имеется текст, сбалансированный по круглым скобкам.

Необходимо построить таблицу, в каждой строке которой будут находиться координаты соответствующих пар скобок, т.е. для текста

```
( ....(.....    .)....(.....  ......).....)
  0     17      42   61      84   95
```

таблица должна быть такой

```
17  42
61  84
 0  95
```

Поскольку текст сбалансирован по круглым скобкам, то как только встретится “)”, это будет пара для последней пройденной “(“.

Поэтому алгоритм решения данной задачи может быть следующим: будем идти по тексту и как только встретим “(“, занесем её координату в стек; как только встретится “)”, возьмём из стёка верхнюю координату и распечатаем её вместе с координатой данной ”)”.

Рассмотрим программу, в которой реализована работа со стеком. В ней использованы функции:

push() – положить элемент на вершину стека,

pop() – вытолкнуть верхний элемент из стёка,

peek() – прочитать значение верхнего элемента, не удаляя сам элемент из стека.

```
#include<stdio.h>
#include<alloc.h>
#define STACK struct stack
STACK
{int info; //поле значения элемента стека
STACK *next; //поле ссылки на следующий элемент стека
};
void push (STACK **s, int item)
{
STACK *new_n;
new_n=( STACK*) malloc(sizeof(STACK));//запросили память под
//новый элемент
new_n-->info=item; //заносим значение в новый элемент
new_n-->next=*s; //новый элемент стал головой стёка
*s=new_n; //указателю *s присвоили адрес головы стёка
}
int pop(STACK **s, int *error)
{
STACK *old=*s;
int info=0;
if(*s) //стек не пуст
{ info=old-->info; s=(*s)->next;
free(old); //освобождаем память из-под выбранного элемента
*error =0; // элемент удален успешно
}
else *error=1;
```

```
return (info) ;
}
int peek (STACK **s, int *error)
{
if (*s) {*error=0; return (*s) --> info; }
else { *error=1; return 0; }
}
void main()
{
int error, i;
STACK *s1, *s2;
push (&s1, 42) ;
printf (" \ n peek (s1)= % d ", peek (&s1 , & error ) ) ;
push ( &s1 , 53 ) ;
printf ("\ n peek ( s1 )=% d " , peek ( &s1 , & error ));
push (&s1 , 72 ) ;
printf ( "\n peek ( s1) = %d" , peek (&s1, & error));
push (&s1, 86);
printf ( "\n peek (s1)=%d", peek (&s1, &error));
for ( i=1; i<=4; i ++)
push (&s2, pop(&s1, &error));
for (i=1; i<= 4; i ++)
printf ("\n pop(&s2)=%d", pop(&s2, &error));
}
```

Стеки и очереди являются одними из разновидностей более широкого класса структур данных – списков. Связанный список – это структура следующего вида (рис.7.4)

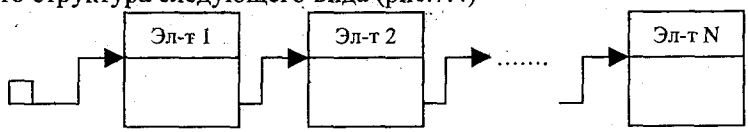


Рис. 7.4

Это *простой однонаправленный список*, в котором каждый элемент, кроме последнего, имеет ссылку на следующий элемент и поле информации. Можно организовать также кольцевой список (в нём последний элемент будет содержать ссылку на первый) или двунаправленный список (когда каждый элемент, кроме первого и последнего, имеет две ссылки: на предыдущий и следующий эле-

мент) и т.д. (рис. 7.5). Кроме того, можно поместить в начале списка дополнительный элемент, называемый заголовком списка. Как правило, заголовок списка используется для хранения информации обо всём списке. В частности, он может содержать счётчик числа элементов в списке. Наличие заголовка приводит к усложнению одних и упрощению других программ, работающих со списками.

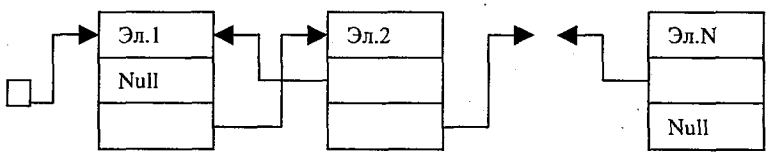


Рис. 7.5

При работе со списком могут быть полезны следующие базовые функции:

insert() – добавить новый элемент в список таким образом, чтобы список оставался упорядоченным в порядке возрастания по значению одного из полей;

take_out() – удалить элемент с заданным полем (если он есть в списке);

is_ present() – определить, имеется ли в списке заданный элемент;

displey() – распечатать значения всех полей элементов списка;

destroy_list() – освободить память, занимаемую списком.

Довольно часто при работе с данными бывает удобно использовать структуру с *иерархическим* представлением, которые хорошо изображаются с помощью *дерева* (рис.7.6).

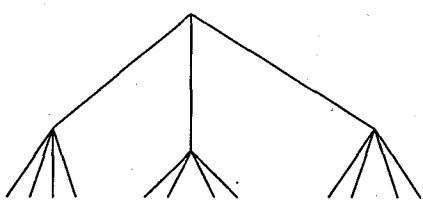


Рис. 7.6

Дерево состоит из элементов называемых узлами (вершинами). Узлы соединены между собой направленными дугами. В случае X –> Y вершина X называется предшественником (родителем), а Y

– преемником (сыном). Дерево имеет единственный узел – корень, у которого нет предшественников. Любой другой узел имеет ровно одного предшественника. Узел, не имеющий преемника, называется *листом.*

Рассмотрим работу с *бинарным* деревом (в котором у каждого узла может быть только два преемника : левый и правый сын). Необходимо уметь:

– построить дерево;

– выполнить поиск элемента с указанным значением в узле;

– удалить заданный элемент;

– обойти все элементы дерева (например, чтобы над каждым из них провести некоторую операцию).

Обычно бинарное дерево строится сразу упорядоченным, т. е. узел левого сына имеет значение меньше, чем значение родителя, а узел правого сына – большее. Например, если приходят числа 17,18,6,5,9,23,12,7,8, то построенное по ним дерево будет выглядеть так как это представлено на рис.7.7.

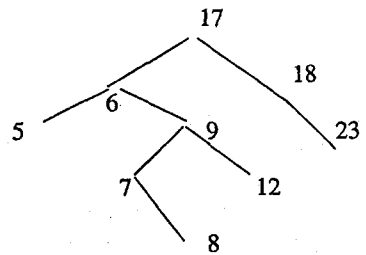


Рис. 7.7

Для того, чтобы вставить новый элемент в дерево, надо найти для него место. Для этого, начиная с корня, будем сравнивать значения узлов (Y) со значением нового элемента (NEW). Если NEW<Y, то пойдём по левой ветви; в противном случае – по правой ветви. Когда дойдём до узла, из которого не выходит нужная ветвь для дальнейшего поиска, это означает, что место под новый элемент найдено. Путь поиска места для числа 11 в построенном выше дереве показан на рис.7.8.

При удалении узла из дерева возможны три ситуации:

- исключаемый узел – лист (в этом случае надо просто удалить ссылку на данный узел);

- из исключаемого узла выходит только одна ветвь;

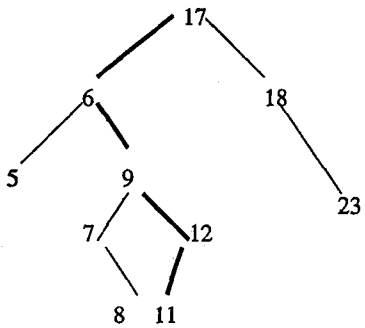


Рис. 7.8.

из исключаемого узла выходят две ветви (в таком случае на место удаляемого узла надо поставить либо самый правый узел левой ветви, либо самый левый узел правой ветви для сохранения упорядоченности дерева). Например, построенное раннее дерево после удаления узла 6 может стать таким, как показано на рис. 7.9.

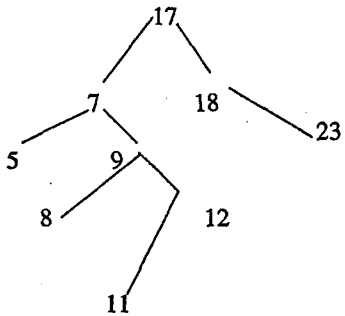


Рис. 7.9.

Рассмотрим задачу обхода дерева. Существуют три способа обхода, которые естественно следуют из самой структуры дерева.

1) Обход слева направо: A,R,B (сначала посещаем левое поддерево, затем – корень и, наконец, правое дерево).
2) Обход сверху вниз: R, A, B (посещаем корень до поддеревьев).
3) Обход снизу вверх: A,B,R (посещаем корень после поддеревьев).

Интересно проследить результаты этих трех обходов на примере записи формул в виде дерева.

Например, формула

(a+b/c)*(d-e*f)

будет представлена так (рис. 7.10).

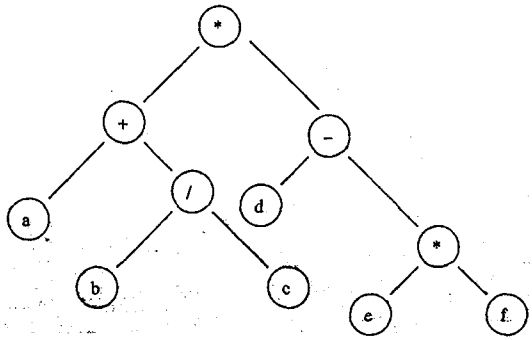


Рис. 7.10.

(дерево формируется по принципу: операция – узел, операнды – поддеревья).

Обход 1 дает обычную *инфиксную* запись выражения (правда, без скобок).

Обход 2 – *префиксную* запись +a/bc–d*ef.

Обход 3 – *постфиксную* (ПОЛИЗ – польская инверсная запись): abc/+dcf*–* .

В качестве примера работы с древовидной структурой данных рассмотрим решение следующей задачи.

Вводятся фамилии абитуриентов; необходимо распечатать их в алфавитном порядке с указанием количества повторений каждой фамилии.

В программе будет использована рекурсивная функция ***der()***, которая строит дерево фамилий, а также рекурсивная функция для печати дерева ***print_der()***, в которой реализован первый способ обхода дерева.

```
#include<alloc.h>
#include<stdio.h>
#defint TREE struct der
TREE
```

```
{ char *w;
int c;
TREE *l;
TREE *r;
};
TREE *der (TREE *kr, char *word)
{
int sr;
if (kr==NULL)
{
kr=(TREE *)malloc (sizeof(TREE));
kr-->w=word; kr-->c=1;
kr-->l=kr-->r=NULL;
}
else if ((sr=strcmp(word, kr-->w))==0) kr-->c++;
else if (sr<0) kr-->l = der(kr-->l, word);
else kr-->r = der(kr-->r, word);
return kr;
}
void print_der(TREE *kr)
{
if (kr)
{ print_der (kr->l);
printf("слово – %–20s \t кол – во повтор. - %d\n", kr->w, kr->c);
print_der (kr-->r);
}
}
void main ()
{ int i;
TREE *kr;
static char word [40] [21];
kr=NULL; i=0;
puts ("Введите <40 фамилий длиной <20 каждая");
scanf("%s", word[i]);
while(word[i] [0]!='\0')
{ kr=der(kr,word[i]);
scanf ("%s", word[++i]);
}
print_der(kr);
}
```

8. ОРГАНИЗАЦИЯ РАБОТЫ С ФАЙЛАМИ

8.1. Понятие потока

Язык Си, кроме стандартного ввода данных с клавиатуры и вывода результатов на экран, предоставляет также возможность обмена при операциях ввода/вывода с внешними устройствами, в том числе, с файлами на диске.

В Си не предусмотрены никакие предопределенные структуры файлов (такие как последовательного или прямого доступа). Все файлы рассматриваются как последовательности, *потоки байтов.*

Поток– это источник или приемник данных. Его можно связать с каким- либо внешним устройством, например, с принтером, клавиатурой, диском и т.д.

В языке Си определены два вида потоков: текстовый и бинарный.

Текстовый поток – это последовательность строк. Каждая строка содержит нуль и более символов и обязательно заканчивается символом – терминатором новая строка '\n'.

Бинарный или двоичный поток – это последовательность двоичных кодов (байтов), как правило, используемая для запоминания кодов машинного языка.

Поток соединяется с файлом или другим устройством посредством его открытия. Эта связь разрывается путём закрытия потока.

Открытие потока осуществляется функцией *fopen(),* а закрытие – функцией *fclose().*

Когда программа начинает работу, то автоматически открываются три потока: stdin, stdout, stderr.

STDIN –стандартный входной поток (по умолчанию связан с клавиатурой).

STDOUT – стандартный выходной поток (по умолчанию связан с экраном монитора).

STDERR – поток стандартных ошибок (связан с экраном монитора).

8.2. Открытие файла

Для того, чтобы связать программу с файлом на диске, необходимо определить в программе переменную типа указатель на файл:

*FILE *fp;*

где *FILE* – это структура, определённая в *<stdio.h>* с помощью средства *typedef* и содержащая некоторую информацию о файле.

После того, как указатель на файл описан, его можно связать с конкретным файлом в момент открытия данного файла. Открыть файл можно с помощью следующей конструкции:

fp = fopen (" имя_ файла", " режим_ доступа");

Константы в круглых скобках имеют следующий смысл:

имя_файла – это параметр, характеризующий имя и тип используемого файла на диске, например, "test.txt";

режим доступа – это параметр, характеризующий как должен использоваться данный файл. Этот параметр может принимать следующие значения:

'r' –файл открывается для чтения (файл должен существовать на диске);

'w' – файл открывается для записи; если файла с указанным именем нет, то он создаётся; если файл существует, то старое содержимое файла уничтожается;

'a' – файл открывается, либо создаётся для дозаписи в конец файла;

'r+' – файл открывается для чтения и записи (файл должен существовать).

'w+' – файл открывается для чтения и записи; старое содержимое, если файл существовал, теряется.

'a+' – файл открывается, либо создаётся для чтения уже существующей информации и добавления новой в конец файла.

Обычно по умолчанию файл считается текстовым, однако можно явно указать с каким файлом будет работать программа: текстовым (t) или бинарным (b), например:

r + t, w + b, at, wt +, и т. д.

Операцию открытия файла можно записать и так:

*FILE *fp = fopen ('test. txt', 'w');*

Теперь указатель fp будет связан с файлом на диске с именем 'test.txt'. Во всех операциях с файлом теперь следует указывать fp.

8.3. Закрытие файла

После того, как работа с файлом закончена, его следует закрыть. Операция закрытия файла имеет вид:

fclose (имя файла), например: fclose (fp);

Так как операции с файлами осуществляются посредством промежуточной буферизации, то операция закрытия осуществляет выталкивание содержимого буфера.

Если операции закрытия файла в программе нет, то по окончанию программы все открытые файлы будут закрыты автоматически. Следует, однако, помнить, что настройка среды ограничивает число одновременно открытых файлов, поэтому следует явно определять операцию их закрытия.

8.4. Операции ввода/вывода в файл (из файла)

Различают следующие операции ввода/вывода:

– ввод-вывод отдельных символов;

– построчный ввод-вывод;

– форматированный ввод-вывод;

– ввод/вывод объектов (данных простых и сложных типов).

Ввод-вывод отдельных символов. Эти операции осуществляются с помощью следующих функций:

1) getc() и fgetc() – ввод символа из файла;

2) putc() и fputc () – вывод символа в файл.

Разница между этими функциями заключается в том, что fgetc() и fputc() – это собственно функции, а getc() и putc() – это макросы.

В программе эти функции записываются следующим образом:

ch= fgetc (fp);

ch принимает значение символа из файла, на который указывает fp.

fputc (ch, fp); – значение символа ch выводится в файл, на который указывает fp.

Пример 8.1. Посимвольного копирования:

```
# include <stdio.h>
void main (void)
{
char ch;
FILE *ist, *pri;
if((ist = fopen("text_1.txt", "r")) = =NULL)
{
printf ("Не могу открыть файл источник \ n");
return;
```

```
}
if ((pri = fopen("text_2.txt","w")) = = NULL)
{
printf (" Не могу открыть файл приемник \n");
return;
}
while ((ch = fgetc(ist)) != EOF)
fputc(ch, pri);
fclose (ist);
fclose(pri);
}
```

Ввод/вывод строк. Операции ввода/вывода строк осуществляется с помощью функций fgets() и fputs () соответственно.

Функция fgets() имеет три аргумента:

*char *fgets(s, n, stream),*

где s- указатель на местоположение строки;

n- предельная длина считываемой строки;

stream- указатель на файл, который будет читаться.

Функция fputs() имеет два аргумента

int fputs(s, stream),

где s- указатель на местоположение строки символов, которая будет записываться в файл;

stream- указатель файла.

Пример 8.2: Считывание файла строка за строкой и копирование.

```
# include<stdio.h>
void main (void)
{
 int n = 80;
char string[80];
FILE *fpr, *fis;
fis = fopen ("text_1", "r");
fpr = fopen ("text_ 2", "w");
while ((fgets(string, n, fis)) != NULL)
 fputs(string, fpr);
fclose (fis);
fclose (fpr);
}
```

При достижении конца файла функция fgets возвращает NULL.

Форматированный ввод/вывод. Форматированный ввод/вывод осуществляется функциями fprintf() и fscanf(). Эти функции работают аналогично функциям printf() и scanf(). Разница заключается в дополнительном аргументе- ссылке на файл, с которым они работают. Этот аргумент указывается первым в списке.

Например:

```
# include <stdio.h>
void main (void)
{
FILE *fp;
 int d;
fp = fopen ("file_1.dat", "a+b");
fscanf (fp, "%d", &d);
fclose (fp);
fp = fopen ("file_2. dat", "r+b");
fprintf (fp,"%d", d);
fclose (fp);
}
```

Здесь продемонстрировано повторное использование указателя на файл после закрытия файла.

Блочный ввод-вывод осуществляется функциями fread() и fwrite().

Определение:

fread (ptr, size, n_ obg, stream);

Читать из файла stream в массив ptr (указатель) не более n_obg объектов размером size.

fwrite (ptr, size, n_obg, stream);

Пишет из массива с указателем ptr n_obg объектов размера size в поток stream (файл).

9. КЛАССЫ ПАМЯТИ И ОБЛАСТИ ВИДИМОСТИ ПЕРЕМЕННЫХ

9.1. Классы памяти

Класс памяти определяет «время жизни» объекта. Под объектом понимается идентификатор переменной, функция либо указатель функции. Кроме этого, класс памяти в совокупности с местоположением переменной в программе, определяет область видимости переменной.

Различают два вида объектов: глобальные и локальные.

Объекты с глобальным временем жизни существуют и имеют значение на протяжении всего времени исполнения программы.

Все функции и их указатели – глобальные объекты. Локальные объекты «захватывают» новую область памяти всякий раз, когда управление передается блоку, в котором они описаны.

Если переменная описана вне всяких блоков, то она считается глобальной. Это описание на так называемом внешнем уровне.

Блок– это описание функции, составная команда, либо часть программы, заключенная в фигурные скобки

```
{
это блок
}
```

Блоки могут быть вложенными:

```
{
 {
это вложенный блок
 }
}
```

Переменные, описанные внутри некоторого блока, являются локальными. При каждом входе в блок им выделяется новая область памяти, а при выходе из блока память освобождается и значение переменной, следовательно, теряется. Локальные объекты видны с момента их описания до конца блока.

Если при выходе из блока нужно сохранить значение переменной, то ее следует описать как статическую в данном блоке, например:

```
{
static int a;
```

...
}

Теперь при выходе из блока переменная *a* будет сохранять в памяти свое место и значение и будет «видна» только в этом блоке, а также в блоках вложенных в него.

Глобальные объекты видны с момента их описания до конца файла всем функциям (блокам).

Если имя локального объекта (переменной) совпадает с именем глобального объекта, то локальный объект маскирует глобальный в этом блоке.

Область действия меток – функция, в которой метка используется.

Пример 9.1:

Область видимости переменных.

```
#include <stdio. h>
int k = 1; /* k = 1 */
void main(void)
{
printf("k = %d \n", k); // Результат: k = 1
{
//1-й вложенный блок
int k = 2, m = 5;
printf(" k = %d \t m = %d \n", k, m); // Результат: k =2, m = 5
{
//2-й вложенный блок
int k=0;
printf(" k = %d \t m = %d \n", k, m); // Результат: k=0, m=5
}
printf(" k = %d \n", k); // Результат: k=2
}
printf(" k = %d \n", k); // Результат:k=1
}
```

В приведенной программе четыре уровня видимости:

1) внешний уровень;
2) тело функции main();
3) 1-й вложенный блок;
4) 2-й вложенный блок

Внешний уровень – уровень файла. Объект внешнего уровня виден "всем в файле". Время его жизни – глобальное. На локальном уровне внешний объект может быть переопределен.

9.2. Описатели классов памяти

Различают два класса объектов: глобальные и локальные, но используют четыре описателя классов: extern, static, auto и register. Назначение и применение их приведено в табл. 9.1

Таблица 9.1

КЛАССЫ ПАМЯТИ

Класс памяти	Ключевое слово	Время действия	Область действия
1. Автоматический	auto	временно	локальная
2. Регистровый	register	временно	локальная
3. Статический	static	постоянно	локальная
4. Внешний	extern	постоянно	глобальная (все файлы)
5.Внешний статический	static	постоянно	глобальная (один файл)

Классы 1, 2 и 3 описываются внутри функций; Классы 4 и 5 описываются вне функций.

По умолчанию (без спецификаторов класса памяти) переменные, описанные внутри функции, являются автоматическими. Переменные, описанные вне функций по умолчанию – глобальные.

Класс памяти extern указывает, что глобальная переменная описана где-то в другом месте (в этом или другом файле). Таким образом, описатель extern позволяет только сослаться на описание, сделанное в другом месте.

Для того, чтобы описание extern было корректным, необходимо чтобы описание самой переменной существовало только один раз в любом из файлов, образующих текст программы, например:

Файл 1	файл 2	файл 3
...	...	...
float p=0.33;	extern float p;	extern float p;

Замечание: **Описание вида:** ***extern тип имя_ переменной = значение;*** **недопустимо.**

Если переменная внутри функции имеет класс памяти extern, то она может быть описана только после описания этой функции, например:

Неправильно

```
...
main ()
 {
...
 }
func_1()
 {
k = k+2;
 }
int k = 10;
...
```

Правильно

```
...
main ()
 {
...
 }
func_1()
 {
extern int k;
k = k+2;
 }
int k =10;
...
```

Ошибка состоит в том, что область видимости переменной начинается с точки ее описания.

Глобальные переменные могут быть инициализированы:

явно: int k = 10; static float a = 20.52;

неявно int k; static float a;

При неявной инициализации переменным присваиваются нулевые значения.

Класс памяти *static* на внешнем уровне маскирует описание переменных внутри файла, в котором они описаны. Другим файлам они не недоступны.

Класс памяти *register* предполагает хранение переменной во внутреннем регистре процессора. Время жизни и область видимости регистровой переменной такая же, как и у автоматической. Если компилятор не может разместить эту переменную в регистре, то она трактуется как автоматическая. По умолчанию значение регистровых переменных не определено.

К регистровой переменной нельзя применить операцию & – определение адреса. Их нельзя описывать на внешнем уровне.

9.3. Правила инициализации переменных

Инициализация переменных определяется конструкцией вида:

Переменная = <инициализатор>;

Основные правила:

1. Если описание переменной начинается со служебного слова EXTERN, то явная инициализация не допустима.

2. Переменные, описанные на внешнем уровне можно явно инициализировать. При неявной инициализации они получают нулевые значения.

3. Статическим переменным можно присваивать значения константного выражения. По умолчанию они инициализируются нулем.

4. Автоматические и регистровые переменные инициализируются при каждом входе в блок, где они описаны. По умолчанию их значения не определены.

5. Данные составного типа (массивы, структуры) можно инициализировать на внешнем или на внутреннем уровне. В последнем случае они инициализируются при входе в блок. При выходе из блока значения автоматических и регистровых переменных теряются.

Инициализация переменных составного типа.

Синтаксис:

Имя переменной ={<список инициализаторов>};

Основные правила:

1.Значения константных выражений из списка инициализаторов присваиваются элементам объекта в порядке их следования.

2.Если в списке инициализаторов меньше элементов, чем в объекте составного типа, то оставшиеся элементы неявно инициализируются 0, а если больше, то регистрируется ошибка.

Пример:

```
int array [3] [3]=
{
{1,1,1}, //инициализация первой строки
{2,2,2}, //инициализация второй строки
{3,3,3}, // инициализация третьей строки
 }
```

Внутренние скобки играют важную роль, особенно в тех случаях, когда имеются сложные описания, например, массивы структур. Здесь нужно быть особенно внимательным.

Пример неверной инициализации:

```
struct
{ int a1,a2,a3,;} st [2] [3]=
{
 {1,2,3}, {4,5,6}, {7,8,9},
 {10,11,12}, {13,14,15}, {16,17,18}
};
```

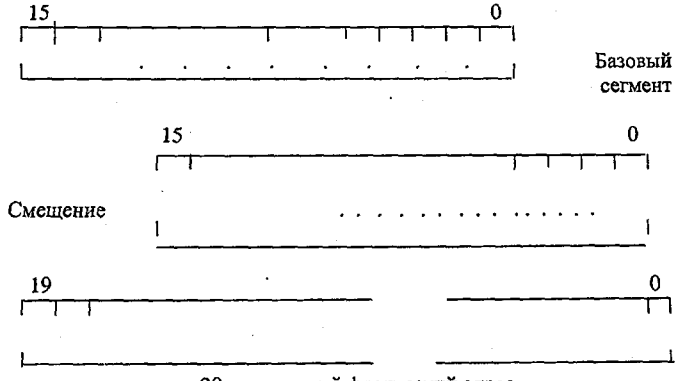
Примеры строковых инициализаторов

```
char str[] ="abc";
char str[3] ="abcd" //неверно
char *str ="abcd";
```

Указатель str будет инициализирован адресом массива типа char, содержащего символы ‘a’,’b’,’c’,’d’,’\0’.

9.4. Модели памяти

Архитектура микропроцессора предусматривает разбиение оперативной памяти на физические сегменты. Размер каждого сегмента равен 64 Кбайт (К=1024). Вычисление физического адреса осуществляется путем сложения 16 - разрядного адреса сегмента, сдвинутого на 4 разряда, с адресом смещения в соответствии со следующей схемой:



20 – разрядный физический адрес

Адрес сегмента адресует память с «точностью» до 16 байт (или с точностью до параграфа); смещение адресует память с точностью

до байта в пределах 64 Кбайт, размещение в памяти которых определяется начальным адресом сегмента.

Распределение памяти для программы на языке Си осуществляется следующим образом:

1. По умолчанию программе выделяется суммарный объем памяти, равный 128 Кбайт.

- 64 Кбайт для кода программы;
- 64 Кбайт для статистических данных и стека.

Такая модель памяти называется малой (*small*). В малой модели для доступа к объектам программы (коду или данным) используются указатели типа *near* – ближние указатели.

2. Если размер кода программы превышает 64 Кбайта, а для размещения стека и данных достаточно 64 Кбайта, то применяется средняя модель памяти (*medium*) . В такой модели для функций используются дальние указатели *far*, а для данных *near*.

3. Если размер программного кода не велик, а данные требуют больше, чем 64 Кбайт, то используется компактная (*compact*) модель памяти.

В такой модели доступ к объектам программы осуществляется с помощью указателей типа *far*, а в максимальной модели с помощью указателей типа *huge*.

4. Если как для данных, так и для кода требуется больше одного сегмента (>64 Кбайт), то применяется либо большая (*large*) модель, либо огромная(*huge*) модель.

Разница между указателями far и huge заключается в том, что в адресной арифметике при вычислении far указателей на данные используется 16- битное смещение. Это ограничивает размер отдельного элемента данных значением 64 Кбайт. Указанное ограничение снято в огромной модели памяти. Адреса huge позволяют использовать массивы, размер которых превышает 64 Кбайт. В адресной арифметике используются все 32 разряда, а тип результата unsigned long.

Работая в любой модели памяти, программист имеет возможность ее модификации, применяя явно модификаторы near, far и huge.

Так, например, если за модификатором near, far или huge следует идентификатор, то соответствующий программный объект бу-

дет размещен либо в стандартном сегменте (для near), либо в другом сегменте (для far и huge).

Объявление

char far ch;

означает, что адрес ch будет иметь тип far.

9.5. Управление экраном и памятью в текстовом режиме

В текстовом режиме можно управлять размерами активного окна, курсором, цветом фона и цветом текста. К функциям управления экраном относятся функции задания и очистки экрана, управления курсором и управление цветом. Функции задания окна и его очистки были приведены в первой главе. Это функции window() и clrscr().

Следующая функция – это функция позиционирования курсора в пределах активного окна. Ее прототип

void gotoxy(int x, int y);

здесь x и y – координаты относительно активного окна.

Функции

int wehrex(); int wehrey();

возвращают координаты x и y текущего положения курсора.

Установкой или заменой текстового режима управляет функция textmode() с прототипом

int textmode(int mode);

здесь аргумент mode задает режим печати, а именно количество строк на экране, количество символов в строке и цвет. Он может быть задан в виде макроса или в виде числового эквивалента [2].

Современная техника позволяет выводить текст в цвете. Можно определить как цвет текста, так и цвет фона. Однако использовать цвет могут только специальные функции, связанные с оконным интерфейсом. Такие функции как printf() для этого не предназначены.

Для изменения цвета текста предназначена функция textcolor(). Ее прототип

void textcolor(int color);

Аргумент color может принимать значения от 0 до 15. Каждому значению соответствует свой цвет. (Число может быть заменено

соответствующим макросом). Кроме того, textcolor() позволяет делать цвет мигающим (blink). Код этого аргумента – 128. Для того, чтобы сделать цвет мигающим надо применить побитовую операцию "OR" к цвету и величине BLINK. Например, для вывода текста красным мигающим цветом надо применить функцию

textcolor(RED|BLIK).

Изменение цвета текста оказывает действие только на вновь выводимый текст и не оказывает никакого действия на ранее введенный текст.

Для установки цвета фона используется функция

void textbackground(int color);

Здесь аргумент color имеет те же значения, что и в функции textcolor().

Функция clrscr() не только очищает активное окно, но и заполняет его цветом, заданным в функции background().

Пример 9.2.
Использование функций работы с текстом.

```
# include <conio.h>
main(void)
{
register int ctext< cback;
textmode(C80); /* 40 символов, 25 строк, цветовой */
for(ctext = BLUE; ctext<=WHITE; ctext++)
 {
 for(cback = BLACK; cback<=LIGHTGRAY; cback++)
 {
 textcolor(ctext);
 textbackground(cback);
 cprintf("ТЕКСТ");
 }
cprintf("\n");
}
textcolor(WHITE|BLINK);
textbackground(BLACK);
cprintf("КОНЕЦ ТЕКСТ");
```

```
textmode(LASTMODE);
RETURN(0);
}
```

Организация видеопамяти при работе с текстом следующая. Каждому символу соответствует 2 байта. В одном из них хранится код символа, в другом – атрибут символа. Байт атрибута имеет следующий вид:

7	6	5	4	3	2	1	0
В	Ф	Ф	Ф	С	С	С	С

где СССС-4 битовый двоичный код символа; ФФФ – 3-битовый двоичный код фона; В – признак мигания(1 – мигание включено,0-выключено).

Так как для цвета фона выделено 3 бита, то цветов фона может быть только 8, а для цветов символа, задаваемого четырьмя битами – 16 цветов.

Более того, любой цвет формируется смешением красного (R-red), зеленого (G-green) и голубого (B-blue) цветов. В байте атрибута мы можем указать, какой бит за какой цвет отвечает:

7	6	5	4	3	2	1	0
B	R	G	B	C	R	G	B

При этом третий бит отвечает за яркость цвета символа. Если мы будем рассматривать последние 3 бита, то комбинация 0001 будет соответствовать цвету BLUE (1), в то время как 1001 будет соответствовать цвету LIGHTBLUE (9). Соответственно 0000– черный цвет (0). 1111– белый цвет (15).

Функция, которая позволяет задавать атрибут, имеет прототип

```
void textattr(int newattr);
```

Например, задать атрибут, соответствующий красному символу на черном фоне, можно следующими способами:

```
textattr(( BLACK<<4)+ RED+ BLINK );
```

или textattr(0x80);

Двоичная запись шестнадцатеричного числа 0x80 имеет вид 10000100. Сравните его с битами атрибута.

В библиотеке предусмотрены функции, позволяющие сохранять, восстанавливать и перемещать содержимое части экрана. Этими функциями являются соответственно

int gettext(int left,int top,int right, int bottom, void*destin);

int puttext(int left,int top,int right, int bottom, void*sourse);

int movetext(int left,int top,int right, int bottom, int destleft, int desttop);

Область экрана задается координатами левого верхнего и правого нижнего углов, область памяти, в которую (или из которой) переносится часть видеопамяти, задается указателями destin и sourse. В функции movetext() два последних аргумента указывают на координаты верхнего левого угла нового расположения экрана. Так как для каждого символа отводится два байта памяти, то при сохранении участка экрана надо предусмотреть буфер соответствующего размера.

Все прототипы функций хранятся в заголовочном файле CONIO.H

СПИСОК РЕКОМЕНДОВАННЫХ ИСТОЧНИКОВ

1. *Керниган Б., Ритчи Д., Фьюэр А.* Язык программирования Си. - М.: Финансы и статистика , 2000.
2. *Березин Б. И., Березин С. Б.* Начальный курс С и С++. – М.: ДИАЛОГ- МИФИ, 1999
3. *Культин Н. Б.* С/С++ в задачах и примерах – СПб.: БХВ- Петербург, 2001
4. *Крячков А. В., Сухинина И. В., Томшин В.К.* Программирование на С и С++. Практикум: Учеб. пособие для вузов. / Под ред. В.К.Томшина – 2-е изд., исправ.– М.: Горячая линия – Телеком, 2000

ОГЛАВЛЕНИЕ

Учебное издание

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ Си

Учебное пособие

Шишкин Анатолий Дмитриевич

Редактор И.Г. Максимова

ЛР № 020309 от 30.12.96.

Подписано в печать 25.04.03. Формат 60x90 1/16. Гарнитура Times New Roman.
Бумага офсетная. Печать офсетная. Печ.л. 6,7. Тираж 300 экз. Заказ № 19
РГГМУ, 195196, Санкт-Петербург, Малоохтинский пр., 98.
ЗАО «Лека», 195112, Санкт-Петербург, Малоохтинский пр., 68.