Куцый О. Я.
Кораблин А. И.

ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

Москва, 2017

ОГЛАВЛЕНИЕ

1. Основы языка Си

Алфавит языка

При написании текста программы можно использовать только определенный набор символов, в состав которого входят:

26 больших латинских букв

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

26 малых латинских букв

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

10 цифр

```
0 1 2 3 4 5 6 7 8 9
```

29 знаков препинания и спецсимволов

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

3 пробельных символа: пробел, табуляция, символ новой строки (клавиша Enter)

Компилятор языка Си различает большие и малые латинские буквы в словах, поэтому слова **add**, **Add** и **ADD** в тексте программе будут иметь разное назначение.

Группы символов

1. Английские буквы, цифры и символ подчеркивания используется для составления слов в тексте программы, называемых идентификаторами.
   Все идентификаторы можно разделить на три группы:
   - Ключевые слова - это идентификаторы, которые имеют определенный смысл в языке и должны использоваться строго по своему назначению (**char, int, sizeof, if, for, static,** …).
   - Зарезервированные слова - это идентификаторы, которые имеют определенный смысл в языке, назначение которых можно изменить (**printf, scanf, EOF, RAND_MAX,** …).
   - Собственные идентификаторы – это идентификаторы, которые программист вводит в программу для своих целей, руководствуясь следующим простым правилом: идентификатор может иметь любую длину и начинаться с буквы или символа подчеркивания, при этом он не должен совпадать с другими идентификаторами и ключевыми словами.

   Пример собственных идентификаторов: **a, txt10, str, _line, _10, _.**
2. Цифры, некоторые английские буквы и спецсимволы используются для обозначения чисел в тексте программы.
3. Пробельные символы используются для разделения идентификаторов и форматирования текста программы.
4. Знаки пунктуации и спецсимволы имеют разнообразное применение и будут по отдельности рассматриваться далее.

Лексемы

Компилятор переводит текст программы на машинный язык, разбивая его на отдельные фрагменты, называемые лексемами.

Лексема - это максимальный фрагмент текста программы, который имеет определенный смысл для компилятора.

Границы лексем определяются пробельными символами или другими лексемами. Числа, идентификаторы, скобки, знаки операций и другие спецсимволы являются лексемами. Между лексемами можно ставить любое количество любых пробельных символов.

Комментарии

Комментарии – это фрагменты текста программы, которые компилятором не рассматриваются. Поэтому их можно составлять, используя любой набор символов, имеющихся в компьютере. Комментарии предназначены для сопровождения текста программы пояснительными записями, а иногда для исключения из компиляции некоторых фрагментов текста программы с целью отладки.

Начало комментария в Си обозначается комбинацией символов **/*** а конец ***/**. Текст комментария может занимать несколько строк. Вложение комментариев не допускается.

Кроме многострочных комментариев, в тексте программы можно использовать однострочный комментарий, начало которого обозначается комбинацией символов **//**, а окончание определяется концом строки.

Примеры комментариев:

```
/* Этокомментарий*/
/* Это
        Многострочный
        комментарий*/
// Это однострочный комментарий (C11)
```

Типы и размеры данных

Любая программа содержит две основные части: информацию, которую она обрабатывает и набор команд, при выполнении которых происходит обработка этой информации. Информацию, с которой работает программа, называют данными, и в программе она находиться в виде чисел, которые хранятся в ячейках оперативной памяти. Виды ячеек, которые можно использовать в программе для хранения чисел, называют типами данных.

Тип данных определяет размер ячейки, и внутреннее представление числа, хранящегося в ней, которое может быть целым со знаком, целым без знака и вещественным (числом
с дробной частью).

В языке Си имеется четыре основных типа данных, два целых и два вещественных. Для обозначения целых типов используются следующие ключевые слова:

char – целое число со знаком размером 1 байт;

int – целое число со знаком размером 2 байта (для 8, 16-разрядных ЭВМ) или 4 байта (для 32, 64-разрядных ЭВМ);

Для обозначения вещественных типов:

float – число с плавающей точкой одинарной точности размером 4 байта;

double – число с плавающей точкой двойной точности размером 8 байт.

Под точностью понимается количество значимых разрядов - первых цифр в представлении числа, которым можно верить. Тип **float** дает 7-8 значимых разрядов, а тип **double** 15-16.

Кроме основных типов в языке имеется четыре модификатора, с помощью которых можно изменить представление числа в ячейке или ее размер.

Модификаторы для изменения размера ячейки:

short – короткое целое размером 2 байта. Может использоваться только с типом **int**.

long – увеличивает размер ячейки. Может использоваться с типами **int** и **double**. Для получения 64-разрядных целых чисел этот модификатор необходимо указать дважды (C11).

Модификаторы для изменения представления целых чисел:

signed – целое число со знаком.

unsigned–целое число без знака.

Модификаторы используются в сочетании с основными типами по следующей форме:

[модификатор_представления] [модификатор_размера] тип

В квадратных скобках показаны необязательные части формы. Тип **int** можно опустить, если имеется, хотя бы один модификатор. Стандарт С99 отменил это правило и, если компилятор не совместим с более ранними стандартами, тип **int** пропускать нельзя.

В таблице показаны возможные сочетания модификаторов с основными типами данных и их наименование.

Полное задание	Краткое задание	Наименование
signed char	char	Символ со знаком
signed int	int	Целое со знаком
signed short int	short	Короткое целое со знаком
signed long int	long	Длинное целое со знаком
signed long long int	long long	-
unsigned char	-	Символ без знака
unsigned int	unsigned	Целое без знака
unsigned short int	unsigned short	Короткое целое без знака
unsigned long int	unsigned long	Длинное целое без знака
unsigned long long int	unsigned long long	
float	-	Число с плавающей точкой
long float	double	Число с плавающей точкой двойной точности
long double	-	-

В следующей таблице показаны размеры ячеек и диапазон чисел, которые могут быть помещены в ячейки соответствующих типов.

Тип	Размер	Диапазон значений
char	1 байт	-128 до 127
int	для 16-ти разрядного процессора - 2 байта для 32-х разрядного процессора - 4 байта	-32 768 до 32 767 -2 147 483 648 до 2 147 483 647
short	2 байта	-32 768 до 32 767
long	4 байта	-2 147 483 648 до 2 147 483 647
long long	8 байт	-9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
unsigned char	1 байт	0 до 255
unsigned	зависит от реализации для 16-ти разрядного процессора - 2 байта для 32-х разрядного процессора - 4 байта	 0 до 65 535 0 до 4 294 967 295
unsigned short	2 байта	0 до 65 535
unsigned long	4 байта	0 до 4 294 967 295
unsigned long long	8 байт	0 до 18 446 744 073 709 551 615
float	4 байта	3.4E +/- 38 (7-8 зн. разрядов)
double	8 байт	1.7E +/- 308 (15-16 зн. разрядов)
long double	16 байт (C11)	1.2E +/- 4932 (33-36 зн. разрядов)

Тип данных стараются выбирать таким образом, чтобы его размер был минимальным и все числа, которые планируется помещать в ячейку, входили в диапазон выбранного типа.

В С11 были добавлены два новых типа. Логический тип _Bool, который ведёт себя также, как и обычный встроенный тип, за одним исключением: любое ненулевое

присваивание _Bool хранится как единица. При подключении заголовочного файла <stdbool.h> дополнительно будет определён псевдоним bool и макросы true(истина), false(ложь). Для поддержки комплексных чисел и вычислений с ними в C11 был добавлен тип _Complex. В зависимости от требуемой точности расчётов возможно создание переменных трёх типов: float _Complex, double _Complex и long double _Complex. В языке C11 поддерживаются стандартные арифметические операции +, -, *, / для комплексных чисел, а при подключении заголовочного файла <complex.h> станут доступны и другие математические операции с комплексными числами.

Константы

Данные в программе могут находиться в виде констант. Константы – это неизменяемые в программе величины.

В языке Си различают четыре вида констант: целые константы, константы с плавающей точкой, константы-символы и строковые литералы.

Целая константа - это десятичное, восьмеричное или шестнадцатеричное представление целого числа, причем восьмеричное представление должно начинаться с цифры 0, а
шестнадцатеричное представление с 0x или 0X.

Примеры целых констант:

Десятичные константы	Восьмеричные константы	Шестнадцатеричные константы
10	**012**	**0xa или 0xA**
132	**0204**	**0x84**
32179	**076663**	**0x7dB3 или 0x7DB3**

Компилятор может отнести целую константу к типам **int, long int, unsigned int, long long int, unsigned long long int** в зависимости от того, в диапазон какого минимального по размеру типа она войдет.

В конце любой константы одним или двумя суффиксами можно уточнить ее размер и представление.

Модификатор	Суффикс
long	**l или L**
unsigned	**u или U**
long long	**ll или LL**

Пример задания целых констант с суффиксами:

Десятичные константы	Восьмеричные константы	Шестнадцатеричные константы
10L или 10l	**012L или 012l**	**0xaL или 0xAL**
79U или 79u	**0115u или 0115U**	**0x4fu или 0x4FU**
123LL или 123ll	**045LL или 045ll**	**0x4A5LL или 0x4A5ll**
12LU или 12lu	**031LU или 031lu**	**0x3ELU или 0x3Elu**

Порядок следования суффиксов в константе не имеет значения.

Константы с плавающей точкой - это представление вещественного числа. Вещественные числа представляются всегда в десятичной системе счисления. В качестве разделителя дробной части используется точка. Целая или дробная части константы могут быть опущены, но не обе сразу.

Пример: **15.75, 5., .0025, -.137**.

Существует экспоненциальная форма представления вещественных констант. Для задания экспоненты в представлении константы используется латинская буква E или e, после которой должно быть указано значение степени. Целая или дробная части в задании константы могут быть опущены, но не обе сразу. Десятичная точка может быть опущена только тогда, когда задана экспонента.

Пример: **1.575E1, 5E0, -.25e-2, 1575e-2.**

Пробельные символы в задании константы не допускаются.

Все константы с плавающей точкой имеют тип **double**, который можно изменить на **float,** добавив суффикс **F** или **f** в конец задания константы или **long double**, добавив суффикс **L** или **l**.

Пример констант с суффиксами: **5.23F, 6E-4f, 4.15L**.

Константа-символ - это один символ, заключенный в одинарные кавычки. Величина константы равна значению номера символа согласно текущей однобайтной кодировке, заключенного в одинарные кавычки и имеет тип **int**.

Пример констант-символов: **'a', 'я', 'F', '?'**.

В одинарных кавычках можно заключить любой символ, имеющийся в ЭВМ или любой ESC-символ.

ESC-символ - это последовательность символов, начинающаяся с обратной косой черты ****. ESC-символы используются в символьных константах и строковых литералах для задания символов, которые нельзя ввести с клавиатуры или которые имеют специальное назначение в языке Си.

В таблице приведен список ESC-последовательностей языка Си.

ESC-символ	Наименование (действие при выводе на экран)
\n	Новая строка (переводит текстовый курсор на следующую строку)
\t	Горизонтальная табуляция (переводит текстовый курсор в следующую позицию табуляции)
\b	Забой (переводит текстовый курсор на один знак влево)
\r	Возврат каретки (переводит текстовый курсор в начало текущей строки)
\a	Сигнал (генерирует звуковой сигнал)
\v	Вертикальная табуляция
\f	Новая страница
\'	Одиночная кавычка
\"	Двойная кавычка
\\	Обратная косая черта
\ddd	Номер символа в восьмеричном представлении. ddd – разряды восьмеричного числа.
\xdd	Номер символа в шестнадцатеричном представлении. dd – разряды шестнадцатеричного числа.

Примеры символьных констант, заданных с использованием ESC-символов: **'\n', '\x1B', '\'', '\\', '\145', '\x2F'**.

Если в константе-символе необходимо задать символ одинарной кавычки или обратной косой черты, то они должны быть заданы только через ESC-последовательность.

Строковый литерал – это последовательность любых символов, в том числе и ESC-символов, заключенная в двойные кавычки.

Пример: **"Hello World", "Иванов\n\tИван\n\t\tИванович\a".**

Если в строковом литерале необходимо задать символ двойной кавычки или обратной косой черты, то они должны быть заданы только через ESC-последовательность.

Пример: **"Hello \"World\"".**

Длинные строки-литералы можно переносить на следующую строку текстового файла, указав в конце строки символ косой черты, как показано в примере ниже:

```
"Это пример переноса
\↵ длинной строки".
```

С точки зрения компилятора, две строки, показанные выше, являются одной строкой текста программы. Такой перенос допускается для любой лексемы.

Каждый символ в строке-литерале будет занимать один байт памяти (тип **char**).

Для получения символьных и строчных констант, использующих универсальную кодировку, перед константой необходимо поставить префикс **L**.

Пример таких констант: **L'a', L'\n', L"Hello World"**. Пробелы между буквой **L** и константой не допускаются.

Каждый символ в строке-литерале с префиксом **L** будет занимать два байта памяти (тип **unsigned short** или **wchar_t**). Такие символы называют широкими символами.

Переменные

Переменные – это поименованные ячейки оперативной памяти, которые предназначены для хранения данных в программе. Переменные позволяют производить считывание и запись информации в связанные с ними ячейки.

Для того, чтобы в программе получить переменные, их необходимо продекларировать. Это можно сделать в тексте программы после открывающейся фигурной скобки предложением, составленным по следующей форме:

```
тип_данных список_имен_переменных;
```

список_имен_переменных – одно или несколько имен переменных, перечисленных через запятую.

Пример:

```
int abc;
double x,y;
```

В переменных, декларируемых таким образом, начальное значение заранее не определено (мусор). Для того чтобы задать переменной нужное начальное значение, необходимо после имени переменной в строке декларации поставить знак равно и указать это значение.

Пример:

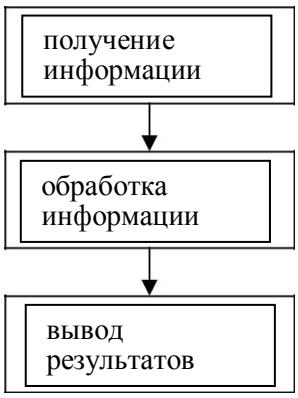
```
unsigned char c ='a';
long m=10,k=2,l={0};
```

Задачи

1. Запишите предложение декларации переменной x, в которой будут храниться числа в диапазоне от 10 до 60000.
2. Запишите предложение декларации переменной, в которой будут храниться числа 0 или 1 и проинициализируйте ее 0.
3. Запишите предложение декларации переменной n, которая будет использоваться в программе для подсчета количества и принимать максимальное значение 15000.
4. Запишите предложение декларации переменной x и y, в которых будут храниться числа с плавающей точкой двойной точности и проинициализируйте их 0.
5. Запишите предложение декларации переменной, в которой будут храниться коды символов и проинициализируйте ее кодом символа A.
6. В строке литерале, показанной ниже, вместо многоточия, нарисуйте символ с номером 186, который нельзя ввести с клавиатуры.
   "Фамилия ... Имя ... Отчество"
7. Сколько байт будут занимать следующие константы: 5, 2.5, 43L, 123UL, 12LL, 3.45LL, 5.2F.

2. Написание простых программ.

Программа – это набор команд, составленных в определенной последовательности, которые выполняет процессор для решения некоторой задачи, связанной с обработкой информации.



На рисунке представлена структура простой программы, которая отражает порядок работы команд в программе и ее реализацию. Как видно из рисунка, программа сначала должна выполнить команды, предназначенные для получения исходной информации, которую в дальнейшем она будет обрабатывать. Исходная информация помещается программой в переменные - ячейки оперативной памяти. Затем программа должна приступить к обработке полученной информации. Перед завершением работы программа должна, каким-то образом распорядиться полученными результатами, иначе они исчезнут из оперативной памяти так же, как и сама программа. Поэтому завершающим набором команд должны быть команды, выполняющие вывод полученных результатов.

Любая программа, написанная на языке Си, должна иметь следующий минимальный фрагмент текста, который называют функцией **main**:

```
int main(void){
   /*Декларация
   переменных*/ /*Команды
   */
}
```

Этот фрагмент определяет главный набор команд в программе, с которого она начнет выполняться. Команды должны быть указаны между фигурными скобками. После открывающей фигурной скобки можно сделать декларацию переменных, необходимых для реализации программы.

Организация обработки данных

Для осуществления обработки информации, в программе необходимо записать предложения, содержащие наборы действий над нужными данными. Такие предложения на языке Си имеют следующую форму:

```
имя_переменной = выражение;
```

выражение – любой набор констант, переменных и функций, связанных знаками операций.

Примеры:

```
a = 10; // Положить в переменную a число 10
a = a + 5; // Увеличить переменную a на 5
c = a + b; /* Сложить содержимое переменных a и b.
              Результат занести в переменную c */
s = 3.14 * d * d / 4; /* Вычислить площадь окружности,
              диаметр которой содержится в переменной d.
              Результат за нести в переменную s */
x1 = (-b + sqrt(d))/(2*a); /* Вычислить корень
              квадратного уравнения */
```

Знак равно обозначает действие, связанное с записью результата в переменную, указанную слева от знака равно. Называют это действие присвоением значения.

В конце каждого простого предложения в языке Си должна стоять точка с запятой, как признак окончания предложения.

Организация ввода/вывода

Для того чтобы реализовать в программе ввод информации и вывод результатов, необходимо воспользоваться стандартными готовыми решениями, имеющимися в языке Си, которые называют библиотечными функциями.

getchar() – это функция, которая возвращает код символа, введенного с клавиатуры. Получение символа программой произойдет после его ввода и нажатия клавиши Enter, при этом будет считан первый символ, если было введено несколько.

Пример использования:

```
int ch;
…
ch = getchar();// Получить код символа и положить
               // его в переменную ch
getchar();     // Остановка программы до нажатия клавиши
//Enter.
```

putchar(код_символа) – функция выводит символ, код которого указан в скобках, на экран. Символ выводится в текущее положение текстового курсора. Курсор при этом перемещается на один символ вправо.

Пример использования:

```
putchar('a');   // Код символа задан константой
putchar(ch);    // Код символа находится в переменной ch
putchar(ch+32); // Код символа вычисляется с помощью
//выражения
```

В последнем примере будет происходить вывод на экран малой латинской буквы, если в переменной **ch** будет находиться код большой латинской буквы.

Так устроена ASCII таблица, которая задает соответствие между начертанием символа и его кода.

puts(строка_литерал) – функция выводит строку символов на экран. После вывода строки, функция автоматически переводит текстовый курсор в начало следующей строки на экране.

Пример использования:

```
  puts("Hello World") ;
```

printf(форматная_строка [, список_информации]) – функция осуществляет вывод на экран числовых и символьных данных в разных форматах.

форматная_строка - строка-литерал. В форматной строке может быть любой набор символов, который без изменения будет выведен на экран, за исключением символа %. Этот символ интерпретируется функцией, как начало задания формата вывода данных. С помощью формата программист указывает функции место вывода информации, а также, в каком виде, и какого типа данные выводить на экран.

список_информации - если в форматной строке заданы форматы вывода данных, то после форматной строки должен идти список переменных или выражений, перечисленных через запятую и их количество, а также порядок следования, должны совпадать с заданными форматами в форматной строке.

Простые форматы для разных типов данных.

Формат вывода	Описание вывода	Соответствие типам данных
%c	символ	char
%d или %i	целое десятичное число со знаком	int
%u	целое десятичное число без знака	unsigned
%o	восьмеричное число	unsigned
%x или %X	шестнадцатеричное число	unsigned
%ld, %lu	форматы для длинных целых	long, unsigned long

%lo, %lx	представление длинных целых в разных системах счисления	long, unsigned long
%f	вещественное число без экспоненты	float
%lf	-||-	double
%Lf	-||-	long double
%e или %E, %le или %lE, %Le или %LE	вещественное число с экспонентой	float, double и long double соответственно
%g, %lg, %Lg	%f или %e, что короче	--||--
%G, %lG, %LG	%f или %E, что короче	--||--
%s	строка символов	
%%	знак процента	

Примечание: в таблице дано строгое соответствие формата вывода, типам данных. Однако форматы для целых чисел могут допускать другие целые типы данных, не указанные в таблице.

Ниже показаны примеры использования функции **printf**.

```
//вывод строки
 printf("Hello World\n");
//вывод значения одной переменной типа float
printf ("Сторона квадрата=%f см.\n",a);
//вывод значения двух переменных s и p типа double
printf ("Площадь=%lf см.кв., периметр=%lf см.\n", s, p);
```

Более сложные форматы конструируются по следующей форме:

%[флаги][ширина][.точность][модификатор]тип

В квадратных скобках показаны необязательные части формата.

тип – задает тип выводимых данных и их вид на экране. На этом месте может стоять один из следующих символов:

Символ	Задаваемый тип данных
c	char
d, i	int
u, o, x, X	unsigned
f, e, E, g, G	float
s	строка

модификатор - символ, уточняющий тип данных. На этом месте может стоять один из следующих символов:

Символ	Задаваемый тип данных
h	short
l	long, double
L	long double, long long

.точность - число, которое задает количество знаков после десятичной точки при выводе вещественных чисел. При указании точности, функция **printf** будет осуществлять округление выводимых чисел до указанного количества знаков по правилам математики.

ширина - число, которое задает размер в символах поля вывода информации. Отсчитывается от положения текстового курсора, когда функция **printf** начнет выводить информацию согласно формату. Используется ширина поля для стабилизации вывода, что бы функция всегда выводила нужное количество символов, например, в ячейки таблицы.

флаги - управляют форматом вывода и представляются следующими символами:

Символ	Действие
-	выровнять информацию относительно поля вывода влево (по умолчанию информация выравнивается вправо)
+	всегда выводить число с указанием знака
0	вывести число с лидирующими нулями (нужно указать ширину поля)
#	вывести число с указанием системы счисления (0 – для восьмеричной, 0x или 0X для шестнадцатеричной)

Примеры использования более сложных форматов:

```
//вывод числа с округлением его до двух знаков
printf(" Сторона квадрата равна %.2f см.\n", a);
// вывод числа, которое содержится в переменной x типа
int, в разных системах счисления
printf("x = %d = %#o = %#x\n", x, x, x);
// вывод числа, которое содержится в переменной x типа int,
   в ячейку таблицы с указанием размера ячейки и
   выравнивания влево
printf("%-8d", x);
```

scanf(форматная_строка, список_адресов_переменных) – функция осуществляет получение числовых и символьных данных с клавиатуры. **форматная_строка** - строка литерал, содержащая форматы ввода данных. Форматы функции **scanf** совпадают с форматами функции **printf**, но требуют строгого соответствия типам переменных.

Все символы, которые не относятся к форматам в форматной строке, будут восприниматься функцией как указание пропустить такие символы при получении информации.

список_адресов_переменных - имена переменных, перечисленных через запятую, перед каждым именем переменной должен стоять знак **&,** которым обозначают операцию получения адреса переменной.

Примеры ввода данных функцией **scanf**:

```
// Получение целого числа и запись его в переменную типа int
scanf("%d", &m);
// Чтение целого числа в шестнадцатеричном формате
scanf("%x", &h);
// Получение двух чисел: целого и вещественного
scanf("%d%f", &i, &p);
// Получение даты в общепринятом формате: ДД.ММ.ГГГГ.
// Точки функция будет пропускать
scanf("%d.%d.%d", &day, &mon, &year);
```

Функция возвращает число, которое определяет количество полученных данных. Это число должно совпадать с количеством форматов, указанных в форматной строке, в противном случае пользователь неправильно ввел данные.

В примере, показанном ниже, переменная **n** должна получить число 3 после ввода даты пользователем.

```
n = scanf("%d.%d.%d", &day, &mon, &year);
```

Функции **getchar** и **scanf** не всегда могут срабатывать, особенно при неправильном вводе информации пользователем, поскольку такая информация не будет принята программой. Поэтому перед вызовом или после вызова этих функций рекомендуется вызывать функцию **fflush**, которая удалит оставшуюся информацию.

Пример:

```
   ch =getchar();
   fflush(stdin);
```

Использование библиотечных функций

Программа расчета стороны квадрата по известной площади. Сторона квадрата рассчитывается по формуле: a = √s, где s - площадь квадрата. Для расчета корня квадратного необходимо воспользоваться стандартной математической функцией **sqrt**, которая работает с числами типа **double**. Предложение, рассчитывающее сторону квадрата может выглядеть следующим образом: **a = sqrt(s);**

Ниже показан текст этой программы. Переменные в программе выбраны типа **double** т.к. функция **sqrt** работает с этим типом вещественных чисел.

```
main()
{
   double a, s;
   puts("Расчет стороны квадрата.") ;
   printf("Введите площадь квадрата: ") ; // Выдать запрос
   scanf("%lf", &s) ; // и получить площадь квадрата
   a = sqrt( s ) ; // Посчитать сторону квадрата
// Вывести результат
   printf("Сторона квадрата равна: %lg\n", a) ;
}
```

В таком виде эта программа работать не будет. Если присмотреться к сообщениям, которые будет выдавать компилятор, то можно увидеть, что он будет предупреждать о возможных ошибках, при организации работы используемых в программе функций. В этой программе использована математическая функция **sqrt**, результат работы которой закладывается в переменную **a**. Результат у этой функции имеет тип **double**, но компилятор об этом не знает, и будет использовать его как тип **int**, который в языке Си часто используется как тип по умолчанию. Так как размеры и представления информации у этих типов разные, то компилятору не удастся правильно организовать работу этой программы до тех пор, пока он не будет ознакомлен с функцией **sqrt**.

Описание библиотечных функций находится в заголовочных файлах, которые имеют расширение **.h**. Поскольку функций много, то все они разбиты на группы, и описание каждой группы находиться в отдельном файле. Описание функций ввода-вывода, которые рассматривались выше, находиться в файле **stdio.h**, а описание математических функций в файле **math.h**.

Для предоставления описания компилятору, в начале текстового файла программы необходимо ввести следующую строку:

#include <math.h> - для включения описаний математических функций. После этого программа будет работать правильно.

Для устранения предупреждений по поводу функций ввода-вывода, в начало текста необходимо добавить вторую строку:

#include <stdio.h>

Имя файла, в котором находиться описание любой функции, можно узнать из справочника по языку Си или запросить справку (F1) в среде программирования.

Задачи

1. Запишите предложение, в котором переменной **x**, присваивается значение 2,45.
2. Запишите предложение, в котором значение переменной **y** уменьшается на 10.
3. Запишите предложение для вычисления среднего арифметического переменных **a1** и **a2**.
4. Запишите предложение, в котором вычисляется значение переменной **y** по следующей формуле: $y = -3{,}67x^3 - 0{,}45x^2 + 12{,}8$.

5. Напишите предложения, которые выведут на экран Вашу фамилию, имя и отчество. Каждое слово должно быть выведено в отдельной строке.
6. Напишите предложение, с помощью которого текстовый курсор на экране будет переведен в начало следующей строки.
7. Допустим, на диске C: в каталоге NC содержится файл README.TXT. Напишите предложение, которое выведет на экран путь к этому файлу.
8. Напишите предложение, с помощью которого на экран будет выведено значение переменной **`count`** типа **`unsigned`**.
9. Напишите предложение, с помощью которого на экран будет выведен символ, код которого содержится в переменной **`ch`**.
10. Напишите предложение, с помощью которого на экран будут выведены значения переменных **`L`** и **`H`** типа **`float`**, в которых содержится длина и высота некоторого объекта. Числа должны быть выведены с точностью до второго знака после запятой, и каждое значение необходимо прокомментировать на экране.
11. Напишите предложение, с помощью которого программа получит от пользователя один символ и положит его код в переменную **`ch`**.
12. Напишите предложения, с помощью которых можно гарантированно остановить работу программы до нажатия клавиши Enter.
13. Программе необходимо получить от пользователя два целых числа и положить их в переменные **`x`** и **`y`**. Напишите фрагмент, с помощью которого это можно сделать.
14. Написать программу для вычисления длины и площади окружности. Для этого воспользоваться формулами:
    - l = 2π r (где π - 3,14, r - радиус окружности) длина окружности
    - $s = \pi r^2$ площадь окружности,
15. Написать программу для вычисления периметра и площади прямоугольника. Для этого воспользоваться формулами:
    - p = 2(a+b) периметр прямоугольника,
    - s = abплощадь прямоугольника,
    - где a и b стороны прямоугольника.
16. Написать программу вычисления объема цилиндра. Для этого используется формула: $v = \pi r^2 h$, где π - 3,14, r - радиус цилиндра, h - высота цилиндра.
17. Написать программу вычисления стоимости покупки, состоящей из тетрадей по цене 5,5 р. и ручек по цене 9,99р.
18. Написать программу расчета стоимости взвешиваемого продукта. Программа должна получить от пользователя стоимость одного килограмма и вес продукта.
19. Написать программу расчета стоимости поездки на автомобиле на дачу. В расчетах учитывать расстояние до дачи, расход бензина на 100 км и стоимость одного литра бензина.
20. Написать программу определения величины дохода по вкладу. Программа должна получить от пользователя размер вклада, срок вклада в днях и годовую процентную ставку. Результат отобразить с точностью до второго знака.
21. Написать программу расчета корней квадратного уравнения ($ax^2 + bx + c = 0$). Для этого необходимо получить коэффициенты a, b и c и вычислить два корня по формуле:

$$x1/2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

3. Операции в Си

Операция - это символ или комбинация символов, которые сообщают компилятору о необходимости произвести определенные действия над операндами.

Операндом называют число, которое участвует в операции. Каждая операция предполагает наличие определенного числа операндов и имеет результат - это число, которое может быть присвоено переменной или использоваться в качестве операнда в другой операции.

В языке Си имеются операции, которые требуют задания одного, двух и трех операндов. В качестве операндов могут быть константы, переменные, результаты других операций и результаты вызовов функций. Некоторые операции требуют задания строго определенных операндов.

Каждая операция имеет определенный приоритет. Приоритет определяет порядок выполнения операций в выражении.

Здесь и далее под выражением понимается набор констант, переменных и результатов вызовов функций, связанных знаками операций.

Арифметические операции

Однооперандные операции.

- арифметическое отрицание. Операнд должен быть указан справа от операции. Используется для всех числовых типов данных. Результат операции – число с обратным знаком.

Пример:

```
// Переменной a присвоить число со знаком, обратным числу в
b a = -b;
// К значению a прибавить значение b с обратным знаком
c = a + -b;
// Вывести на экран с обратным знаком
 printf("Число, обратное значению x %d\n",-x);
```

+ унарный плюс – получение числа с тем же знаком (не используется).

++ инкремент, или увеличение переменной на единицу.

-- декремент, или уменьшение переменной на единицу.

Операнд может быть указан как справа, так и слева от операции и должен быть только переменной. Если операнд указан справа от операции (префиксное использование), то приоритет этой операции будет самым высоким (операция будет выполняться первой), а если слева (постфиксное использование), то самым низким в выражении.

Применяются эти операции для сокращения записи выражений. Например, для увеличения значения переменной **count** на единицу можно применить выражение

```
   count = count + 1;
```

а можно и короче записать, используя операцию **++**

```
count++;
// или
++count;
```

В некоторых случаях можно достичь большего сокращения. Например, перед вычислением выражения **y = a * x + b** необходимо уменьшить значение переменной **x** на единицу. Для этого можно записать следующие предложения:

```
   x = x - 1;
   y = a * x + b;
```

а можно и сократить одну строку

```
   y = a * --x + b;
```

Если сначала необходимо вычислить выражение, а потом изменить значение переменной, то необходимо поставить операцию справа от операнда. Например, предложения

```

```

```
y = a * x +b;
x = x - 1;
```

можно привести к следующему виду

```
y = a * x-- + b;
```

В одном выражении не рекомендуется применять эти операции более одного раза к одной и той же переменной, поскольку результат всего выражения в этом случае заранее не определен и зависит от реализации компилятора и его настроек.

Двухоперандные операции.

+ сложение двух операндов.

- вычитание второго операнда из первого. ***** умножение двух операндов.

/ деление первого операнда на второй. **%** остаток от деления целых чисел.

Операции сложения, вычитания и умножения используются так же, как и в арифметике и никаких особенностей в программировании не имеют.

У операции деления есть некоторые особенности.

1. При делении целых чисел получается целый результат. Например, результатом выражения **7/2** будет число 3, а не 3,5. Если хотя бы одно из чисел будет вещественным, то результат будет вещественным. Например, результатом выражения **7.0/2** будет число 3,5.
2. Делить на ноль нельзя. Если программа выполнит эту операцию, она буде выгружена системой за неправильные действия и прекратит свою работу. Это касается и операции получения остатка от деления.

Все операции, кроме операции **%**, можно использовать со всеми рассмотренными типами данных.

Операцию получения остатка от деления можно применять только при работе с целыми типами. Результатом выражения **7%4** будет число 3. Если необходимо разделить 7 яблок между 4 людьми поровну без разрезания яблок, то останется 3 яблока, которые на 4 человека не делятся нацело.

Эта операция в программировании часто применяется для определения кратности. Например, известно, что номер високосного года кратный четверке - должен разделиться на 4. Поэтому, если в переменной **year** содержится номер года, то для определения, является ли этот год високосным, необходимо применить выражение **year % 4** и если результат будет равным нулю - то год високосный.

Можно применять эту операцию и по прямому назначению. Например, в языке Си имеется функция **time**, которая возвращает системное время по Гринвичу - количество секунд, прошедшее с 1 января 1970 г. Для того, чтобы получить текущие секунды, минуты и часы необходимо воспользоваться операцией **%**. В примере ниже показано, как из общего количества секунд выделить текущие часы, минуты и секунды в программе.

Пример:

```
t = time (0);   // Получить общее количество секунд
s = t % 60;     // Выделить текущие секунды
m = t/60;       // Определить общее количество минут
m = m%60  ;     // Выделить текущие минуты
h = t/3600;     // Определить общее количество часов

h = h%24  ;     // Выделить текучий час
```

Иногда эту операцию используют для ограничения диапазона получаемых чисел, поскольку остаток от деления не может быть больше и даже равен знаменателю. Например, в языке Си имеется функция **rand**, результатом работы которой является псевдослучайное число в диапазоне от **0** до **RAND_MAX**. Значение **RAND_MAX** зависит от реализации функции, но должно быть не меньше 32767. Для того, чтобы ограничить

диапазон получаемых чисел, можно получить остаток от деления результата функции **rand** на значение, на единицу больше требуемого верхнего диапазона. Например,

```
// получить псевдослучайное число в диапазоне от 0 до
19 x = rand()%20 ;
```

Операции отношений и логические операции

Логические операции используются для реализации в программе анализа значений переменных и результатов выражений.

Результатом логических операций в языке Си всегда будет целое число: 0 или 1. Других результатов при применении логических операций нет. Ноль показывает, что логическое выражение не верно, или говорят, что оно ложно. Если результат равен единице, то говорят, что выражение истинно.

Логические операции могут применяться для всех, ранее рассмотренных типов данных.

Операции отношения.

< меньше,
> больше,
<= меньше или равно,
>= больше или равно,
== равно,
!= не равно.

Все операции отношения двухоперандные. Примеры простых логических выражений показаны ниже:

```
интересует значение переменной, если оно больше 10
   a > 10
интересует любое значение переменной, кроме 0
   x != 0
интересует високосный
   год year%4 == 0
```

Логические операции.

Одноопе рандные операции.

! логическое отрицание. Применяется для получения обратного результата от логических выражений. Ниже показаны результаты работы этой операции:

```
!1 равно 0
!0 равно 1
```

Если логическим операциям указать значение, отличное от нуля, то оно будет восприниматься как истина. Пример:

```
!5 равно 0
```

Примеры отрицания простых логических выражений:

```
не интересует значение a, если оно больше 10
 !(a > 10)
интересует любое значение x, кроме 0
 !(x == 0)
```

Круглые скобки в выражениях нужны, поскольку операция логического отрицания имеет более высокий приоритет, нежели те операции, которые применены в выражениях.

Двухоперандные операции.

Эти операции предназначены для объединения простых логических выражений в более сложные.

&& - логическое И. При объединении этой операцией двух логических выражений в одно, составное выражение будет истинно только тогда, когда оба простых выражения будут истинны. Например, выражение:

```
a > 10 && a < 20
```

будет истинно, если в переменной a будет находиться число в диапазоне от 11 до 19.

|| - логическое ИЛИ. При объединении этой операцией двух логических выражений в одно, составное выражение будет истинно тогда, когда хотя бы одно простое выражение будет истинным. Например, выражение:

```
a < 10 || a > 20
```

будет истинно, если в переменной a будет находиться число меньше 10 или больше 20.

Условная операция

Это единственная операция в языке Си, которая требует указания трех операндов. Эта операция записывается по следующей форме:

```
выражение1 ? выражение2 : выражение3
```

Если результат **выражения1** истинно, то результатом этой операции будет результат от вычисления **выражения2**, в противном случае от вычисления **выражения3**. В качестве выражений могут использоваться любые выражения, но очень часто в место **выраже-ния1** используется логическое выражение.

Пример:

```
// x присвоить большее из двух чисел a и b
 x = a > b ? a : b ;
// переменной y присвоить абсолютное значение x
y = x >= 0 ? x : -x ;
```

Побитовые операции

Побитовые операции применимы только для целых типов данных и часто используются для анализа и управления значениями двоичных разрядов чисел. Результат работы этих операций есть смысл смотреть только в двоичной системе счисления.

Одноопрерандные операции.

~ двоичное отрицание – инвертирует разряды двоичного числа. Пример:

~ $\underline{11010101_2}$ число

00101010_2 результат. Применяется для получения обратных чисел.

Двухоперандные операции.

Все двухоперандные побитовые операции, за исключением сдвиговых, выполняются над парами двоичных разрядов двух чисел по таблице.

& побитовое И. Эта операция выполняется по следующей таблице:

разряды	результат
00	0
01	0
10	0
11	1

Пример выполнения:

$$\begin{array}{ll} & 01101010_2 \text{ число1} \\ \& & 01011101_2 \text{ число2} \\ \hline & 01001000_2 \text{ результат} \end{array}$$

Часто эту операцию в программировании используют для тестирования состояния двоичного разряда числа - определения его содержимого. Для этого выполняют операцию И над числом и вторым числом, которое должно содержать 1 в интересующем разряде и 0 в остальных. Если в результате операции получится 0, то и в интересующем разряде находится 0, в противном случае 1.

Пример проверки 4-го разряда:

11001110 число

$\&\ \underline{00010000_2}$ сформированное число (маска)

00000000_2 результат- число 0

Если число находится в переменной **x**, то выражение для выполнения тестирования четвертого разряда в этой переменной, на языке Си будет выглядеть следующим образом:

```
(x & 16) == 0
```

Второе широкое применение этой операции для сброса двоичного разряда числа - записи в интересующий разряд 0. Для этого необходимо выполнить операцию И над числом и вторым числом, которое должно содержать в интересующем разряде 0 в остальных 1. Оно обратно числу, используемому для тестирования, и может быть получено операцией НЕ.

Пример сброса 4-го разряда:

10011010 число

$\&\ \underline{11101111_2}$ сформированное число (маска)

10001010_2 результат

Если необходимо в числе, находящемся в переменной **x**, в четвертый двоичный разряд записать 0, то это может быть сделано следующим выражением:

```
x = x & 0xEF;
// или
x = x & ~16;
```

Число 11101111_2 в шестнадцатеричной системе счисления имеет представление EF.

Вместо константы 0xEF, можно было применить константу 239 или 0357, поскольку это разные представления одного и того же числа.

| побитовое ИЛИ.

Эта операция выполняется по следующей таблице:

разряды	результат
00	0
01	1
10	1
11	1

Пример выполнения:

01101000_2 число1

$|\ \underline{01011101_2}$ число2

01111101_2 результат

Широко эта операция применяется для установки двоичного разряда числа в 1. Для этого выполняют операцию ИЛИ над числом и вторым числом, которое должно содержать 1 в интересующем разряде и 0 в остальных.

Пример включения 4-го разряда:

01001010_2 число

$|$

$\underline{00010000_2}$ сформированное число (маска)

01011010_2 результат- число 0.

Если необходимо в числе, находящемся в переменной **x**, в четвертый двоичный разряд записать 1, то это может быть сделано следующим выражением:

```
x = x | 16;
```

^ побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ.

Таблица для этой операции следующая:

разряды	результат
00	0
01	1
10	1
11	0

Пример выполнения:

01101010_2 число1

^ 11011001_2 число2

10110011_2 результат

Часто в программировании используется свойство этой операции, которое позволяет восстановить одно из чисел, участвовавших в операции, если известно второе число и результат. Допустим, что известен результат операции 10110011_2 и одно из чисел, над которым выполнялась операция 11011001_2 (см. предыдущий пример). Тогда, выполнив операцию ИСКЛЮЧАЮЩЕЕ ИЛИ над результатом и одним из чисел, получим второе число.

10110011_2 результат

^

01101010_2 число2

11011001_2 число1

Используя это свойство можно осуществить обмен содержимым двух переменных без применения дополнительной переменной. Ниже показана реализация этого алгоритма для переменных **a** и **b**.

```
a = a ^ b; //В переменной a получится результат операции.
b = a ^ b; //Операция над результатом и числом b даст число a
a = a ^ b; //Операция над результатом и числом a даст число b
```

<< сдвиг двоичных разрядов числа влево. **>>** сдвиг двоичных разрядов числа вправо.

Второй операнд у этих операции должен указывать, на сколько позиций сдвигать разряды. Те разряды, которые выходят за рамки числа, теряются, а вновь прибывающие приходят с нулями при сдвиге влево. При сдвиге вправо знакового числа, если оно отрицательное, происходит заполнение числа единицами, для сохранения знака.

Пример:

10110011_2 число

<< 3_{10} размер сдвига

10011000_2 результат

Сдвиг вправо чисел со знаком и без знака:

10110011_2 число без знака

>> 3_{10} размер сдвига

00010110_2 результат

10110011_2 знаковое целое

>> 3_{10} размер сдвига

11110110_2 результат

Сдвиговые операции можно использовать для формирования чисел, необходимых для тестирования и установки разрядов числа. Например, для того, что бы получить число для тестирования разряда с номером **n**, можно воспользоваться выражением **1 << n**.

Операция сдвига числа влево на **n** разрядов равносильна умножению этого числа на 2^n, а сдвиг числа вправо на **n** разрядов равносилен делению числа на 2^n. Поскольку сдвиговые операции выполняются в компьютере быстрее, нежели арифметические операции, то иногда их используют для реализации умножения или деления числа на 2^n.

Операции присваивания

Все операции присваивания – двухоперандные и требуют, чтобы левым операндом была переменная, значение которой они изменяют. Если тип правого операнда не совпадает с левым, правый операнд будет преобразован к типу левого. Результатом любой операции присваивания будет число, занесенное в переменную, стоящую слева от операции.

= простое присваивание. Поскольку операция имеет результат, то в тексте программ на языке Си можно встретить следующее предложение:

```
a = b = c = d = 0;
```

в результате, которого всем переменным присваивается одно и то же значение.

Кроме простого присваивания в Си существуют сложные операции присваивания, которые позволяют сократить запись выражений типа **a = a + b**, где одна и та же переменная стоит справа и слева от операции присваивания и представить их в виде **a += b**. Все сложные операции присваивания показаны ниже:

Для арифметических операций

+=	сложение с присваиванием
-=	вычитание с присваиванием
***=**	умножение с присваиванием
/=	деление с присваиванием
%=	присваивание остатка от деления

Для побитовых операций

<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	побитовое И с присваиванием
|=	побитовое ИЛИ с присваиванием
^=	ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием.

Прочие операции

Одноопернадные операции **&** получение адреса переменной. Используется для переменных любых типов. Результатом работы этой операции будет адрес первого байта, с которого начинается переменная в оперативной памяти.

Пример:

```
scanf("%d", &a) ;//функции sacnf передается адрес переменной
```

sizeof операция определения размера. Записывается по следующей форме:

```
   sizeof(тип_данных)
или
sizeof(выражение)
```

Результатом работы этой операции будет целое число, которое будет отражать размер операнда в байтах. Результатом выражения **sizeof(double)** будет число 8, а выражение **sizeof(a + b)** даст число 4, если переменные **a** и **b** типа **long**.

Удобно использовать эту операцию с типом **int** в случае написания переносимого под разные платформы текста программы, т.к. размер у этого типа в языке Си плавает и зависит от разрядности процессора.

(тип_данных) операция приведения типа. Позволяет преобразовать значение одного типа данных в другой. Например, для того, чтобы получить остаток от деления целых частей переменных **a** и **b** типа **double** можно воспользоваться выражением **(int)a% (int)b**. Без приведения к типу **int** компилятор будет выдавать ошибку, т.к. операцию получения остатка от деления целых чисел нельзя применять к переменным типа **double**.

Двухоперандные операции

, операция последовательного вычисления. Задается по следующей форме:

```
выражение1 , выражение2
```

Результатом операции будет результат от вычисления **выражения2**. Удобна в тех случаях, когда по синтаксису языка Си требуется указать одно выражение, а для организации работы программы требуется два.

Пример:

```
x = (a > b) ? a += 10, a - b : b += 10, b - a ;
```

переменной **x** присвоить абсолютное значение разности двух чисел **a** и **b**, при этом большее число необходимо увеличить на 10 перед вычислением разности.

Арифметические преобразования данных

В выражениях языка Си могут встречаться разные типы данных, при этом компилятор осуществляет автоматически преобразование операндов к одному типу, а затем реализует операцию. Типом результата будет тип, к которому осуществлялось преобразование операндов.

Арифметические преобразования осуществляются следующим образом:

1. Все операнды типов **char** или **short** преобразуются к **int**, все операнды типов **unsigned char** или **unsigned short** преобразуются к **unsigned int**, все операнды типа **float** преобразуются к типу **double**.
2. Любую пару разнотипных операндов компилятор рассматривает в следующем порядке: если один операнд типа **long double**, то второй преобразуется к типу **long double**, если один операнд типа **double**, то второй преобразуется к типу **double**, если один операнд типа **unsigned long**, то второй преобразуется к типу **unsigned long**, если один операнд типа **long**, то второй преобразуется к ти-
   пу **long**.
3. В операторе присваивания операнд (результат) в правой части преобразуется к типу переменной в левой части, при этом тип может, как повышаться, так и понижаться.

Благодаря этим преобразованиям в программах удается получать правильные результаты и меньше заботиться о диапазонах типов.

Пример:

```
char c1=100, c2 = 100, c3 ;
c3 = c1 + c2 ;                          /* результат -56 */
printf("c3 = %d\n", c3);
printf("c1+c2 = %d\n", c1 + c2);   /* результат 200 */
```

Результат от сложения переменных типа **char** получается правильным, т.к. компилятор перед вычислением выражения преобразует их к типу **int**, но, присвоив результат переменной типа **char**, получаем искажение информации, поскольку число 200 выходит за рамки диапазона этой переменной.

Приоритет и порядок выполнения операций в Си

Наподобие операций в математике, где для операций определен их приоритет и порядок выполнения, в Си то же существует приоритет и порядок выполнения операций в выражении (см. табл.). Порядок выполнения операций можно указывать круглыми скобками
(2+3)*4.

В таблице показаны все операции языка Си и размещены они в порядке убывания приоритета. В одной строке показаны операции, имеющие одинаковый приоритет. Порядок выполнения таких операций в одном выражении оговаривается в третьем столбце таблицы.

Операция	Вид операции	Порядок выполнения нескольких операций в выражении
`() [] . ->`	Выражение	Слева направо
`- ~ ! * & ++ -- sizeof преобразование типа`	Унарный	Справа налево
`* / %`	Мультипликативный	Слева направо
`+ -`	Аддитивный	Слева направо
`<< >>`	Сдвиг	Слева направо
`< > <= >=`	Отношение (неравенство)	Слева направо
`== !=`	Отношение (равенство)	Слева направо
`&`	Побитовое И	Слева направо
`^`	Побитовое исключающее ИЛИ	Слева направо
`|`	Побитовое ИЛИ	Слева направо
`&&`	Логическое И	Слева направо
`||`	Логическое ИЛИ	Слева направо
`?:`	Условная	Справа налево
`= *= /= %= += -= <<= >>= &= != ^=`	Простое и сложные присваивания	Справа налево
`,`	Последовательное вычисление	Слева направо

Для того чтобы определить результат некоторого выражения, необходимо сначала определить порядок выполнения операций в выражении. Затем последовательно, согласно порядку, выполнить операции, заменяя их вместе с операндами результатом. В конце получиться результат всего выражения.

Пример:

```
2 + 3 << 4 / 2 >= 1 != 4 * 2     выражение
  3    4   1   5     6    2      порядок выполнения операций
1. 2 + 3 << 2 >= 1  != 4 * 2     выполнена операция 4  / 2
2. 2 + 3 << 2 >= 1  != 8         выполнена операция 4  * 2
3. 5 << 2 >= 1 != 8              выполнена операция 2  + 3
4. 20 >= 1 != 8            выполнена операция 5 << 2
5. 1 != 8                  выполнена операция 20 >= 1
6. 1                       выполнена операция 1 != 8
```

Задачи

1. Чему будет равен результат от вычисления выражений:
   - 10 / 3 =
   - 2.5 / 5 =
   - 11 % 7 =
   - 5 % 15 =
   - 6 > 10 =
   - 8 >= 8 =
   - 5 != 5 =
   - 3 & 1 =
   - 4 | 1 =
   - 2 << 2 =
   - 60 >> 1 =

2. Запишите выражение, с помощью которого можно было бы определить, делится ли число, находящееся в переменной **m** на 8 без остатка.
3. В переменной **m** содержится число 3. Чему будет равен результат выражения **++m - 3.**
4. Переменные **x** и **y** содержат число 4. Чему будет равен результат выражения **++m + y--**.
5. Запишите выражение, которое позволит протестировать содержимое 5-го разряда переменной **m**.
6. Запишите предложение, которое позволит занести в 3-й разряд переменной **m** 0.
7. Запишите предложение, которое позволит занести в 4-й разряд переменной **m** 1.
8. Определите результат выражения: 1+8/2*2.
9. Определите результат выражения: 10 >> 2 + 4/2 - 3 << 2.

10. Написать программу, которая бы определяла скорость, с которой автомобиль проехал некоторое расстояние. Расстояние ввести в километрах, а время в виде дробного числа, в котором целая часть будет задавать количество минут, а дробная количество секунд.
11. Написать программу определения величины дохода по вкладу. Программа должна получить от пользователя размер вклада и срок вклада в днях. Если срок вклада больше 1 года, то для расчетов использовать ставку 2%, в противном случае 1,5%. Результаты выводить с округлением до второго знака после запятой.
12. Написать программу, которая бы проверяла знание таблицы умножения. Программа должна вывести на экран два случайных числа в диапазоне от 2 до 9 и спросить пользователя, какой будет результат от перемножения этих чисел. Затем проверить ответ и выдать оценку: 5 - если ответ верный или 2 - если ответ неверный.
13. Написать программу пересчета величины временного интервала, заданного в минутах, в величину, выраженную в часах и минутах.
14. Написать программу, которая преобразует введенное с клавиатуры дробное число в денежный формат типа: 12 руб. 50 коп.

4. Управляющие операторы в языке Си.

Оператор – это законченная конструкция языка, реализующая определенные действия

в программе. В языке Си различают четыре вида операторов.

1. Простой оператор - это простое предложение, которое закрывается точкой с запятой.

Пример простых операторов:

```
a = b + c;
printf("Hello Wirld.\n");
```

2. Составной оператор - это последовательность операторов, заключенная в фигурные скобки. В основном такие операторы используются как часть управляющих операторов.

Пример составного оператора:

```
{
  y = 2 * x + 3;
  printf("y= %lf\n",y);
}
```

3. Пустой оператор - это отдельно стоящая точка с запятой. Например, простой оператор y = 2 * x + 3; ; закрыт двумя точками с запятой. Вторая точка с запятой будет являться законченной конструкцией, которая ничего не будет делать -

пустой оператор. В таком виде этот оператор не используется, но может быть полезен в управляющих операторах.

4. Управляющий оператор - это оператор, с помощью которого можно управлять порядком выполнения операторов в программе.

В языке Си различают три группы управляющих операторов.

Операторы выбора **if** и **switch**.

Операторы повторения **for**, **while** и **do-**

while. Операторы перехода **break**,

continue и **goto**.

Условный оператор if.

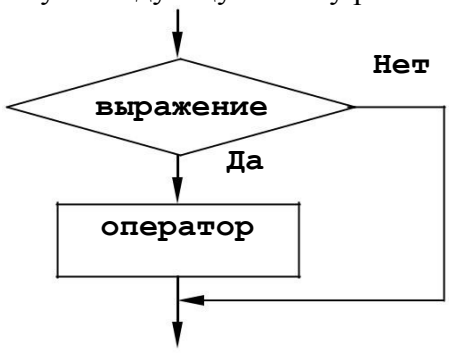
Простая форма этого оператора предназначена для реализации выполнения, или обхода другого оператора, используемого в конструкции **if**, и имеет следующий вид:

```
if(выражение)
   оператор
```

выражение - это любое выражение языка Си, которое вычисляется и результат анализируется на предмет истинности. Поэтому очень часто используют логическое выражение. В случае применения другого выражения, результат анализируется следующим образом: 0 - ложь, отличное от нуля значение - истина.

оператор - это один из четырех видов операторов: пустой, простой, составной или управляющий.

Эта конструкция организует следующую схему работы программы.



Если выражение истинно, то будет выполнен оператор, стоящий в этой конструкции, в противном случае оператор будет пропущен программой.

Допустим, программа получает от пользователя пароль в виде числа в переменной **p**. Если полученное число не равно заданному числу, например, 123, то программа должна выдать на экран сообщение о неправильности ввода пароля. Это можно реализовать с помощью простой конструкции оператора **if** следующим образом:

```
if(p != 123)
  printf("Ошибка при вводе пароля.\n") ;
```

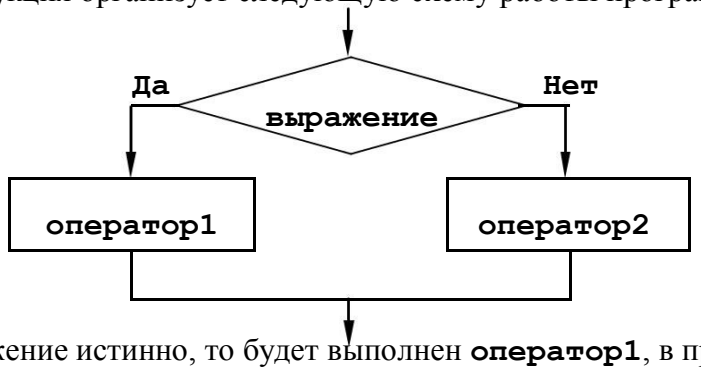
Если при решении этой задачи необходимо подсчитывать количество попыток, сделанных пользователем, то в конструкции **if** придется выполнить еще один простой оператор, увеличивающий значение некоторой переменной **n** на единицу. Это можно сделать, применив, составной оператор.

```
If (p != 123)
{
  printf("Ошибка при вводе пароля.");
  n++ ;
}
```

Простую форму оператора **if** можно продолжить словом **else** и еще одним оператором. Получиться законченная конструкция условного оператора, реализующая схему, называемую развилкой, в которой будет выполнен один из двух операторов. Форма задания следующая:

```
if(выражение)
    оператор1
else
   оператор2
```

Эта конструкция организует следующую схему работы программы.



Если выражение истинно, то будет выполнен **оператор1**, в противном случае **оператор2**.

Называют эту конструкцию **if-else**. Она полезна в тех случаях, когда в программе необходимо выполнить один из двух операторов. Например, программа должна поздравить пользователя, если он угадал число и огорчить в случае неудачи. Фрагмент реализации показан ниже:

```
If (a == b)
       printf("Поздравляю! Вы угадали.\n");
else
       printf("Попробуй еще.\n");
```

Если при удачной попытке программе необходимо прибавить пользователю некоторое количество очков, то на месте **оператора1** придется выполнить еще один оператор.

```
If (a == b)
{
       printf("Поздравляю! Вы угадали.\n");
  bonus += 5; // Увеличить на 5 имеющееся количество очков

}
else
       printf("Попробуй еще.\n");
```

Если не применить составной оператор, то конструкция **if** окажется разорванной оператором **bonus += 5**, и компилятор будет выдавать ошибку, связанную с неправильным применением ключевого слова **else**.

Вместо простых и составных операторов, в конструкции **if** можно использовать любой управляющий оператор, в том числе и оператор **if**. При этом может получиться следующая схема:

```
if (выражение 1)
if (выражение 2)
   оператор1
```

Оператор1 будет выполнен только тогда, когда будут истинны оба выражения. Эту конструкцию можно привести к более простой, использовав сложное логическое выражение.

```
if (выражение 1 &&
  выражение2) оператор1
```

Но она имеет продолжение, т.к. можно добавить слово **else** и второй оператор.

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

к таких конструкциях слово **else** со своим оператором будет отнесено компилятором ближайшему **if**, который не является законченной конструкцией (если их ничего не разделяет), т.е. ко второму оператору **if**.

Если необходимо, чтобы слово **else** было отнесено к первому оператору **if**, то необходимо воспользоваться фигурными скобками, как показано ниже.

```
if(выражение1) {
if(выражение2)
       оператор1
}
else
       оператор2
```

Поскольку первый оператор **if** в предыдущей конструкции остался открытым, то можно добавить еще одно слово **else**. Получиться следующая законченная конструкция:

```
if (выражение1)
if (выражение2)
   оператор1
else
   оператор2
else
    оператор3
```

Более интересная конструкция получается, если оператор **if** применен после слова **else**.

```
 if(выражение1)
оператор1
else if(выражение2)
           оператор2
```

Эта конструкция имеет свое продолжение, т.к. второй оператор **if** может быть продолжен словом **else**, после которого можно опять поставить оператор **if** и т.д.

```
if(выражение1)
оператор1
else if(выражение2)
оператор2
else
if(выражение3)
  оператор3
...
else операторN
```

Называют эту конструкцию **if-else-if**. Она полезна в тех случаях, когда в программе необходимо выполнить один из многих операторов. Например, программа должна определить, с каким символом она имеет дело в переменной **ch**: цифрой, большой латинской буквой или маленькой латинской буквой. Это можно сделать следующим фрагментом:

```
if (ch>= '0' && ch<= '9')
printf("Цифра.");
else if (ch >= 'A' && ch <= 'Z')
            printf("Большая латинская буква.");
else if (ch >= 'a' && ch <= 'z')
            printf("Малая латинская буква.\n");
```

```
else
            printf("Неизвестный символ.\n");
```

Оператор switch (переключатель)

Этот оператор предназначен для осуществления выбора одного оператора из многих и по назначению похож на конструкцию **if-else-if**. Используется в тех случаях, когда для каждого значения переменной или выражения требуется выполнить определенный набор операторов.

Форма задания:

```
switch (выражение)
{
case константа1:
       операторы1 break;
 case константа2:
       операторы2 break;
...
case константаN:
       операторыN break;
...
default: последовательность операторов
         break;
}
```

выражение - целочисленное выражение языка Си, очень часто имя переменной, значение которой анализируется.

константа - любая константа числового типа или выражение, состоящее из одних только констант. Значения констант в одной конструкции **switch** не должны повторяться.

операторы - любая последовательность операторов, которая обычно заканчивается оператором **break**.

Работает оператор **switch** следующим образом. Вычисляется выражение, и программа переходит на тот набор операторов, который идет после константы, значение которой равно результату выражения. Оператор **break** прерывает выполнение оператора **switch** и отправляет программу вниз за рамки конструкции.

Набор операторов, идущий после слова **default**, будет выполняться только тогда, когда результат выражения не будет равен ни одной из констант, используемых в операторе **switch**. Эту часть конструкции можно опускать.

Ниже показан фрагмент программы калькулятор, который с помощью оператора **switch** анализирует значение переменной **op** - кода операции, и в зависимости от значения реализует нужную операцию. В переменных **a** и **b** содержаться операнды операции.

```
switch(op)
{
case '+': /* Реализация сложения */
  printf("%lg + %lg = %lg\n",a,b, a + b);
  break ;
case '-': /* Реализация вычитания */
  printf("%lg - %lg = %lg\n",a,b, a - b);
  break ;
```

```
case '*': /* Реализация умножения */
   printf("%lg + %lg = %lg\n",a,b,a * b);
   break ;
case '/': /* Реализация деления */
   printf("%lg + %lg = %lg\n",a,b,a / b);
   break ;
case '%': /* Реализация остатка от деления целых чисел */
   printf("%d %% %d = %d\n",(int)a,(int)b,(int)a % (int)b);
   break ;
default: printf("Ошибка при вводе выражения.\n");
}
```

Цикл for

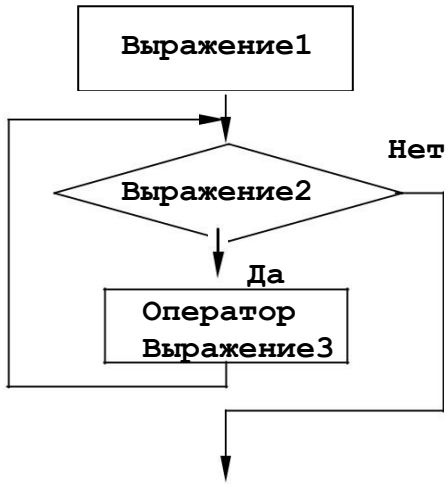
Циклы - это конструкции языка программирования, которые позволяют повторять некоторый набор операторов в программе определенное количество раз.

Несмотря на то, что в языке Си имеется три конструкции циклов, любую задачу, которая требует повторения операторов, можно реализовать с помощью любого из этих циклов. При использовании циклов в программах руководствуются удобством применения той или иной конструкции при решении некоторой задачи.

Цикл **for** удобно применять, когда заранее известно количество повторений некоторого набора операторов. Форма задания следующая:

```
for ([выражение1]; [выражение2]; [выражение3])
       оператор
```

Эта конструкция реализует следующую схему работы программы:



Выражение1 выполняется один раз при входе программы в цикл и используется для задания начального значения параметру цикла - переменной, с помощью которой происходит управление конструкцией.

Выражение2 анализируется каждый раз перед выполнением оператора и должно задавать условие окончания повторения.

Выражение3 срабатывает каждый раз после выполнения оператора и используется для изменения значения параметра цикла.

Оператор - это повторяющаяся часть, которая может быть пустым, простым, составным или управляющим оператором. Один проход программы от **выражения2** до **выра-жения3** называют шагом цикла.

Ниже показан пример типичного использования цикла **for**. Его можно использовать для нумерации столбцов таблицы, выводимой программой на экран.

```
for (i = 1; i <= 9; i++)
     printf(" %d ", i );
```

В этом примере переменная **i** является параметром цикла и одновременно ее значение используется в функции **printf** как номер столбца. В таблице девять столбцов, поэтому **выражение1** задает начальный номер столбца, **выражение2** оговаривает, до какого значения должен меняться параметр цикла, а **выражение3** задает шаг изменения. Цикл закончит свою работу, когда переменная **i** примет значений 10.

Количество повторений можно изменять в процессе работы программы, если вместо константы, ограничивающей изменение параметра цикла в **выражении2** применить переменную.

```
f = 1;
for (i=2; i <= n; i++)
     f *= i;
```

В этом примере решается задача получения факториала числа. Факториалом числа N называют произведение целых чисел от 1 до N. Поскольку заранее неизвестно, факториал какого числа необходимо вычислить, то цикл будет повторять оператор **f *= i** до значения переменной **n**, которая должна содержать само число.

В конструкции циклов можно применять пустой оператор, как показано в примере:

```
for (i=0; i<1000000; i++);
```

С помощью такого фрагмента можно реализовать задержку в работе программы. Этот цикл будет выполнять пустой оператор 1000000 раз, на что уйдет некоторое время, которое зависит от скорости работы процессора.

Если в цикле необходимо повторять несколько операторов, то их необходимо заключить в фигурные скобки, сделать составным оператором.

```
for (i = 1; i <= n; i++)
{
  printf("Введите %i-е число:",n);
  scanf("%lf", &a) ;
  sum += a;       /* sum = sum + a */
}
```

Этот фрагмент получает от пользователя n вещественных чисел и подсчитывает их сумму. Переменная **sum** перед выполнением фрагмента должна содержать число 0.

Выражения в круглых скобках цикла **for** можно опускать, но точки с запятой должны всегда присутствовать. Пропуск выражения приводит к тому, что в блок-схеме исчезает соответствующая этому выражению часть. В фрагменте, показанном ниже опущено первое и третье выражения в цикле **for**.

```
printf("Введите последовательность чисел,
\ заканчивающуюся 0.\n");
for (; a!=0; )
{
  scanf("%lf", &a);
  sum += a;       /* sum = sum + a */
}
```

Этот фрагмент получает от пользователя неограниченное количество чисел и подсчитывает их сумму. Закончить последовательность чисел необходимо числом 0. Перед выполнением цикла значение переменной **a** не должно быть равно 0, иначе цикл выполняться не будет.

Имеет право на жизнь и такой фрагмент

```
for( ; ; ) ;
```

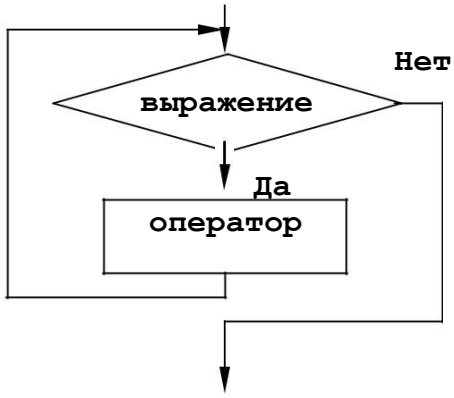
бесконечный цикл, который приводит к "зависанию" программы.

Цикл while

Этот цикл удобно применять, когда заранее неизвестно количество повторений. Момент выхода из цикла определяется в процессе его работы. Форма задания цикла **while** напоминает форму оператор **if** и имеет следующий вид:

```
while(выражение)
        оператор
```

Эта конструкция реализует следующую схему работы программы:



Выполнение оператора повторяется до тех пор, пока выражение дает результат не равный 0. Эта схема может быть получена и при использовании цикла **for**, если опустить первое и третье выражения. Ниже показан фрагмент получения неограниченного количества чисел от пользователя и подсчета суммы, реализованный с помощью цикла **while**.

```
while( a != 0 )
{
   scanf("%lf", &a) ;
   sum += a;       /* sum = sum + a */
}
```

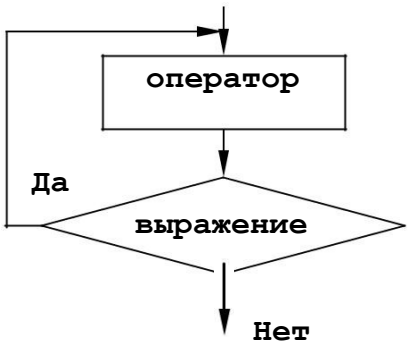
Рассмотренные ранее циклы относятся к одному виду. Они реализуют цикл "пока" – цикл с предусловием, в котором сначала анализируется условие, затем выполняется оператор. В такой схеме организации программы повторяющаяся часть в цикле может ни разу не выполниться, если условие при входе в цикл будет ложным.

Цикл do-while

Управляющий оператор **do-while** реализует конструкцию цикла "до" – цикл с постусловием, в котором сначала выполняется оператор, затем анализируется условие. Форма задания следующая:

```
do
   оператор
while( выражение );
```

Схема работы цикла показана ниже.



Этот цикл удобно применять в тех случаях, когда хотя бы один раз повторяющаяся часть должна быть выполнена.

В рассмотренных выше примерах получения неограниченного количества чисел от пользователя ощущается некоторое неудобство, т.к. значение переменной **a** перед выполнением цикла не должно равняться 0, поскольку при таком значении циклы **for** и **while** ни разу не выполнят свои операторы. При реализации этой задачи удобнее будет воспользоваться циклом **do-while** и необходимость в отслеживании значения переменной **a** отпадет.

```
do
{
   scanf("%lf", &a) ;
   sum += a ;      /* sum = sum + a */
} while( a != 0 ) ;
```

Вложенные циклы

Когда внутри одного цикла находиться в качестве оператора или в составе операторов другой цикл, то второй цикл называют вложенным. При такой реализации внутренний цикл будет срабатывать на каждом шаге наружного цикла, и выполнять заданное количество повторений оператора. Например, для вывода на экран таблицы, содержащей факториалы чисел от 1 до 10 можно воспользоваться следующей конструкцией вложенных циклов:

```
   for(n = 1; n <= 10; n++)
   {
        f = 1;
        for(i=2; i<= n; i++)
             f *= i ;
        printf("%2d -> %d\n", n, f) ;
   }
```

Переменная **n** в этом фрагменте задает число, факториал которого вычисляется во внутреннем цикле. Поскольку значение **n** меняется от 1 до 10, то на экран выводится таблица, в каждой строке которой показано число (значение **n**) и его факториал.

Оператор break

Оператор **break** можно использовать внутри любого цикла и оператора **switch**. Выполнение этого оператора приводит к прерыванию работы конструкции, в которой он используется и переходу программы за рамки конструкции. Переход осуществляется всегда в низ на первый оператор, идущий за конструкцией.

В других местах программы оператор **break** применять нельзя.

В циклах его обычно используют по следующей схеме, показанной на примере оператора **while**:

```
while(выражение)
{
   операторы
   if( условие ) break ;
   операторы
}
```

Этот управляющий оператор полезен в тех случаях, когда обеспечить выход из цикла по условию затруднительно, т.к. момент выхода может быть определен внутри цикла. Ниже показан фрагмент, реализующий посимвольный разбор строки, вводимой пользователем, при этом подсчитывается количество полученных символов. Символ новой строки, который получается при нажатии клавиши Enter, всегда завершающей ввод данных, не должен учитываться.

```
n = 0;
while(1)
{
   ch = getchar() ;
   if(ch =='\n') break;
   n++ ;
}
```

Условие окончания работы цикла можно проанализировать после работы функции **getchar**, потому что в этот момент в программе появляется очередной символ, введенный пользователем, который еще не подсчитан и таким символом может оказаться символ новой строки.

В качестве выражения в цикле **while**, показанном в предыдущем примере, применена ненулевая константа, значение которой конструкцией будет интерпретироваться как истина. Так часто поступают, когда выполнение цикла прерывается оператором **break**.

Оператор continue

Оператор **continue** можно использовать только внутри любого цикла. Выполнение этого оператора приводит к переходу цикла на следующий шаг и пропуску операторов в цикле, идущих после **continue**.

В циклах его обычно используют по следующей схеме, показанной на примере оператора **while**:

```
while(выражение)
{
   операторы
   if( условие ) continue ;
   операторы
}
```

Применяют его тогда, когда внутри повторяющихся в цикле операторов может возникать условие, при котором необходимо пропустить оставшуюся часть операторов и перейти на следующий шаг цикла. Так, при решении задачи расчета корней квадратного уравнения и реализации проверок правильности ввода пользователем данных, получение значения коэффициента a при x^2 может быть реализовано следующим образом:

```
while(1)
{
   printf("Введите коэффициенты квадратного уравнения:");
   printf("a = ");
   scanf("%lf",&a);
   if(a == 0)
   {
```

```
            printf(" Значение a не может быть = 0.\n");
            continue;
      }
      printf("b = ");
      scanf("%lf",&b);
      ...
}
```

В этом примере применен цикл для того, чтобы дать возможность пользователю повторить расчет, а оператор **continue** для пропуска последующих операторов в случае неправильного ввода значения коэффициента **a**.

Оператор goto

Оператор **goto** предназначен для выполнения перехода в нужное место программы. Используется редко, т.к. его применение считается плохим стилем программирования не только в языке Си. Избежать применения этого оператора можно, используя, условный оператор или оператор цикла, т.к. переход вниз - это пропуск операторов, который реализуется конструкцией **if**, а переход вверх - это повторение, реализуемое любым циклом.

Форма задания:

```
goto метка;
...
метка: оператор1;
```

метка - любое имя, заданное по правилам языка Си, которое не требует предварительного объявления. Метка может располагаться ниже или выше оператора **goto**, но в пределах тела функции.

Иногда оператор **goto** бывает полезен и ниже показан пример возможного применения этого оператора:

```
while(выражение1)
{  ...
   while(выражение2)
   { ...
         while(выражение3)
          {    ...
               if(условие)    goto m1;


               ...
          }
     ...
   }
}
m1: оператор;
```

В этом примере показано три вложенных цикла. При такой реализации решения некоторой задачи, внутри цикла, содержащего **выражение3**, может понадобиться прервать выполнение сразу трех циклов. Оператор **break** прерывает выполнение только того цикла, в котором он используется, но оператором **goto** можно перейти за рамки этих циклов, пометив меткой соответствующий оператор.

Задачи

1. Перепишите оператор **if**, исправив допущенные ошибки: **if a > b a = b ;**
2. Перепишите оператор **for**, исправив допущенные ошибки: **for(a < 10) sum += a++;**

3. Запишите конструкцию, с помощью которой программа вычисляла бы выражение y = 1/x если x не равен 0.
4. Запишите конструкцию, которая реализовывала бы следующий алгоритм:

   если **x** не равен 0

   вычислить выражение y = 1/x

   вывести значение переменной **y** на

   экран в противном случае

   вывести сообщение о том, что x не может быть равен 0.
5. Записать конструкцию, с помощью которой программа выдавала бы на экран один символ, код которого содержится в переменной **ch**, если значение этой переменной не равно символу новой строки.
6. Запишите конструкцию, с помощью которой можно прервать выполнение цикла, если значение переменной **a** равно 0.
7. Записать конструкцию, с помощью которой десять раз повторится оператор **printf("Hello World\n") ;**
8. Написать программу расчета корней квадратного уравнения ($ax^2 + bx + c = 0$). В программе сделать необходимые проверки, а именно коэффициент не должен быть равен 0 и дискриминант не должен быть отрицательным.
9. Написать программу, которая проверяет, является ли введенное пользователем целое число четным.
10. Написать программу, которая вычисляет оптимальный вес для пользователя, сравнивая его с реальным, и выдает рекомендацию о необходимости поправиться или

    похудеть и насколько килограмм. Оптимальный вес вычисляется по формуле:

    Рост (см) – 100.
11. Написать программу, которая выводит пример на умножение двух однозначных чисел, запрашивает ответ от пользователя, проверяет его и выводит сообщение "Правильно!" или "Вы ошиблись" и правильный ответ.

12. Написать программу, которая запрашивает у пользователя номер дня недели и выводит одно из сообщений: "Рабочий день","Суббота" или "Воскресенье".
13. Написать программу, которая вычисляет дату следующего дня. Дату получить в формате: день.месяц.год.
14. Напишите программу, которая запрашивает у пользователя номер дня недели, затем выводит название дня недели или сообщение об ошибке, если введены неверные данные.
15. Написать программу, которая переводит время, заданное в формате ЧЧ:ММ:СС (часы:минуты:секунды) в секунды. Программа должна проверять правильность введенных пользователем данных.
16. Написать программу, которая выводит на экран Ваши имя и фамилию 10 раз.
17. Реализовать программу, отображающую работающие часы. Для получения текущего времени, воспользоваться функцией **time**.
18. Написать программу, которая выводит таблицу степеней двойки от нулевой до десятой не используя функцию **pow**.
19. Написать программу, которая генерирует последовательность из 10 случайных чисел в диапазоне от 1 до 10, выводит эти числа на экран и вычисляет их среднее арифметическое.
20. Реализовать программу, отображающую число в двоичной системе счисления. Для этого необходимо в цикле **for** реализовать тестирование каждого разряда числа и вывести его значение в виде 0 или 1. Число программа должна получить от пользователя.
21. Составьте программу печати равнобедренного прямоугольного треугольника из вездочек

    *

    * *

```
***
****
*****
```

используя цикл **for**. Программа должна запрашивать размер вертикального катета треугольника.

22. Напишите программу с циклами, которая рисует равнобедренный треугольник:

```
    *
   ***
  *****
 *******
*********
```

Программа должна запрашивать высоту треугольника.

23. Написать программу, которая выводит таблицу умножения на указанное одноразрядное число.
24. Написать программу для вывода таблицы умножения в следующем виде

```
1 x 1 = 1   2 x 1 = 2   3 x 1 = 3   4 x 1 = 4    5 x 1 = 5    6 x 1 = 6
1 x 2 = 2   2 x 2 = 4   3 x 2 = 6   4 x 2 = 8    5 x 2 = 10   6 x 2 = 12
1 x 3 = 3   2 x 3 = 6   3 x 3 = 9   4 x 3 = 12 5 x 3 = 15    6 x 3 = 18
…
```

25. Написать программу вычисления стоимости покупки, состоящей из любого количества товаров. Закончить получение информации при вводе количества, равного 0.
26. В калькуляторе реализовать операцию, которая бы выполняла возведение числа **a** в степень **b**. Для этого воспользоваться функцией **pow(a,b)**, результатом которой будет число $\mathbf{a}^{\mathbf{b}}$. Код операции можно обозначить символом **^**.
27. Написать программу, вычисляющую сумму и среднее арифметическое последовательности положительных чисел, которые вводятся с клавиатуры. Ввод чисел закончить 0.
28. Сделать программу проверки знания таблицы умножения. Программа должна задать десять вопросов, определить количество правильных ответов и выставить следующие оценки:

Количество правильных ответов	Оценка
10	5
8, 9	4
6, 7	3
< 6	2

Если оценка равна 2 или 3, то выдать на экран таблицу умножения с советом изучить ее.

29. Сделать калькулятор, который бы вычислял значения различных математических функций. Программа должна получить выражение типа s45 (синус 45 градусов) и выдать результат в виде sin 45 = 0.70709.
30. Реализовать калькулятор, который бы вычислял как выражения, так и значения математических функций. Для этого, получение выражения нужно выполнить двумя вызовами функции **scanf**. Первый вызов должен получить первое число в выражении и код операции. Второе число должно быть получено вторым вызовом функции **scanf** в том случае, если введен код операции, для которой требуется указание второго операнда.

5. Препроцессор языка Си

Препроцессор - это составная часть компилятора языка Си, выполняющая обработку текстового файла перед компиляцией. Препроцессор - это мощный инструмент, часто используемый для повышения удобочитаемости, модифицирования и переносимости текстов программ.

Управляется препроцессор специальными командами, называемыми директивами препроцессора, которые можно указывать в текстовом файле программы.

Препроцессор выполняет следующие основные функции:

1. Вставку содержимого указанного файла.
2. Поиск фрагмента текста и замену его на другой фрагмент.
3. Удаление не нужных в данный момент фрагментов текста программы.

Все эти функции выполняет практически любой текстовый редактор, однако, в отличие от текстовых редакторов, препроцессор не изменяет исходный файл, а создает новый временный файл, который в дальнейшем и обрабатывается компилятором.

У директив препроцессора свой синтаксис и свои правила. Для всех директив существует общее правило – одна директива одна строка. Это означает, что директива препроцессора не должна занимать более одной строки и на одной строке должна быть указана одна директива, а также в директивах препроцессора нельзя применять лишние пробельные символы.

Все директивы препроцессора начинаются с символа **#.**

Директива #include

Директива **#include** дает указание препроцессору вставить в то место, где она стоит, содержимое указанного файла.

Существует две формы задания этой директивы:

```
#include <имя файла>  // 1
```

При использовании этой формы, указанный в угловых скобках файл, будет отыскиваться препроцессором в известных ему каталогах. Обычно эти каталоги называются in-clude и указываются в настройках среды программирования.

Эта форма используется для вставки содержимого стандартных заголовочных файлов, поставляемых разработчиком компилятора.

```
#include "имя файла"  // 2
```

При указании имени файла в двойных кавычках, препроцессор сначала будет искать файл в текущем каталоге, в котором находиться файл программы на Си, содержащий эту директиву. Если указанный файл не будет найден в текущем каталоге, препроцессор будет осуществлять поиск в известных ему каталогах.

Эта форма используется для вставки содержимого собственных заголовочных файлов, которые создает программист при написании программы.

В обоих случаях можно указывать путь к файлу, используя принятые в системе соглашения. Например,

```
#include "c:\myprog\my.h"
```

При указании пути к файлу должна применяться одна косая черта, т.к. указание имени файла в двойных кавычках не имеет ничего общего со строкой-литералом, поскольку препроцессор осуществляет текстовую обработку файла программы, а синтаксис языка Си используется только компилятором. В переходном файле, созданном препроцессором, не будет содержаться ни одной директивы, т.к. они туда не переносятся, а заменяются результатами работы препроцессора.

Директива #define

Директива **define** имеет три формы задания и используется для реализации поиска фрагмента текста и замены его на другой фрагмент.

```
// 1-я форма
#define макро_имя строка_замены
```

макро_имя - имя, заданное по правилу формирования идентификаторов;

строка_замены - фрагмент текста, на который будет заменено **макро_имя**.

Выполняя эту директиву, препроцессор будет осуществлять поиск **макро_имени**, на-

чиная с места, в котором указана директива, до конца файла и заменять на указанную строку замены. Из поиска будут исключаться комментарии, строки-литералы и идентификаторы, содержащие в своем составе **макро_имя**.

Эта форма директивы **define** применяется:

1. Для сокращения текста программы, за счет замены повторяющихся фрагментов текста коротким макроименем:

```
#define STOP printf("Нажмите Enter");getchar();\
fflush(stdin)
```

Далее, в нужных местах текста программы можно использовать имя **STOP** для остановки программы до нажатия клавиши Enter, как показано в примере ниже:

```
…
STOP;
…
```

Имя **STOP** препроцессор заменит на указанную в директиве **define** строку.

Если строка замены получается длинной, то ее можно перенести на следующую строку, так же, как строку-литерал:

```
#define STOP { printf("Для продолжения нажмите Enter") ; \↵
getchar() ; \↵
fflush(stdin) ;
}
```

2. Для задания имен константам, что повышает наглядность текста программы и ускоряет перестройку программы при использовании других значений.

Например, в программе, которая выводит на экран таблицу, содержащую десять строк
и десять столбцов, константа 10 в одних местах будет задавать число строк, в других число столбцов. Поэтому, по значению константы нельзя будет определить, с чем работает фрагмент, содержащий это значение без рассмотрения действий, которые выполняются. Кроме того, при перестройке программы на таблицу другого размера, необходимо пройтись по тексту и осуществить замену одних констант на требуемое количество строк, а других констант на требуемое количество столбцов, что может сопровождаться ошибками
и иногда требует много времени.

Такие задачи лучше реализовывать, используя имена констант вместо самих констант, как показано ниже:

```
#define WIDTH 10
#define ROW 10
...
   for(i = 0; i < ROW; i++)
   {
         ...
         for(j = 0; j < WIDTH; j++)
         ...
}
```

Перестройка таких фрагментов осуществляется редактированием только строки замены в директиве **define**. Наглядность текста программы, используемой именованные константы, выше, поскольку по именам видно, что первый цикл **for** отвечает за высоту таблицы, а второй за ширину.

```
// 2-я форма
#define макро_имя(список_имен_параметров) строка_замены
```

макро_имя - имя, заданное по правилу формирования идентификаторов;

список_имен_параметров - имена, перечисленные через запятую, которые при

использовании будут идентифицировать фрагменты текста, используемые для подстройки строки замены;

строка_замены - фрагмент текста, на который будет заменено **макро_имя** и в котором должны использоваться имена параметров, указанные в круглых скобках.

Как и в предыдущем случае, эта форма директивы заставляет препроцессор выполнять поиск и замену **макро_имени** на указанную строку и подстраивать строку замены фрагментами текста, которые будут указаны в круглых скобках вместо имен параметров.

Пример:

```
#define PI 3.1415
…
#define AREA(R) PI*R*R
```

В этом примере в круглых скобках указано одно имя параметра, которое затем используется в выражении, вычисляющем площадь круга, указанном в строке замены. При использовании в дальнейшем имени **AREA** в круглых скобках необходимо указывать фрагмент текста, которым будет заменяться имя параметра в выражении:

```
s = AREA(x);
```

После работы препроцессора этот оператор будет выглядеть следующим образом:

```
s = 3.1415*x*x;
```

При использовании макроимен, которые задаются второй формой директивы **define**, приходится делать записи, напоминающие вызов функции, поэтому эти макроимена еще называют макрофункциями.

При использовании макрофункций иногда приходится сталкиваться с трудно обнаруживаемыми ошибками, как показано ниже:

```
#define SQ(a) a*a
…
x = 2;
…
y = SQ(x+1);
```

В переменной **y** окажется число 5 вместо 6, т.к. после замены последний оператор будет иметь следующий вид:

```
y = x+1*x+1;
```

Такого рода ошибки можно исключить, применив круглые скобки в строке замены директивы **define**. Круглые скобки должны быть расставлены по следующему правилу: каждое имя параметра в строке замены и все выражение должны быть заключены в круглые скобки:

```
#define SQ(a) ((a)*(a))
```

Однако это не спасает от следующего неправильного результата:

```
x = 2;
…
y = SQ(++x);
```

В переменной **y** окажется число 12 или 16 вместо 9, т.к. после замены последний оператор будет иметь следующий вид:

```
y = ((++x)*(++x));
```

Порядок вычисления таких выражений стандартами языка не регламентируется, кроме того значение переменной **x** увеличивается не на единицу, а на два.

Такие ошибки нельзя исключить, поэтому необходимо избегать применения унарных операций **++** и **--** в макрофункциях.

При задании макроимен принято использовать большие буквы, что позволяет легко отличать их от других идентификаторов в тексте программы.

```
// 3-я форма
#define макро_имя
```

В этой форме директивы строка замены не указывается. Использовать ее можно для создания имен, повышаемых наглядность текста программы и которые впоследствии должны быть удалены препроцессором.

```
#define  VARIABLES
#define  OPERATORS
#define PROGRAMM
 PROGRAMM main()     // Программа

 {
 VARIABLES           // Начало описания переменных
        int a, b;
 OPERATORS           // Начало операторов
        printf(…);
        scanf(…);
        a = b + 5;
```

Но в основном эта форма используется для управления группой директив условной компиляции, которые будут рассмотрены ниже.

Директива #undef

Синтаксис:

```
#undef макро_имя
```

Директива **#undef** отменяет текущее определение **макро_имени**. Поэтому все встречающиеся появления **макро_имени** будут игнорироваться предпроцессором. Для удаления определения макрофункции с использованием **#undef**, нужно указать только идентификатор макрофункции, не задавая список параметров.

Можно применить директиву **#undef** к идентификатору, у которого нет определения. Тем самым программист получает дополнительную гарантию того, что данный идентификатор не определен.

Директивы условной компиляции

Эти директивы позволяют осуществлять удаление ненужных в текущем процессе компиляции фрагментов текста программы. Поэтому их называют директивами условной компиляции. Директивы условной компиляции используются для управления значениями макроконстант, удаления отладочных фрагментов при создании рабочей версии программы, для адаптации текста программы под текущую платформу.

Директива #ifdef.

Она дает возможность в зависимости от наличия **макро_имени,** определяемого директивой **#define**, управлять одним или двумя фрагментами текста.

#ifdef означает "if defined" (если определено). Синтаксис:

```
#ifdef макро_имя
```

```
   фрагмент текста
1 #else
   фрагмент текста
2 #endif
```

Если директивой **#define** ранее было определено **макро_имя**, тогда первый фрагмент текста будет оставлен, а второй фрагмент текста будет удален, в противном случае директива сработает наоборот.

Директива **#else** и второй фрагмент текста могут не указываться.

Любая директива условной компиляции должна заканчиваться директивой **#endif**.

Директива #ifndef.

Форма этой директивы совпадает с директивой **#ifdef**, но работает она наоборот. **#ifndef** означает "if not defined" (если не определено).

Синтаксис:

```
#ifndef макро_имя
   фрагмент текста
   1
#else
   фрагмент текста
2 #endif
```

Если директивой **#define** ранее было определено **макро_имя**, тогда первый фрагмент текста будет удален, а второй фрагмент текста будет оставлен, в противном случае директива сработает наоборот.

Директива #if.

В отличие от предыдущих директив, эта директива может управлять многими фрагментами текста.

Синтаксис директивы:

```
  #if константное_выражение
        1 фрагмент текста 1
#elif константное_выражение
        2 фрагмент текста 2
#elif константное_выражение
        3 фрагмент текста 3
…
#else
        фрагмент текста N
#endif
```

До компиляции будет допущен только один фрагмент текста, возле которого окажется выражение, дающее ненулевой результат, если просматривать выражения сверху вниз.

Директивы **#elif** и **#else** вместе со своими фрагментами текста могут не указываться.

константное_выражение – выражение, составленное из одних констант и операций. Часто используется логическое выражение. В качестве констант используются макроконстанты, значения которых конструкция и анализирует.

Вместо выражения может использоваться оператор препроцессора **defined**, с помощью которого можно проанализировать наличие определения **макро_имени**.

Синтаксис оператора:

```
defined(макро_имя)
```

Директива #error

Директива **#error** заставляет компилятор прекратить компиляцию и выдать сообщение об ошибке. Эта директива используется в основном для отладки. Форма директивы выглядит следующим образом:

```
#error [сообщение_об_ошибке]
```

сообщение_об_ошибке в двойные кавычки не заключается.

Директива #line

Директива **#line** изменяет содержимое **__LINE__** и **__FILE__**, которые являются зарезервированными идентификаторами в препроцессоре. В первом из них содержится

номер компилируемой в данный момент строки кода. А второй идентификатор — это строка, содержащая имя компилируемого исходного файла. В общем виде директива **#line** выглядит таким образом:

```
#line номер ["имя_файла"]
```

где **номер** — это положительное целое число, которое становится новым значением **__LINE__**, а необязательное **имя_файла** — это любой допустимый идентификатор файла, становящийся новым значением **__FILE__**. Директива **#line** в основном используется для отладки.

Директива #pragma

Директива **#pragma** — это определяемая реализацией компилятора директива, которая позволяет передавать компилятору различные инструкции. Возможности этой директивы и относящиеся к ней подробности должны быть описаны в документации по компилятору.

Операторы препроцессора # и

Имеется два оператора препроцессора: **#** и **##**. Они применяются в сочетании с директивой **#define** с параметрами.

Оператор **#**, который обычно называют оператором превращения в строку, превращает параметр, перед которым стоит, в строку, заключенную в кавычки. Ниже показан пример применения этого оператора:

```
#include <stdio.h>

#define str(s) #s

main(){
   printf( str(Мне нравится C) );
}
```

Препроцессор превращает строку

```
   printf( str(Мне нравится C) );
```

в

```
   printf( "Мне нравится C" );
```

Оператор **##**, который называют оператором склеивания, или конкатенации двух лексем. Пример использования:

```
#include <stdio.h>

#define concat(a,b)   a##b
```

```
main()
{
  int xy = 10;
  printf("%d", concat(x,y));
}
```

Препроцессор превращает строку

```
  printf("%d", concat(x,y));
```

в

```
  printf("%d", xy);
```

Другие зарезервированные имена в препроцессоре

__DATE__ - строка в виде **месяц/день/год**, то есть дата перевода исходного кода в объектный.

__TIME__ - время компиляции программы. Это время представлено строкой, имеющей вид **час:минута:секунда**.

__func__ - строка содержащая имя текущей функции. Добавлено в C11.

Задачи

1. Написать директиву препроцессора, которая реализует вставку собственного заголовочного файла с именем my.h.
2. Определить имя для константы 3.1415.
3. Создать именованные константы TRUE и FALSE для обозначения логических результатов.
4. Создать именованные константы, которые бы задавали размеры выводимой на экран таблицы.
5. Создать макрофункцию для вычисления квадрата числа.
6. Определить макрофункцию swap для обмена содержимого двух переменных целого типа через ИСКЛЮЧАЮЩЕЕ ИЛИ.
7. Создать две макрофункции для вычисления корней квадратного уравнения.
8. Создать одну макрофункцию для вычисления корней квадратного уравнения с параметрами a, b, c и sign (для обозначения знака + или -).
9. С помощью директив условной компиляции создать отладочный фрагмент, который будет выводить на экран значения переменных x и y, если хотя бы одна из них будет меньше нуля и выдавал номер строки в тексте программы, где это было обнаружено.
10. Создать именованную константу для обозначения ширины таблицы, значение которой зависело бы от количества строк в таблице следующим образом:

Количество строк	Количество столбцов
от 0 до 5	5
от 5 до 8	6
от 8 и далее	10

6. Массивы

Массив – это набор переменных одного типа.

Предназначены массивы для организации хранения и однотипной обработки набора информации.

Форма декларации массива:

```
тип_данных имя_массива[размер];
```

Квадратные скобки при декларации массива обязательные. В них в виде целого числа больше нуля, или константного целого выражения указывается количество переменных в массиве, называемое размером массива. В процессе работы программы размер такого массива остается неизменным. Переменные, входящие в состав массива, называют элементами массива.

В следующем примере показана декларация массива **array** типа **int** из 100 переменных.

```
int array[100];
```

Элементам в массиве присваиваются целые номера, называемые индексами. Нумерация элементов в массивах в языке Си всегда начинается с нуля и каждому

следующему элементу присваивается номер на единицу больше предыдущего. Индекс первого элемента всегда равен нулю, а индекс последнего всегда на 1 меньше размера массива.

Доступ к элементам массива реализуется с помощью операции **[]**, которая указывается после имени массива, а внутри квадратных скобок помещают индекс элемента массива. Например, оператор

```
array[5] = 0;
```

присваивает элементу с номером 5 массива **array** значение 0.

Кроме указания конкретного номера элемента массива, в квадратных скобках можно указать целое выражение, результат которого будет восприниматься программой как индекс элемента массива. В следующем примере показано, как обнулить все переменные в массиве **array**.

```
for(i=0; i < 100; i++)
   array[i] = 0 ;
```

Кроме операции **[]**, с массивами можно использовать операцию **sizeof**, результатом которой будет размер массива в байтах:

```
sizeof(array) == 400
```

В памяти массив занимает непрерывную область, в которой элементы располагаются в порядке индексации. Ниже показано расположение в памяти массива **array**, начинающегося с адреса 200, который называют адресом массива.

Элемент ->	0	1	2	3	...	n
Адрес ->	200	204	208	212	…	200+n*sizeof(int)

Адрес массива всегда совпадает с адресом самого первого байта элемента массива с номером 0, а благодаря непрерывному расположению элементов в массиве, адрес любого элемента можно вычислить, прибавив к адресу начала массива число, равное номеру элемента, умноженному на размер типа массива.

Для идентификации адреса массива в языке Си применяется имя массива:

```
array==&array
array==&array[0]
```

Во время выполнения программы не проверяется соблюдение границ массивов. Поэтому программист должен сам, где это необходимо, ввести проверку границ индексов.

Типовые задачи с массивами

Для обработки переменных в массивах используется цикл **for** и простые задачи с массивами реализуются с помощью одного простого оператора, указанного в цикле **for**.

1. Ввод чисел в массив

```
for(i=0; i<100;i++)
  scanf("%d",&array[i]);
```

Переменная **i** должна быть любого целого типа.

2. Вывод чисел из массива

```
for(i=0; i<100; i++)
  printf("%d, ",array[i]);
```

3. Подсчет суммы чисел в массиве

```
sum = 0; // Подсчет суммы массива
for(i = 0; i < 100; i++)
  sum += array[i];
```

Тип переменной **sum** должен позволить получить в этой переменной правильный результат.

В некоторых задачах выполнять простой оператор в цикле приходится в зависимости от условия, как показано ниже в задаче поиска.

4. Поиск максимального числа в массиве

```
max = array[0] ;
for(i = 1; i < 100; i++)
  if(max < array[i])     // Если встретилось большее число
     max = array[i] ;    // его нужно запомнить
```

Переменная **max** должна быть того же типа, что и элементы массива.

Реже, для реализации задачи с массивами приходится использовать два цикла. Это наблюдается в задачах, которые должны несколько раз повторить некоторую задачу для получения нужного решения. Ниже показана задача сортировки массива, которая использует задачу поиска минимального числа (второй цикл). Обнаружив минимальное число и положив его в начало массива, задача поиска минимального числа повторяется (первый цикл), но в ограниченной области от **i** до конца массива.

5. Сортировка массива

```
for(i = 0; i < 100-1; i++)    // Повторение поиска минимума
  for(j = i+1; j < 100; j++) // Поиск минимума
    if(array[i] > array[j])
    {
       c = array[i];          // Путем обмена меньшее
      array[i] = array[j];// закладывается на место с
      array[j] = c;          // номером i
    }
```

Переменные **i** и **j** должны быть любого целого типа. Переменная **c** должна быть того же типа, что и элементы массива.

Строки

Строка в языке Си – это набор символов, заканчивающийся символом с номером 0. При задании строки-константы, завершающий символ в конце строки всегда будет добавлен компилятором автоматически. Так, строка **"Hello World"**, содержит 11 символов, однако в памяти она будет занимать 12 байт, т. к. последним символом этой строки будет завершающий символ с номером 0.

Для организации хранения и обработки строк в языке Си специального типа данных нет, поэтому для этих целей используются массивы типа **char**. Ниже показана декларация такого массива.

```
char str[20] ;
```

В массиве **str** можно хранить набор целых чисел в диапазоне от -128 до 127, набор символов и строку, причем строка может содержать от 0 до 19 символов.

Под строки в универсальной кодировке используются массивы типа **unsigned short**, который в соответствии со стандартом можно обозначить идентификатором **wchar_t**.

```
wchar_t wstr[20] ;
```

Для обработки строк в языке Си имеется много пар стандартных функций для строк типа **char** и **wchar_t**.

1. Вывод строк на экран. Ниже показаны три варианта вывода строки, содержащейся в массиве **str**.

```
puts( str ) ;                          // 1
printf( str ) ;                        // 2
printf("Это строка: %s\n", str);       // 3
wprintf( wstr );                       // 4
wprintf(L"Это строка: %s\n", wstr); //  5
```

В первом и втором варианте функциям указывается адрес массива типа **char**, содержащего строку. Во втором варианте важно, чтобы в строке не было символа **%**. В третьем варианте вывод делается с использованием формата **%s**. Для вывода строк типа **wchar_t** можно использовать функцию **wprintf**.

2. Ввод строк с клавиатуры. Ниже показаны варианты получения строки и размещения ее в массиве **str**.

```
gets( str ) ; // 1 – получение всего текста
scanf("%s", str) ; // 2 – получение текста до первого
                   //     пробельного символа
wscanf(L"%s", wstr) ; // Под тип wchar_t
```

Функции завершают полученный фрагмент текста символом с номером 0.

3. Обработка строк. Ниже показаны часто используемые пары функций, описание которых находиться в файле **string.h** и **wchar.h**.

Функция	Выполняемое действие
strlen(str)	Возвращает длину строки в массиве **str**
wcslen(wstr)	
strcpy(str1, str2)	Копирует строку **str2** в массив **str1**
wcscpy(wstr1, wstr2)	
strcat(str1, str2)	Добавляет строку **str2** к концу строки в массиве **str1**.
wcscat(wstr1, wstr2)	
strcmp(str1, str2)	Сравнивает с учетом регистра пары символов, имеющих одни и те же индексы, в двух строках и возвращает целое число: **0** - если все пары символов совпадают, т. е. строки одинаковые **> 0** - если в **str1** попался символ с номером больше чем в **str2** **< 0** - если в **str1** попался символ с номером меньше чем в **str2**
wcscmp(str1, str2)	
strncpy(str1, str2, n)	Копирует первые **n** символов строки **str2** в массив **str1**
strncat(str1, str2, n)	Добавляет первые **n** символов строки **str2** к концу строки в массиве **str1**.
strncmp(str1, str2, n)	Сравнивает с учетом регистра первые **n** символов в строках **str1** и **str2**
strchr(str, ch)	Ищет символ **ch** в строке **str** и возвращает его адрес. В случае отсутствия в **str** указанного символа возвращает **0**.
strstr(str1, str2)	Ищет строку **str2** в строке **str1** и возвращает ее адрес. В случае отсутствия в **str1** указанной строки возвращает **0**.

Функции, название которых начинается на **wcs** предназначены для работы со строками в универсальной кодировке (тип **wchar_t**).

4. Преобразование строк. Описание этих функций находиться в файле **stdlib.h**.

Функция	Выполняемое действие
atoi(str)	Преобразует содержимое строки в целое число типа **int**.
atol(str)	Преобразует содержимое строки в целое число типа **long int**.
atoll(str)	Преобразует содержимое строки в целое число типа **long ling int**.
atof(str)	Преобразует содержимое строки в дробное число типа **double**.

Все функции рассматривают содержимое строки как представление числа в десятичной системе счисления и возвращают результат преобразования. Если преобразование невозможно, функции возвращают 0. Это означает, что в начале строки нет символов (цифр), которые можно трактовать как разряды числа.

Если на содержимое строки нужно посмотреть с точки зрения другой системы счисления, можно воспользоваться функциями: **strtol**, **strtoll**, **strtoul**, **strtoull**. В первом параметре указывается адрес строки, во втором параметре можно указать 0, в третьем параметре основание системы счисления.

5. Форматные функции для работы со строками. Описание этих функций находиться в файле **stdio.h**.

sprintf(str, форматная_строка [, список_информации]) - выводит информацию в строку **str**.

scanf(str, форматная_строка, список_адресов_переменных) – функция осуществляет разбор содержимого строки **str**, согласно форматам, указанным в форматной строке. Есть аналоги и под строки типа **wchar_t**: **swprintf** и **swscanf.**

Массивы переменной длины.

В C89 размер массива должен быть объявлен с помощью константных выражений. Поэтому компилятор C89 устанавливает фиксированный размер массива, не изменяющийся в процессе выполнения программы. Однако это не относится к C11, в котором определено новое мощное средство: массивы переменной длины. Стандарт C11 позволяет в объявлении размера массива использовать любые выражения, в том числе такие, значение которых становится известным только во время выполнения. Объявленный таким образом массив называется массивом переменной длины. Однако переменную длину могут иметь только локальные массивы (т.е. видимые в блоке). Пример:

```
int size=a+b;
double array[size];
array[0]=25;
```

Двухмерные и n-мерные массивы.

Массивы, которые рассматривались выше, называют одномерными массивами, т. к. каждый элемент такого массива определяется одним индексом, указываемым внутри пары квадратных скобок. Кроме одномерных массивов в языке Си можно использовать двухмерные и n-мерные массивы. Такие массивы декларируются с указанием соответствующего количества пар квадратных скобок и указанием в каждой паре размера.

```
int m2[5][10] ; // декларация двухмерного массива int
m3[8][5][10] ; // декларация трехмерного массива int
m4[3][8][5][10] ;// декларация четырехмерного массива
```

Двухмерный массив **m2** можно рассматривать как таблицу или матрицу, содержащую 5 строк и 10 столбцов. Трехмерный массив **m3** можно рассматривать как набор из 8 таблиц или матриц, содержащих 5 строк и 10 столбцов. Четырехмерный массив **m4** можно рассматривать как три набора из 8 таблиц или матриц, содержащих 5 строк и 10 столбцов.

Нумерация элементов в n-мерных массивах делается так же, как и в одномерных массивах, целыми числами начиная с нуля.

Доступ к переменным таких массивов реализуется с помощью операции **[]**, которая указывается после имени массива столько раз, сколько размерностей имеется в

массиве, а внутри квадратных скобок помещают индекс каждого элемента массива. Например, оператор

```
m2[3][6] = 0;
```

присваивает элементу массива **m2**, лежащему на пересечении 3-й строки и 6-го столбца значение 0.

Работа с двухмерными и n-мерными массивами осуществляется так же, как и с одномерными, с помощью циклов **for**. При этом количество циклов напрямую зависит от количества размерностей массива, если необходимо обработать все элементы массива. Ниже показан пример обнуления всех переменных двухмерного массива **m2**.

```
for(i = 0;i<5; i++)
  for(j = 0;j<10; j++)
    m2[i][j] = 0;
```

Поскольку при работе с n-мерными массивами используются вложенные циклы, в каждом цикле должна использоваться своя переменная целого типа.

В памяти двухмерные массивы располагаются по строкам. В начале массива всегда будут расположены переменные 0-й строки, затем 1-й и т. д. Трехмерные массивы располагаются по таблицам. В начале массива всегда будет располагаться таблица с номером 0, затем с номером 1 и т. д. Сами таблицы будут расположены по строкам.

Ниже показано расположение в памяти массива **m2**, начинающегося с адреса 400.

Строка № 0					Строка № 1				Строка № 2				Строка № 3				Строка № 4			
0	1	2	…	9	0	1	…	9	0	1	…	9	0	1	…	9	0	1	...	9
400	404	408	…	436	440	444	…	476	480	484	…	516	520	524	…	556	560	564	…	596

В средней строке показана нумерация столбцов, а в нижней - адрес каждой переменной массива.

Поскольку n-мерный массив в памяти располагается так же, как и одномерный, все задачи с n-мерными массивами могут быть реализованы и с помощью одномерных массивов. Только программист должен взять на себя расчет положения нужной переменной, который выполняют операции **[].** Ниже показана формула для расчета положения переменной в двухмерном массиве, содержащем 10 столбцов:

адрес_переменной = адрес_массива + (номер_строки * 10 + номер_столбца) * размер_переменной.

для трехмерного массива, содержащего таблицы 5 x 10 формула будет выглядеть следующим образом:

адрес_переменной = адрес_массива + (номер_таблицы * 50 + номер_строки * 10 + номер_столбца) * размер_переменной.

Эти формулы могут пригодиться при организации в рамках одномерного массива таблиц с переменным количеством строк и столбцов или переменного количества таблиц.

Существует и обратная возможность, а именно, с любым n-мерным массивом, можно работать как с одномерным массивом. Так двухмерный массив можно представить, как таблицу, содержащую одну строку с номером 0, а трехмерный массив можно представить, как одну таблицу с номером 0, содержащую одну строку с номером 0. При этом надо использовать диапазон нумерации столбцов от 0 до количества переменных в массиве. Ниже показано, как обнулить все переменные массива **m2** с помощью одного цикла

```
for(i = 0;i<50; i++)
  m2[0][i] = 0;
```

При работе с двухмерными и n- мерными массивами после имени массива разрешается указывать не все пары квадратных скобок. При этом такие указания будут компилятором трактоваться как адреса того, что задают оставшиеся пары квадратных скобок. Ниже показаны различные варианты для трехмерного массива:

m3[5][3][6] – переменная таблицы с номером 5, находящаяся на пересечении строки с номером 3 и столбца с номером 6 массива **m3**;

m3[5][3] – адрес строки с номером 3 в таблице с номером 5 массива **m3**; **m3[5]** – адрес таблицы с номером 5 массива **m3**; **m3** – адрес самого массива **m3**.

Благодаря этим возможностям двухмерный массив типа **char** можно рассматривать как массив строк, а адрес нужной строки будет получен путем указания ее номера в одной паре квадратных скобок. Ниже показан пример вывода строк из массива **ms** на экран:

```
char ms[10][100] ; // Массив для хранения 10 строк каждая
                   // размером не более 99 символов
…
for(i=0;i<10; i++)
  puts( ms[i] );
```

Инициализация массивов

В языке Си все, что можно декларировать, все можно инициализировать.

Инициализация массивов чем-то похожа на инициализацию обычных переменных, но отличается тем, что для массива приходится указывать много значений, которые необходимо перечислить после знака "равно" через запятую в фигурных скобках. Ниже показаны примеры инициализации массивов различных размерностей:

```
int m[5] = {5,8,7,9,2} ;
int m2[3][4] = {3,6,5,8,7,9,1,3,2,7,6,8};
int m3[2][2][3] = {3,6,5,8,7,9,1,3,2,7,6,8};
```

Распределение значений по переменным массива будет сделано согласно порядку размещения этих переменных в памяти.

Значения, указанные в фигурных скобках, называют списком инициализации. При задании этих списков важно, чтобы количество указываемых значений не превышало количество переменных в массиве. Минимум в списке инициализации можно указать одно значение.

При использовании неполных списков инициализации, распределение значений будет таким же. Первое значение будет заложено в первую переменную, второе – во вторую и т.д. Те переменные, которым из списка инициализации ничего не достанется, будут проинициализированы нулем – это правило неполных списков. Поэтому, если необходимо проинициализировать нулем все переменные массива, а их может быть много, достаточно в списке инициализации указать одно число 0.

При инициализации двухмерных и n-мерных массивов лучше воспользоваться пробельными символами и расположить числа в списке инициализации в виде таблицы или набора таблиц, как показано ниже:

```
int m2[3][4]={
 3,6,5,8,
 7,9,1,3,
 2,7,6,8};
```

При таком расположении лучше видно, какие значения, в какие переменные массива попадают.

При задании полного списка инициализации размер массива можно не указывать. В этом случае компилятор сам определит размер такого массива. Разрешается не указывать только крайний левый размер или единственный. Ниже показаны такие примеры инициализации, которые удобны для массивов, размеры которых, при работе над программой могут меняться:

```
int m[] = {5,8,7,9,2} ;
int m2[][4] = {3,6,5,8,7,9,1,3,2,7,6,8} ;
int m3[][2][3]={3,6,5,8,7,9,1,3,2,7,6,8} ;
```

При инициализации двухмерных и n-мерных массивов, значения для отдельных элементов этих массивов (строк, таблиц, ...) можно заключать в пары фигурных скобок. Этот вариант удобен в том случае, когда для некоторых элементов необходимо указать неполные списки инициализации:

```
int m2[3][4] =
 {{3,6,5,8},
  {7,9},
  {2,7,8}};
```

Массивы типа **char**, предназначенные для хранения строк, могут инициализироваться как обычные массивы:

```
char s[10] = {72,101,108,108,111};// Hello
char s2[10]={'H','e','l','l','o'};
```

Оба массива инициализируются строкой, содержащей слово **"Hello".** Переменные, которым из списка ничего не достанется, будут проинициализированы 0, что хорошо для строк. Однако, самый удобный способ инициализации таких массивов – инициализация строкой, как показано ниже:

```
char s3[10] = "Hello";
```

Массивы строк инициализируют списком строк:

```
char ms[5][10] = {"if", "for", "while", "switch", "break"} ;
```

В C11 появился новый синтаксис инициализации массивов, который позволяет указывать индекс инициализируемого элемента, причём все пропущенные элементы будут равны нулю. Пример:

```
int d1[10]={[1]=1, 2, [4]=3, 5,6,7};
int d2[] = {[0]=1,[5]=-1,[14]=3};
```

Важно: массивы переменной длины инициализировать нельзя.

Задачи

1. Написать программу, которая вводит с клавиатуры в одномерный массив 5 целых чисел, после чего выводит количество ненулевых элементов.
2. Написать программу, которая вычисляет среднее арифметическое ненулевых элементов введенного с клавиатуры массива целых чисел.
3. Написать программу, которая вычисляет среднее арифметическое элементов массива без учета минимального и максимального элементов массива.
4. Написать программу, которая проверяет, находится ли введенное с клавиатуры число в массиве. Массив должен вводиться во время работы программы.
5. Написать программу, которая проверяет, представляют ли элементы введенного с клавиатуры массива возрастающую последовательность.
6. Написать программу, которая определяет, сколько раз введенное с клавиатуры число встречается в массиве.
7. Написать программу, которая определяет, сколько различных чисел находится в массиве.
8. Написать программу, которая проверяет, есть ли во введенном с клавиатуры массиве элементы с одинаковым значением.
9. Написать программу, которая определяет количество людей, чей рост превышает средний.
10. Написать программу, которая вводит по строкам с клавиатуры двумерный массив и вычисляет сумму его элементов по столбцам.
11. Написать программу, которая определяет номер строки квадратной матрицы (двухмерного массива), сумма элементов которой максимальна.

12. Напишите программу, которая вычисляет длину введенной с клавиатуры строки.
13. Написать программу, которая во введенной с клавиатуры строке преобразует строчные буквы английского алфавита в прописные.
14. Написать программу, которая удаляет из введенной с клавиатуры строки начальные пробелы.
15. Написать программу, которая проверяет, является ли введенная с клавиатуры строка целым числом.
16. Написать программу, которая проверяет, является ли введенная с клавиатуры строка двоичным числом.
17. Написать программу, которая проверяет, является ли введенная с клавиатуры строка шестнадцатеричным числом.

7. Функции

Функция – самостоятельная задача в программе, оформленная определенным образом. В языках структурного программирования функция – это максимальный строительный элемент программы.

Функции дают определенные преимущества при создании программ:

- уменьшение текста программы и самой программы. Достигается это за счет оформления повторяющихся задач в виде функций;
- упрощение решения сложных задач за счет разбиения задачи на более мелкие и оформления этих задач в виде функций;
- упрощение переноса задач в другие программы. Достигается это путем копирования текста функции в другую программу или создания библиотеки функций.
- При использовании функций имеется и некоторый недостаток – уменьшение скорости работы программы.

Форма определения функции:

```
тип_результата имя_функции([список_параметров])// заголовок
{
   декларация переменных                          // тело функции
   операторы решения задачи
}
```

тип_результата – тип возвращаемого функцией значения, которое будет получаться при решении задачи, вложенной в функцию. Если функция не будет возвращать результат, то на этом месте указывается ключевое слово **void** – функция без результата. Такую функцию нельзя использовать в выражениях. Тип результата можно пропустить, в этом случае будет считаться, что функция должна вернуть результат типа **int**.

имя_функции – любое имя, заданное по правилам языка Си.

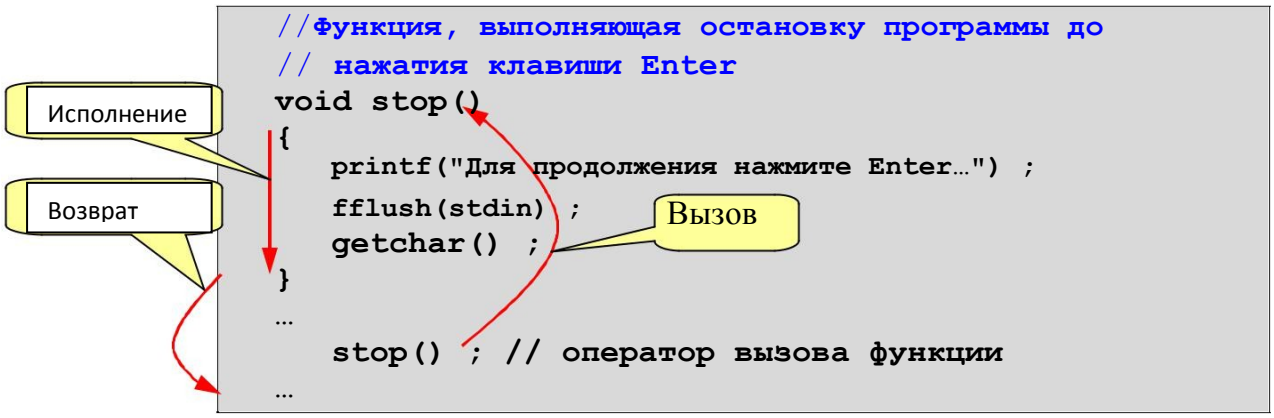
список_параметров – список переменных, через которые функция получит исходную информацию, необходимую для решения задачи. Перед именем каждой переменной необходимо указать ее тип данных. Параметры функции часто называют формальными параметрами (аргументами), т. к. они определяют тип данных, порядок и количество передаваемой в функцию информации, но не саму информацию. При вызове функции ей указываются фактические параметры (аргументы). Если параметры не используются, вместо них рекомендуют ставить ключевое слово **void**.

Первую строку в определении функции называют заголовком функции. За заголовком должно следовать тело функции, начинающееся с открывающей фигурной скобки.

Определение функции можно делать в любом месте текста программы, но не внутри других функций. При создании многомодульных программ – программ, текст которых

расположен в нескольких файлах, определение функции должно быть дано в одном файле.

Пример простой функции и её вызова:




```
//Функция, выполняющая остановку программы до
// нажатия клавиши Enter
void stop()
{
   printf("Для продолжения нажмите Enter…") ;
   fflush(stdin) ;
   getchar() ;
}
…
   stop() ; // оператор вызова функции
…
```

Работа любой функции происходит в три этапа:

1. Вызов – происходит, когда программа встретит оператор вызова функции, являющийся одним из операторов перехода. Перед вызовом в параметры функции будет скопирована информация, указанная в круглых скобках в операторе вызова.
2. Исполнение – выполнение операторов функции.
3. Возврат – переход на оператор, идущий следом за оператором вызова функции. Возврат происходит после выполнения всех операторов в функции, когда программа выйдет на закрывающую фигурную скобку функции.

Оператор return

Оператор **return** – оператор перехода, который реализует возврат из функции. Любая функция должна заканчиваться оператором **return**. Если его не указать, он будет встроен компилятором автоматически.

У оператора **return** две формы использования:

```
return; // 1
```

Эта форма используется для **void** функций (функций без результата) для преждевременного завершения работы таких функции.

```
return выражение; // 2
```

Такая форма используется в функциях, которые должны возвращать результат. Для этого в операторе **return** указывается выражение, результат которого будет являться результатом работы функции. Исполнение операторов в функции, возвращающей результат, должно заканчиваться оператором **return**.

Операторов **return** в функции может быть много. Пример функции, возвращающей результат:

```
// Функция, подсчитывающая сумму целых чисел в диапазоне
// от a до b
int summa(int a, int b) {
   int sum=0;//Обнуляем переменную, созданную под результат
   if(a > b) {
        a= a ^ b; // Алгоритм обмена
        b = a ^ b; // через
        a = a ^ b; // ИСКЛЮЧАЮЩЕЕ ИЛИ
   }
   while(a <= b) sum+=a++ ; //Подсчет суммы
   return sum ; //Возврат результата
}
. . .
   y = summa(1, x) ; // оператор вызова функции
```

Прототип функции

Для вызова функции, компилятор организует передачу в нее исходной информации, которая указывается в операторе вызова. Этой информацией заполняются параметры функции. Если тип передаваемой информации не соответствует типу параметра функции, компилятор будет вынужден осуществить преобразование передаваемой информации к типу параметра. Преобразование информации может происходить и при приеме результата.

Если компилятору функция не известна, то передача информации осуществляется согласно типу этой информации, а в качестве типа результата принимается тип **int**. Такое поведение компилятора может быть неприемлемым при работе со многими функциями, а такая ситуация может наблюдаться, когда вызов функции будет расположен выше ее определения, или определение будет сделано в другом файле. Поэтому, в таких случаях, компилятор должен быть информирован о типе результата и о типах параметров. Делается это с помощью прототипа – заголовка функции, скопированного в нужное место и закрытого точкой с запятой. Иногда прототип называют объявлением функции.

Пример прототипа функции, показанной выше:

```
int summa(int a, int b);
```

Названия параметров в прототипе можно пропустить, в этом случае прототип будет выглядеть следующим образом:

```
int summa(int, int);
```

Прототип функции, получающей одномерный массив при вызове:

```
int solve(int data[]);
```

при этом остаётся неясным, а сколько элементов массива **data** будет использоваться при вызове данной функции. Стандарт C11 позволяет задать в прототипе функции количество используемых элементов. Например, объявление функции, которая будет обращаться как минимум к одному элементу массива выглядит так:

```
int solve(int data[static 1]);
```

Прототипы функций можно располагать только в местах декларации переменных. Типичное расположение прототипов – в начале файла программы или в собственном заголовочном файле.

Рекурсия

Рекурсия – это вызов функции самой себя. Ниже показан пример прямой рекурсии:

```
int func()
{
   …
   func() ; // Вызов самой себя
   …
}
```

Рекурсия образует цикл, который начнет завершаться, когда очередной вызов функции пропустит вызов самой себя.

Рекурсивный цикл бесконечно выполняться не может, поскольку на каждый вызов функции тратиться память в специальной структуре программы, называемой программным стеком.

Поскольку рекурсия образует цикл, то ее можно применять для всех задач, связанных с повторениями. Ниже показана функция, вычисляющая факториал числа – результат перемножения целых чисел от 1 до самого числа:

```
int factorial(int n)
{if(n < 0) return 0 ;
 if(n == 0) return 1 ;
return n * factorial(n-1); // Вызов самой себя
}
```

В простых задачах, связанных с повторениями, рекурсия может усложнить понимание решения задачи, однако рекурсию удобно использовать в задачах, при решении которых может возникать надобность в решении той же задачи на основе новой информации. Примером такой задачи может быть задача прохода по древовидной структуре данных.

Другим примером задач, удобных для рекурсии, могут служить задачи, в результате решения которых будет получаться информация, нуждающаяся в обработке в порядке, обратном порядку получения. К такой задаче относится задача представления числа в нужной системе счисления, решение которой показано ниже:

```
// Функция вывода числа n в системе счисления с основанием
// base
void print(int n, int base)
{
  int i;

  if(n < 0)   // Определение знака в числе

  {
      putchar ('-'); n = -n;
  }

  if((i = n/base)!=0) // Результат деления на основание

     print(i,base); // Вызов самой себя с передачей результата

  putchar(n % base + '0');// Получение остатка от деления
}
```

В функции **print** используется способ получения значений разрядов числа в нужной системе счисления путем целочисленного деления числа и получаемых результатов на основание той системы счисления, куда делается переход. При каждом делении определяется остаток от деления, который и будет равен значению разряда. Получение разрядов происходит, начиная с младшего разряда, а вывод необходимо делать наоборот, начиная со старшего разряда. Ниже показан порядок вызовов функции **print** при выводе числа 172 в десятичной системе счисления.

```
print(172, 10) Первый
вызов 172 -> n
172/10 = 17 -> i
     print(17, 10) Второй
     вызов 17 -> n
     17/10 = 1 -> i
          print(1, 10) Третий
          вызов 1 -> n
          1/10 = 0 -> i
               Четвертый вызов пропускается
          1%10 = 1+'0' = '1' -> на экран (Третий вызов)
     17%10 = 7+'0' = '7' -> на экран (Второй вызов)
172%10 = 2+'0' = '2' -> на экран (Первый вызов)
```

Макрофункции, не зависящие от типа

Достаточно часто при написании функций они могут отличаться только типом входных параметров, например, функции, которые находят среднее арифметическое имеют вид:

extern inline int avrInt(int a, int b) { return (a+b)/2; }
extern inline float avrFloat(float a, float b) { return (a+b)/2; }

extern inline long avrLong(long a, long b) { return (a+b)/2; }
extern inline double avrDouble(double a, double b) { return (a+b)/2; } и т.д.

Так как язык Си не поддерживает перегрузку функций, то все функции должны иметь различные имена и, следовательно, при изменении типа переменной придётся изменять и имя вызываемой функции на другое. Решения этой проблемы есть в стандарте C11. Используя макрофункцию и ключевое слово **_Generic(**контролирующее выражение**)**, мы можем поручить компилятору выбирать правильные функции в зависимости от типа контролирующего выражения. Например, для вычисления среднего арифметического можно создать следующую макрофункцию:

```
#define avrg(X,Y)       \
     _Generic((X)+(Y), \
     int: avrInt,       \
     float: avrFloat,   \
     long: avrLong,     \
     default: avrDouble)((X),(Y))
. . .
int main(void) {
     long a=3;
     int  b=7;
     long avr=avrg(a,b);
. . .
}
```

Порядок работы приведённого примера следующий:

- вычисляется тип контролирующего выражения (a)+(b);
- если результирующий тип есть в списке макрофункции, то компилятор подставляет соответствующее имя функции, указанное после **имени типа:**
- если нет, то имя функции берётся после **default:**
- при запуске программы будет вызвана функция **avrLong()**.

Задачи

1. Написать функцию, которая возвращает максимальное из двух целых чисел, полученных в качестве аргумента.
2. Написать функцию, которая сравнивает два целых числа и возвращает результат сравнения в виде одного из знаков: >, < или =.
3. Написать функцию, которая вычисляет доход по вкладу. Исходными данными для функции являются: величина вклада, процентная ставка (годовых) и срок вклада (количество дней).
4. Написать функцию, которая возвращает 1, если символ, полученный функцией в качестве аргумента, является гласной буквой русского алфавита, и ноль — в противном случае.
5. Написать функцию, которая возвращает 1, если символ, полученный функцией в качестве аргумента, является согласной буквой русского алфавита, и 0 — в противном случае.
6. Написать функцию, возводящую число в целую положительную степень с помощью рекурсии.

8. Классы памяти

С появлением в программе собственных функций, может возникнуть надобность в использовании различных видов переменных, с которыми в языке Си тесно связаны два понятия:

1. Область видимости – это место в программе, где можно использовать переменные. Если переменные декларируются в разных областях видимости, им могут быть присвоены одинаковые идентификаторы. В одной области видимости имена переменных должны быть разными.
2. Время жизни – промежуток времени, в течение которого существует переменная.

Все переменные в программе, подразделяются на два основных вида, называемые классами памяти.

Первый вид широко используемых переменных – локальные переменные. Это переменные, которые декларируются внутри блока, помечаемого фигурными скобками. Такие переменные еще называют автоматическими переменными, из-за своего поведения. Они автоматически появляются (создаются), когда нужны программе и автоматически исчезают (удаляются), когда больше не нужны. Надобность в них определяется ходом выполнения программы. При входе программы в блок, отмеченный фигурными скобками, если в блоке имеются продекларированные переменные, они будут созданы. Когда программа будет покидать блок, произойдет удаление созданных в блоке переменных. Таким образом, время жизни локальных переменных определяется временем нахождения программы блоке, а область видимости – область блока.

В связи с автоматическим созданием, локальные переменные не инициализируются без явного указания инициализации, чтобы не тратить на это время. А поскольку они удаляются, и исчезает информация, которую они хранят, их используют только для временного хранения информации на момент решения некоторой задачи, обычно в рамках функции.

Кроме локальных переменных в языке Си можно применять глобальные переменные – это переменные, которые декларируются за рамками функций. Живут глобальные переменные ровно столько, сколько живет программа и использовать их можно в любом месте программы. Такие переменные используются для хранения общей информации для всей программы или большой ее части.

Все глобальные переменные находятся в одной области видимости, поэтому имена глобальных переменных в программе должны быть разными и не совпадать с именами функция, поскольку функции так же находятся в глобальной области видимости.

Глобальные переменные создает компилятор при компиляции текста программы, поэтому он их автоматически инициализируют нулем.

В программах, по возможности, необходимо избегать использования глобальных переменных, т. к. значение такой переменной может быть изменено функциями программы, даже когда программист этого не ожидает. Это может приводить к тяжело отслеживаемым ошибкам в программе.

Для уточнения класса памяти и получения других видов переменных, в языке Си существует четыре спецификатора класса памяти: **`auto`**, **`register`**, **`static`** и **`extern`**.

Используются они при декларации переменных по следующей форме:

`[класс_памяти] тип_данных список_переменных`;

Спецификатор **`auto`** может использоваться при декларации локальных перемененных, для того, чтобы явно подчеркнуть их автоматическое поведение.

Первоначально спецификатор класса памяти **`register`** применялся только к локальным целым переменным и параметрам функций. Однако стандарт языка Си расширил использование спецификатора **`register`**, теперь он может применяться к переменным любых типов и массивам, но по-прежнему, только к локальным.

В первых версиях компиляторов языка Си спецификатор **`register`** сообщал компилятору, что переменная должна храниться в регистре процессора, а не в оперативной памяти, как все остальные переменные. Это приводит к тому, что операции с регистровой переменной осуществляются намного быстрее, чем с обычными переменными, поскольку такая переменная уже находится в процессоре и не нужно тратить время на выборку ее значения из оперативной памяти (и на запись в память).

В настоящее время определение спецификатора **`register`** существенно расширено.

Стандарты C89 и C11 попросту декларируют "доступ к объекту так быстро, как только возможно".

Ниже приведен пример использования регистровых переменных, в объявлении которых указан спецификатор **register.** Эти переменные используются в функции возведения целого числа **m** в степень **e**.

```
int pwr(register int m,  register int e)
{
  register int result;

  result = 1;

  for(; e; e--) result *= m ;

return result;
}
```

В этом примере к переменным **e**, **m** и **result** применен спецификатор **register** потому, что они многократно используются для решения задачи внутри цикла.

При использовании регистровых переменных запрещается применять к ним операцию взятия адреса, т. к. адрес имеют только те переменные, которые находятся в оперативной памяти.

Спецификатор **static** можно применять как для глобальных, так и для локальных переменных.

Для глобальных переменных спецификатор **static** сужает область видимости таких переменных до рамок файла, в котором они декларируются. Статические глобальные переменные могут быть полезны для того, чтобы избежать конфликтов имен в глобальной области при создании программы несколькими программистами, а также для создания функций (библиотеки функций), использующих глобальные переменные, которые планируется задействовать в разных программах.

Для локальных переменных спецификатор **static** продлевает время жизни такой переменной до конца жизни программы. Статические локальные переменные не удаляются и в связи с этим не теряют хранимую информацию. Поэтому, когда для реализации задачи, в рамках функции необходимо сохранять информацию от одного вызова функции к другому, можно воспользоваться глобальными переменными, но лучше задействовать статические локальные переменные, если это возможно. Ниже показана функция **random**, генерирующая псевдослучайные числа. Получение нового псевдослучайного числа происходит путем вычисления его от предыдущего числа, которое хранится в локальной статической переменной **next**. Поскольку при первом запуске функции предыдущего числа не существует, функция **random** получает это число от функции **time**, но это делается один раз, за счет применения второй статической переменной **start**, которая инициализируется числом 1, для того, чтобы сработал оператор if, в котором ее значение меняется на 0.

```
int random()
{
   static int next, start=1;
   if(start)
   {
        next=time(0);
        start = 0;
   }
   next = next * 1103515245 + 12345;
   return(unsigned int)(next/65536)%32768;
}
```

Спецификатор **extern** используется для описания глобальных переменных, которые декларируются в других файлах.

Компилятор последовательно обрабатывает файлы программы и при работе над одним файлом, он ничего не знает, что содержится в других файлах. Поэтому он всегда будет выдавать ошибки, когда встретит использование глобальных переменных без предварительного их описания.

Ниже показан пример использования спецификатора **extern**.

```
            Файл 1
extern int a; // Описание

int main()
{
    printf("a = %d\n", a) ;
return 0;
}
```

```
            Файл 2
int a = 25; // Декларация
void func()
{
    …
    a = …
    …
}
```

Кроме спецификаторов классов памяти, при декларации переменных можно использовать квалификаторы **const** и **volatile**, указываемые перед типом переменной.

Квалификатор **const** позволяет получить в программе неизменяемые переменные, или по-другому их можно назвать именованными константами, значение которым можно задать только в момент их декларации.

Квалификатор **volatile** указывает компилятору на то, что значение переменной может быть изменено другой программой или задачей, которые будут одновременно выполняться с компилируемой программой. Знание таких подробностей важно для компилятора потому, что большинство компиляторов Си автоматически оптимизируют некоторые выражения, предполагая при этом неизменность значения переменной, если она не встречается в левой части оператора присваивания. В этом случае при очередной ссылке на переменную может использоваться ее предыдущее значение, заложенное в регистр процессора, а не текущее. А это может привести к ошибке, если в выражении присутствует переменная, значение которой может измениться извне. Квалификатор **volatile** предотвращает такое поведение компилятора, заставляя его организовывать программу таким образом, чтобы она каждый раз извлекала значение из переменной, когда оно потребуется.

Квалификатор **restrict,** появившийся в стандарте C11, применяется только к указателям и позволяет сообщить компилятору, что данный указатель является единственным способом, с помощью которого можно обращаться к данному объекту. Эта информация даст возможность компилятору сгенерировать более эффективный код. Чаще всего квалификатор **restrict** используется с параметрами функций, например,

```
void update(int *restrict ptrA, int *restrict ptrB) {...}
```

Также в стандарте языка C11 был добавлен квалификатор **_Atomic**. Пример такого объявления:

```
long _Atomic count=0;
```

С объявленной переменной **count** можно выполнять любые операции, определённые для указанного типа, но они будут реализованы компилятором, как атомарные операции, т.е. при наличии в системе нескольких потоков исполнения кода и обращении из этих потоков к переменной **count** эти обращения не преведут к состоянию гонки (race condition).

9. Указатели

Указатель – переменная адресного типа, содержимое которой воспринимается как адрес некоторой информации в памяти. Для получения указателя, его необходимо продекларировать, указав перед именем такой переменной знак астериска – «*****»:

```
int a, mas[10], *p ;
```

a – обычная переменная, **mas** – массив из 10-ти переменных, **p** – указатель.

Тип данных при декларации обычных переменных и массивов определяет, какого вида информация будет храниться в этих переменных. При декларации указателя тип данных определяет вид информации, адрес которой будет храниться в указателе.

Операции с указателями

Несмотря на то, что содержимое указателя является целым числом без знака, с указателями нельзя выполнять все операции, которые выполнимы с целыми числами, а только те, которые имеют некоторый смысл при работе с указателями.

Первая операция, с которой обычно начинают работу с указателями, это операция присвоения значения, с помощью которой в указатель можно заложить адрес информации. При этом компилятор будет следить за тем, чтобы справа от этой операции был указан адрес информации, соответствующей типу указателя.

```
p = &a;       // 1
p  = mas;     // 2

p  = 0x100  ; // 3
```

В первом случае в указатель закладывается адрес переменной **a**, во втором случае адрес начала массива (адрес самой первой переменной массива). И то и другое совпадает с типом указателя. В третьем случае компилятор может выдать предупреждение или ошибку, так как константа **100** относится к типу **int**, но не является адресом информации этого типа. В таких случаях нужно применить операцию приведения к типу указателя, как показано ниже:

```
p = (int *)0x100; // 3
```

Тип указателя – это тип, с которым он декларировался и звездочка, которая отличает указатели от обычных переменных при декларации.

Указателю можно задать нулевое значение, присвоив ему 0.

```
p = 0;
```

Для обозначения числа 0 при работе с указателями во многих заголовочных файлах языка Си определяется макроконстанта **NULL**. Считается, что если указатель содержит такое значение, то он не готов к работе. Поэтому часто можно встретить декларацию указателя и инициализацию его значением **NULL**, для того, чтобы впоследствии определить, был ли заложен в указатель реальный адрес или нет.

Вторая операция, ради которой и используют в программах указатели, это операция *****, реализующая доступ к информации, адрес которой хранится в указателе. Ее любят называть операцией разыменовывания указателя.

```
p  =  &a  ;//1
*p  =25;  //2
a =25; // 3
```

Второй и третий операторы приводят к одному и тому же, в переменную **a** будет заложено значение 25.

С указателями можно применять шесть операций, с помощью которых к указателю можно прибавить, или вычесть с него целое число. При работе с указателями их называют операциями адресной арифметики, поскольку они работают не так, как с числами. К ним относятся следующие операции: **+**, **-**, **++**, **--**, **+=** и **-=**. Эти операции применяются, когда указатель содержит адрес массива, и дают возможность вычислять адреса нужных элементов массива, не обращая внимания на тип массива.

```
p = mas;
*p = 5  ; // 1
```

```
p++; или  p += 1; или p = p + 1; // 2
*p = 5  ; // 3
```

В примере, показанном выше, первый оператор занесет число 5 в переменную массива
с номером 0. После выполнения одного из операторов, указанных во второй строке, третий оператор приведет к изменению значения переменной массива с номером 1, т. к. прибавление целого числа к содержимому указателя приведет к изменению его значения на число, умноженное на размер типа, с которым декларировался указатель. Сделано это для того, чтобы через указатель было удобно работать с массивами. Поэтому, прибавление к указателю числа 1, приведет к получению адреса следующей переменной в памяти.

Вместо операции *****, для получения доступа к элементам массива, адрес которого содержит указатель, можно применять операцию **[]**, которая с указателями будет работать так же, как и с массивами. Можно и наоборот, к массиву применить операцию ***.**

```
p = mas;
p[3] = 5 ;        // 1
*(mas+3) = 5;     // 2
```

Эти операторы приведут к одному и тому же, в элемент массива с индексом 3 будет занесено число 5. Однако с массивами принято использовать квадратные скобки, а с указателями операцию *****.

Кроме адресной арифметики, с указателями можно применять все шесть операций от-ношений: **>**, **>=**, **<**, **<=**, **==** и **!=**, а также логическое НЕ. При этом компилятор разрешает сравнивать между собой адреса однотипной информации.

Еще одна операция, разрешенная с указателями, это операция приведения к типу, которая может быть использована для получения содержимого указателя в виде целого числа и наоборот и для получения адреса на другой тип информации. Так, в примере ниже показано, как получить доступ к первому байту переменной **a** и изменить его значение:

```
p = &a;
*((char*)p) =0;
```

Это может использоваться для работы с отдельными байтами или группами байт переменных, а также для работы с разнотипной информацией через один указатель.

Указатели можно измерять операцией **sizeof**, которая даст его размер в байтах. При этом размер указателя никогда не зависел от типа указателя, а определяется архитектурой ЭВМ, а именно, принятыми вариантами адресации ячеек памяти. Так в 32-х разрядных ЭВМ, типа IBM PC, размер любого указателя равен 4 байта, что определяется размером шины адреса, она 32-х разрядная. Несмотря на то, что в 16-разрядных ЭВМ, типа IBM PC, шина адреса была 20-ти разрядная, размеры указателей составляли 2 байта (ближние указатели) и 4 байта (дальние указатели). Это объяснялось применением соответствующей архитектуры оперативной памяти в этих ЭВМ, которая разбивалась на сегменты размером 64 Кб, и в рамках одного сегмента можно было работать с помощью ближних указателей. Для работы с любым участком памяти приходилось использовать дальние указатели, в которых хранился номер сегмента, и смещение от начала сегмента до начала информации.

Поскольку указатель является переменной, находящейся в оперативной памяти, к нему применима операция взятия адреса **&**. Но для того, чтобы положить адрес указателя на хранение, понадобиться переменная, продекларированная с тем же типом данных, но двумя звездочками.

```
int **pp;
…
pp = &p;
```

Такие переменные называют указателями на указатели. Они предназначены для хранения адресов других указателей и работы с этими указателями, а так же с информацией, на которую они указывают.

```
*pp = &a; // 1
**pp = 32; // 2
```

Если в **pp** содержится адрес указателя **p**, то первый оператор приведет к занесению в указатель **p** адреса переменной **a**, а второй оператор приведет к изменению значения переменной **a**.

Язык Си разрешает декларацию указателя и с большим количеством звездочек, однако широко применяются простые указатель, декларируемые с одной звездочкой.

При декларации указателей можно использовать ключевое слово **void** при указании типа указателя.

```
void *pv;
```

Такие указатели предназначены для хранения любых адресов, однако не позволяют работать с информацией в памяти, так как программе неизвестно, с точки зрения какого типа данных нужно рассматривать информацию, находящуюся по адресу, который будет содержать такой указатель.

```
pv=&a;
pv=&p;
pv=&pp;
*pv = 0; // Ошибка, обращение к содержимому ячейки памяти
```

Кроме того, указатели могут декларироваться со словом **const**.

```
const int *pa;
```

Такие указатели запрещают изменение содержимого ячеек памяти, адреса которых они будут содержать.

Применение указателей

Правильное понимание и использование указателей имеет большое значение в языке Си, т. к. дает возможность создавать более эффективные программы. Обычно указатели используются в следующих случаях:

1. Для повышения эффективности работы функций.
   - Для передачи больших объемов информации в функции, т. к. в этом случае вместо передачи самой информации в функцию передается ее адрес в оперативной памяти. В связи с этим, в языке Си различают два варианта передачи в функцию информации. Первый – передача информации по значению, при этом происходит копирование указанной в вызове функции информации в параметры функции. Второй вариант – передача по адресу, когда функции указывается адрес информации, при этом в параметр функции происходит копирование ука-
     занного адреса, которые намного меньше занимает места, нежели сама информация. Для приема адреса у функции должен быть параметр указатель. В практике, когда объем информации превышать 16-20 байт, такую информацию стараются передавать в функцию по адресу.
   - Для получения из функции нескольких результатов, т. к. оператором **return** функция может вернуть только один результат. В этом случае действуют по принципу функции **scanf**, которая может вернуть сколько угодно результатов, зная адреса переменных, куда она должна положить информацию, получаемую от пользователя.
2. Для организации в памяти различных динамических структур, предназначенных для хранения и обработки информации. К таким структурам относятся: динамические массивы, линейные списки, построенные на динамических массивах, связанные списки и др.

Несмотря на то, что задачи, перечисленные в первом пункте, могут быть решены с помощью глобальных переменных, в глобальную область выходят только в том случае, если другие пути получаются неэффективными и усложняют построение программы.

Ниже показан пример использования указателя для передачи в функцию массива типа **double**. Для этих целей в функции создают два параметра. Один параметр – указатель, который будет содержать адрес начала массива, другой параметр – переменная целого типа, которая получит размер массива или количество обрабатываемых переменных, начиная с указанного адреса.

```
// Подсчет суммы чисел в массиве
double sum_mas(double *pm, int size)
{
   int i;
   double sum = 0;
   for(i=0; i< size; i++)
        sum += *(pm + i) ;
return sum;
}
…
   double m[10], s ;
   s = sum_mas(m, 10) ; // Вызов функции
```

адрес массива

размер массива

При вызове функции в первом параметре необходимо указать адрес начала массива типа **double**, во втором размер этого массива. Вместо адреса начала, можно указать адрес любой переменной массива, а вместо размера - количество переменных, которые необходимо обработать, начиная с указанного адреса.

Ниже показаны еще два варианта описания параметров функции для передачи в нее массива.

```
double sum mas(double pm[], int size)   // 1
double sum_mas(double pm[10])           // 2
```

Первый вариант ничем не отличается от варианта, использованного ранее, т. к. параметр функции, являющийся указателем, можно описать с пустыми квадратными скобками, вместо звездочки, тем самым подчеркнуть, что это будет указатель на массив. Во втором варианте параметр описан как массив из 10-ти элементов. Второй параметр в этом случае не нужен. Но сам параметр **pm** по-прежнему будет являться указателем, но передавать ему нужно будет адрес начала массива типа **double**, содержащего 10 элементов. В результате такая функция становится менее универсальной.

В языке Си передать массив в функцию, можно только указав его адрес, по значению массивы не передаются. Поэтому, имея в распоряжении адрес массива, функция может с массивом делать что угодно, в том числе и изменять содержимое массива.

При передаче строки, в функции создается параметр-указатель типа **char** или **wchat_t**. Размер массива в этом случае передавать не нужно, т. к. окончание строки в массиве определяется символом с номером 0.

```
// Перевод символов строки в верхний регистр
void strupr(char *str)
{
   while(*str)
        *str++=toupper(*str);
}
```

указатель на строку

Функция **toupper** возвращает латинскую букву в верхнем регистре, если в параметре указан номер латинской буквы в нижнем регистре, в противном случае возвращается тот же номер. Цикл **while** в функции будет выполняться до тех пор, пока не будет обнаружен символ с номером 0.

При передаче в функцию двухмерных и n-мерных массивов, можно поступать так же, как и с одномерными массивами, передавая адрес начала массива, количество строк

и столбцов. В этом случае вычисление положения переменной по номеру строки и столбца приходится брать на себя, но функция получится более универсальной.

Пример функции, которая реализует отражение массива слева на право (можно использовать в графике):

```
void ltor(short *pm, int rows, int cols)
{
   int i,j;
   short c;
   for(i = 0; i < rows; i++)
     for(j = 0; j<cols/2; j++)
     {//Перестановка элементов массива
       c = *(pm+i*cols+j);
       *(pm+i*cols+j) = *(pm+i*cols+cols-j-1);
       *(pm+i*cols+cols-j-1) = c;
     }
}
…
   short image[50][40] ;
…
   ltor((short*)image,50,40); // Вызов функции и различные
   ltor(image[0], 50, 40);    // варианты указания адреса
   ltor(&image[0][0],50,40);  // начала массива
```

Благодаря возможности в языке Си создавать указатели на двухмерные и n-мерные массивы, параметры такой функции можно описать по-другому и работать придется проще, но функция будет привязана к массивам, имеющим одно и то же количество столбцов:

```
void ltor2(short (*pm)[40], int rows)
{
   int i,j;
   short c;
   for(i = 0; i < rows; i++)
     for(j = 0; j< 40/2; j++)
     {//Перестановка элементов массива
       c = pm[i][j];
       pm[i][j] = pm[i][cols-j-1];
       pm[i][cols-j-1] = c;
     }
}
. . .
   short image[50][40] ;
   ltor2(image, 50); // Вызов функции
```

В функции **ltor2** параметр **pm** описан как указатель на двухмерный массив, содержащий в каждой строке 40 переменных (имя указателя и звездочка должны быть заключены в круглые скобки). При добавлении к такому указателю 1 его содержимое увеличится на **40*sizeof(short) == 80** для перехода на следующую строку двухмерного массива. Через такие указатели удобно работать с массивами используя операцию **[]**, как показано в функции. Таким образом, можно создать указатель на любой n-мерный массив:

```
int (*pm3)[5][10];
```

Указатель на трехмерный массив, содержащий таблицы размером 5 строк и 10 столбцов. При добавлении к такому указателю 1 его содержимое увеличится на **5*10*sizeof(int) == 200** для перехода на следующую таблицу трехмерного массива.

Для получения из функции нескольких результатов в функции предусматривают нужное количество параметров – указателей, которым при вызове функции указывают адреса переменных, куда функция должна заложить результаты.

```
// Вычисление корней квадратного уравнения
int sqr(double a,double b,double c,double *px1,double *px2)
{
   double d = b*b-4*a*c;
   if(d < 0) return 0 ;  // Корней нет
   if(a == 0) return 0; // Не квадратное уравнение
   *px1 = (-b + sqrt(d))/2/a; // <- 1-й корень по адресу в px1
   *px2 = (-b - sqrt(d))/2/a; // <- 2-й корень по адресу в px2
return 1; // Корни получены
}
…
   double x1, x2;
…
   if(sqr(1, 2, -3, &x1, &x2)) // Вызов функции
        printf("x1 = %lg, x2 = %lg\n", x1, x2) ;
  else
        printf("Корней нет!\n") ;
```

параметры-указатели

адреса переменных

Функция **sqr** вычисляет корни квадратного уравнения, получая коэффициенты **a**, **b** и **c**, при этом в качестве результатов возвращает три числа. Первое число типа **int** возвращается оператором **return**, которым функция сообщает о полученных результатах, а по адресам переменных, которые должны содержаться в указателях **px1** и **px2** закладываются значения двух корней квадратного уравнения.

Для создания в памяти различных динамических структур хранения информации нужно использовать стандартные функции, с помощью которых можно выделить память под эти структуры и их элементы. К таким функциям относятся:

```
malloc(кол-во_байтов)
calloc(размер_одной_переменной, кол-во_переменных)
realloc(адрес_выделенного_участка, кол-во_байтов)
```

Функции **malloc** необходимо указать размер требуемого участка памяти в байтах. Функции **calloc** указывается размер в байтах одной переменной и количество требуемых переменных. Функция **realloc** является более универсальной, т. к. с ее помощью можно выделить память, указав для первого параметра значение **NULL**, а впоследствии изменять размеры имеющихся участков (независимо от того, какой функцией было сделано выделение), указывая для первого параметра адрес начала выделенного участка, а во втором параметре новый требуемый размер в байтах.

Все три функции, в случае успешного выделения памяти или изменения размера ранее выделенного участка (**realloc**), возвращают адрес начала, полученного или измененного участка. При этом гарантируется, что будет получен непрерывный участок памяти размером не менее запрошенного объема, кроме того, пользоваться этим участком будет только та программа, для которой участок был выделен.

Если не удастся подобрать для программы свободный участок памяти указанного размера, все три функции возвращают нулевой адрес **NULL**, как признак ошибки. Кроме того, функция **calloc** после выделения памяти обнуляет все байты на этом участке, поэтому работает медленнее. При выделении памяти функциями **malloc** и **realloc** участки памяти будут с мусором.

Если изменять размер участка функцией **realloc**, вся информация, находящаяся на участке, будет скопирована в новый участок при изменеии его положение, правда, если при этом функции не удастся изменить размер участка, она возвратит **NULL**, но участок с прежним адресом останется в распоряжении программы.

В примере показано, как выделить память под динамический массив, размером 1000 переменных типа **double** разными функциями.

```
double *pm;
…
pm  =  malloc(1000*sizeof(double));  // или
pm = calloc(sizeof(double), 1000); // или
pm = realloc(NULL, 1000*sizeof(double));
```

Поскольку при выделении памяти нет гарантии, что будет получен участок памяти запрошенного размера, даже если будет выделяться 1 байт, поскольку вся память уже может быть распределена между программами, после работы каждой функции необходимо делать проверку содержимого указателя на значение **NULL** и только после этого работать с полученным участком.

```
if(pm == NULL)
{
  printf("Для работы нет свободной памяти!!!") ;
  exit(1) ; // Если программа дальше продолжаться не может
}
```

Функция **exit** используется обычно для аварийного завершения программы. Единственному параметру в этой функции указывают код завершения программы: 0 – нормальное завершение, другое значение – ненормальное завершение.

Имея в указателе **pm** адрес начала непрерывного участка, программа может организовывать на этом участке хранение и обработку информации, работая с участком через этот указатель и операции адресной арифметики. Выходить за рамки участка не разрешается.

```
for(i = 0; i < 1000; i++)
   *(pm+i) = rand() ; //Заполнение массива случайными числами
```

Если при работе окажется, что размер участка необходимо изменить, это можно сделать функцией **realloc**, указав в первом параметре адрес начала выделенного участка, во втором параметре новый размер участка в байтах.

```
pm = realloc(pm, 2000*sizeof(double)) ;
```

В примере показано, как увеличить динамический массив, адрес которого содержится в **pm**, в два раза, и довести его до 2000 переменных. После попытки изменить размер участка, необходимо сделать проверку содержимого указателя **pm**, как было показано выше. При таком варианте работы с функцией **realloc** будет потерян адрес начала имеющегося участка, если в процессе изменения размера функция вернет **NULL**.

Поэтому, если необходимо продолжить работу программы без потери выделенной памяти, перед изменением размера, необходимо предварительно сохранить адрес начала выделенного участка в другом указателе, или присваивать значение другому указателю.

```
pm2=realloc(pm,2000*sizeof(double));
if(pm2 == NULL)
  printf("Для работы нет свободной памяти!");
else pm = pm2 ; // Запоминаем новый адрес
```

Когда все задачи будут решены, для которых понадобилось выделять участок памяти, он должен быть освобожден функцией **free**, которой необходимо указать адрес начала ранее выделенного участка памяти.

```
free(pm);
```

После освобождения участка памяти, несмотря на то, что в указателе останется его адрес, обращаться к нему нельзя.

Массивы указателей

В массивы можно объединять переменные любых типов, в том числе и указатели.

```
int *pm[10] ; // Массив из 10 указателей
```

Используются массивы указателей в задачах, в которых приходится оперировать набором адресов.

Ниже показан пример организации хранения массива константных строк, при этом в массиве хранятся не сами строки, а их адреса.

```
char *ps[5] = {"continue","break","while","switch","goto"} ;
```

Для организации хранения массива константных строк можно воспользоваться двухмерным массивом, но в этом случае будет много неиспользуемых байт, т. к. в двухмерном массиве придется указать количество столбцов, равное размеру максимальной строки с учетом конечного нуля.

В задачах, связанных с перестановкой информации, если информация занимает много байт, лучше воспользоваться массивом указателей на эту информацию и переставлять указатели вместо информации.

В примере показано, как через массив указателей **ps** отсортировать массив строк, не переставляя сами строки в массиве.

```
char ms[5][100], *ps[5];
…
//Заполнение массива указателей
 for(i = 0;i < 5; i++) ps[i] = ms[i];
…
// Сортировка строк
    for(i = 0; i < 4; i++)
      for(j = i+1; j < 5; j++)
         if(strcmp(ps[i],ps[j])> 0)
         { //Перестановка адресов строк
             char *pc = ps[i];
             ps[i]=ps[j];
             ps[j] = pc;
         }
…
// Вывод массива строк
for(i = 0; i < 5; i++) puts(ps[i]);
```

Указатели на функции

Кроме указателей на информацию, язык Си позволяет создавать указатели на функции, которые используются для вызова функций, зная их адреса. Декларация указателя на функцию зависит от прототипа функции:

```
double имя_функции(double); // Прототип многих
                                математических функций
```

Декларация указателя на такие функции выглядит следующим образом:

```
double (*pf)(double);
```

Имя указателя и звездочка должны быть взяты в круглые скобки. После имени указателя в круглых скобках должны быть перечислены типы параметров. Такому указателю можно присвоить адрес любой математической функции, который задается именем функции:

```
pf = sqrt; // Присвоение указателю адреса функции
y = pf(x); // Вызов функции sqrt через указатель pf
```

Используются указатели на функции в следующих случаях:

- Для вызова функций по адресу, который будет получен в процессе работы программы. По этому принципу в программах можно использовать динамически подключаемые библиотеки функций.
- Для создания универсальных функций, в которых решение некоторой задачи может зависеть от используемых типов данных или от программы, в которой будут использоваться эти функции.

Примером таких функций в языке Си могут служить функции быстрой сортировки и бинарного поиска:

```
qsort(адрес_массива, размер_массива ,
          размер_одной_переменной, адрес_функции_сравнения);
```

В четвертом параметре указывается адрес собственной функции сравнения, которая получит адреса двух элементов и вернет результат сравнения по принципу функции **strcmp** для сортировки по возрастанию.

Функция **bsearch** осуществляет быстрый поиск в отсортированных массивах и возвращает адрес найденного значения.

```
bsearch(адрес_искомого_значения, адрес_массива,
            размер_массива, размер_одной_переменной,
            адрес_функции_сравнения);
```

Реализация функций сравнения для массивов типа **double** может быть сделана следующим образом:

```
int comp(const void *el1, const void *el2)
{
  return *((double*)el1) - *((double*)el2);
}
```

Задачи

1. Написать функцию, определяющую максимальное число в массиве типа **int**.
2. Написать функцию для сортировки массивов типа **int**.
3. Написать функцию, определяющую максимальное и минимальное число в массиве типа **int**.
4. Написать функцию, возвращающую номер строки в двухмерном массиве, в которой содержится максимальное число.
5. Написать функцию, подсчитывающую сумму по каждому столбцу в двухмерном массиве.
6. Написать функцию, удаляющую из строки фрагмент текста, задаваемый индексом начала и количеством символов.
7. Написать функцию для преобразования латинских букв, содержащихся в строке, в верхний регистр.
8. Написать функцию, которая бы определяла, содержит ли строка представление целого числа в десятичной системе счисления.
9. Написать программу, которая получает неограниченное количество чисел от пользователя и определяет сумму всех чисел, среднее арифметическое, максимальное, минимальное число, 5 наибольших и 5 наименьших чисел. Хранение чисел организовать в динамическом массиве.

10. Собственные типы данных

Кроме встроенных типов данных и указателей язык Си предоставляет возможность программистам создавать различные собственные типы данных. К ним относятся структуры, объединения и перечисления, а также возможность создавать новые имена типов данных с помощью ключевого слова **typedef**.

На собственные типы данных, как и на любые переменные, распространяется понятие области видимости, а именно, в какой области видимости создается собственный тип данных, в такой области им и можно пользоваться.

Структуры

Структура – это набор переменных возможно разного типа. Используются структуры для объединения в одной переменной разнообразной информации, являющейся характеристиками некоторых объектов в программе. Переменные, входящие в состав структур, называют элементами.

Описание структурного типа данных делается по следующей форме:

```
struct [ИМЯ]
{
   описание элементов}
[список_декларируемых_переменных];
```

Описание элементов делается так же, как и декларация переменных, но без указания инициализации.

После закрывающей фигурной скобки можно указать список декларируемых переменных созданного типа данных.

Названием структурного типа данных служат два слова, слово **struct** и имя, указанное после слова **struct**. Поэтому, если имя не указать, то воспользоваться в последствии этим типом данных не будет возможности, за исключением декларируемых переменных, которые могут быть указаны в конце формы.

Создавать структурные типы данных можно в местах декларации переменных или внутри других структур, причем до первого использования создаваемого типа. При этом необходимо помнить, что на собственные типы данных распространяется область видимости. Поэтому, создав локально тип данных, локально им можно будет пользоваться.

Часто можно встретить достаточно простые структуры, в состав которых входят переменные одного типа:

```
struct _DATE  // Тип данных для хранения даты
{
   int day, mon, year; //Номер дня, месяца и года
};
```

В состав структурного типа данных могут входить переменные любых типов и любые наборы данных. Есть только одно ограничение - в состав структуры не может входить переменная той же структуры, но не указатель.

```
struct _COMPUTER // Тип для хранения данных о компьютере
{
   char processor[30];      // Марка процессора
   unsigned freq;           // Частота процессора
   unsigned mem  ;          // Объем оперативной памяти
   struct _DATE date_prod;// Дата производства
   float cost;              // Стоимость
   struct _COMPUTER *next;// Адрес следующей переменной
};
```

Декларация переменных структурного типа ничем не отличается от переменных других типов:

```
struct _DATE d1, d2, md[50], *pd;
struct _COMPUTER c1,c2,mc[20],*pc;
```

Каждая структурная переменная получает тот набор элементов, которые были указаны при описании ее типа данных и объем структурной переменной будет не меньше суммы объемов ее элементов. Элементы, входящие в состав структурных переменных, сохраняют за собой имена, данные им при описании.

Ниже показано расположение элементов структурных переменных в памяти, которые будут располагаться друг за другом в порядке их описания в типе данных.

struct _DATE

day				mon				year			
200	201	202	203	204	205	206	207	208	209	210	211

struct _COMPUTER

processor						freq		mem		date_prod						cost		next	
0	1	2	…	28	29					day		mon		year					
400	401	402	…	428	429	430	…	434	…	438	…	442	…	446	…	450	…	454	…

В нижней строке условно показаны адреса элементов.

Со структурным типом данных можно выполнять только несколько операций и самой ходовой операцией является операция доступа к элементу структуры, обозначаемая точкой:

```
d1.day = 25;      // Номер дня
d1.mon = 6;       // Номер месяца
d1.year = 2004;   // Номер года
```

В примере показано, как в структурную переменную **d1** положить на хранение дату 25.06.2004 г.

С элементами структурных переменных можно выполнять такие же действия, как и с переменными таких же типов данных:

```
c1.processor="Pentium";//Массиву нельзя присвоить значение
strcpy(c1.processor, "Pentium"); //но можно скопировать
```

Если элементами структурного типа данных являются наборы данных (массивы, структуры), то к ним можно применять такие же операции, что и к обычным наборам:

```
c2.processor[0] = 0;        // Занесение пустой строки
c2.date_prod.year =  2006; // Задание года производства
```

Вторая операция, полезная для структур – это операция присвоения значения. Структурной переменной можно присвоить значение другой переменной того же типа данных.

```
d2 = d1;
c2 = d1; //Ошибка-разные типы переменных
c2.date_prod = d1;
```

Третья операция, которую можно применять как с переменными структурного типа, так и с самим структурным типом данных – это операция **sizeof**. При работе со структурным типом данных лучше положиться на результат этой операции, нежели вычислять его объем, суммируя объемы элементов, т.к. между элементами структур могут быть пропуски в виде неиспользуемых байтов, которые применяются для выравнивания положения элементов структур.

И, наконец, операция взятия адреса, с помощью которой можно узнать положение структурной переменной в памяти.

```
pc = &c1;
```

При работе через указатель на переменную структурного типа данных, язык Си предоставляет специальную операцию доступа к элементам структурной переменной, обозначаемую стрелкой ->.

```
(*pc).mem = 512; // Можно и так, но лучше
pc->mem = 512;   // через специальную операцию ->
```

При работе с массивами структур необходимо помнить, что каждый элемент массива является структурной переменной, к которой применима операция точка:

```
md[0].day = 16;
md[0].mon = 5;
md[0].year=2005;
```

Для реализации задач с самим массивом применяется как обычно цикл **for**:

```
for(i = 0; i < 50; i++)
     md[i].day=1, md[i].mon = 1, md[i].year=2000;
```

операция "запятая"

В примере во все структурные переменные заносится дата 01.01.2000 г.

Массивы структурных переменных являются неотъемлемой частью при организации работы с базами данных. Так массив **mc** можно рассматривать как одну таблицу реляционной базы данных (без первичного ключа), содержащую информацию о компьютерах.

Битовые поля

В качестве элементов структурных типов данных можно использовать целые переменные нестандартных размеров, называемые битовыми полями. Размеры таких элементов могут начинаться с одного бита и указываются в виде целого числа при описании структурного типа данных.

```
struct POLE
{
   int a;
   int b : 5 ;        // Знаковое поле размером 5 бит
   unsigned c : 1 ;   // Поле без знака размером 1 бит
   char d :  2 ;      // Знаковое поле размером 2 бита
} ;
```

Элемент **a** является обычной переменной типа **int**, остальные элементы являются битовыми полями. Тип данных при описании битовых полей определяет представление целого числа в таком элементе и никакого отношения не имеет к размеру поля. Так поля **b** и **d** будут предназначены для хранения целых чисел со знаком, а поле **c** целого числа без знака.

С битовыми полями структур работают так же, как и с обычными переменными целых типов данных.

```
struct POLE p;
…
p.b = -10  ;      // 1
p.c = 1  ;        // 2
p.d = 3  ;        // 3
```

Однако имеются следующие особенности:

- нестандартный размер приводит к нестандартному диапазону, в котором может работать битовое поле. В связи с этим, оператор 3 присваивает полю **d** значение, которое выходит за рамки диапазона этого поля, т. к. его диапазон от -2 до 1;
- с битовыми полями можно выполнять все операции, кроме получения адреса и измерения размера операцией **sizeof**;
- битовые поля нельзя объединять в массивы.

Битовые поля используются для уменьшения объема, занимаемого структурными переменными, а также для того, чтобы избежать применения двоичных операций.

Инициализация структур

Инициализация структур делается так же, как и массивов, но при этом следует учитывать порядок описания элементов в структурных типах данных и их типы:

```
struct _DATE d = {25, 1, 2004}; // Дата 25.01.2004 г.
```

При инициализации структур можно давать неполные списки инициализации, при этом работает такое же правило, как и при инициализации массивов. В примере элементы **cost** и **next** структурной переменной **c** будут проинициализированы 0.

```
struct _COMPUTER c = {"Pentium",2600,512,25,1,2004,452.5};
```

Для наборов (массивы, структуры), входящих в состав структурного типа данных значения в списке инициализации можно заключать в дополнительные фигурные скобки и указывать для них неполные списки.

```
struct _COMPUTER c= {"Pentium",2600,512, {25,1,2004},
                                                  452.5};
```

Для инициализации всех элементов структурной переменной 0 достаточно указать в фигурных скобках один нуль.

```
struct _COMPUTER c = {0};
```

Выравнивание полей структуры

При размещении в памяти полей структуры компилятор выполняет выравнивание этих полей для оптимизации доступа к ним. Как результат поле структурной переменной может занимать больше байт в памяти, чем это требуется исходя из её типа. Для того чтобы программист мог сам управлять выравнивание полей в стандарте C11 появился модификатор **_Alignas**, который явно задаёт выравнивание для поля. Модификатор **_Alignas** неприменим к битовым полям и переменным с модификатором **register**. Есть две формы: **_Alignas(**выражение**),** где выражение – это целочисленное константное выражение, являющееся степенью двойки и вторая форма **_Alignas(**тип**),** где тип – это любой существующей тип. Окончательная величина выравнивания будет наибольшее из указанного в модификаторе **_Alignas** и минимального значения требуемого для размещения поля структуры в памяти, т.е. если указано **_Alignas(**2**)double data;** то выравнивание будет равно восьми. Для получения требуемого значения выравнивания можно воспользоваться оператором **_Alignof(**тип**).** Результатом этой операции будет целочисленная константа типа **size_t**. Пример:

```
struct Test {  // Все поля по 8 байтов
    short a;
   _Alignas(8) long  b;
   _Alignas(8) float f;  };
```

Объединения

Тип данных объединение по использованию очень похож на структурный тип. Отличается от структур только тем, что вместо ключевого слова **struct** применяется другое ключевое слово **union.** Так выглядит описание объединения.

```
union MY
{
char c;
int i;
double d;
 char str[20];
};
```

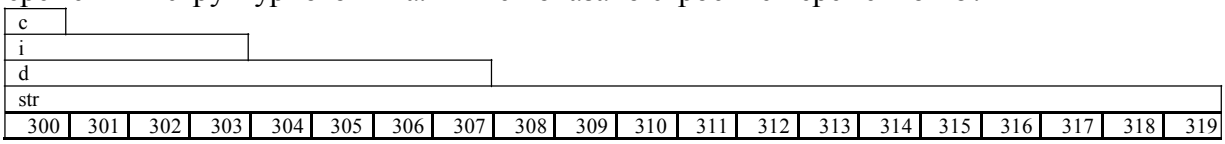
Название этого типа данных, как и при работе со структурами, будут обозначать два слова, слово **union** и имя, указанное в описании типа.

```
union MY a, b;
```

С элементами объединения и переменными типа объединение можно выполнять такие же операции, как и со структурами.

```
a.c = 'Z';
a.i = 123;
a.d = 3.1415;
strcpy(a.str,"Hello");
b = a;
```

Однако строение переменных этого типа данных в корне отличается от строения переменных структурного типа. Ниже показано строение переменной **a**.



В отличие от структур, у которых элементы в памяти расположены друг за другом, в объединении все элементы лежат в памяти друг на друге и начинаются с одного и того же адреса. В связи с этим, первые байты у элементов объединения всегда являются общими, что приводит к изменению содержимого всех элементов при задании значения какому-то одному элементу и объем памяти, занимаемой объединением, равен объему самого большого элемента.

Поскольку у элементов объединения имеются общие байты, то элементы объединений можно использовать для хранения информации по очереди, а саму переменную типа объединение можно применять как универсальную переменную, предназначенную для хранения в разные моменты времени различной по типу информации (см. предыдущий пример). При использовании объединений как универсальных переменных, по сравнению со структурами, удается сэкономить память.

Второе применение объединений связано с эффектом наложения всех элементов друг на друга. Так, например, через элемент **c** можно получить доступ к первому байту элемента **i**, а элемент **i** дает возможность работать с первыми четырьмя байтами элемента **d**. А если в объединении создать переменную структурного типа с битовыми полями, то можно получить доступ к отдельным битам или группам битов других элементов.

```
union CHAR    // Объединение структурной переменной
{
   struct     // Описание структурного типа данных
   {
      unsigned b0 : 1; //Поля для работы с битами
      unsigned b1 : 1; // одного байта
      unsigned b2 : 1;
      unsigned b3 : 1;
      unsigned b4 : 1;
      unsigned b5 : 1;
      unsigned b6 : 1;
      unsigned b7 : 1;
   } bits;          // Переменная структурного типа с 8 полями

     char c;        // Наложение переменной c на битовые поля
};                  // переменной bits

   union CHAR a;
   a.c = 123;                    // Заносим целое число

   printf("%d", a.bits.b3) ; // Выводим значение 3-го бита
```

Перечисления

Перечисление – это набор именованных констант целого типа. Форма описания перечисления похожа на форму структур, но в начале ставят ключевое слово **enum**:

```
enum ИМЯ
{
   список_имен_констант}
[список_декларируемых _переменных];
```

Используются перечисления для организации хранения и обработки информации целого типа, имеющей небольшое количество значений. Это дает возможность повысить наглядность текста программы и использовать проверку типов данных со стороны компилятора. Например, для хранения и обработки дней недели можно использовать целые числа в диапазоне от 0 до 6.

```
int day;
day = 0;        // Присваивается значение 0
if(day == 5)…   // Проверка на равенство 5
```

А можно создать перечисление с названиями дней недели и оперировать именами, а не числами.

```
enum WDAYS {
Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday
// 0        1         2       3       4         5    6
};
…
enum WDAYS day;         // Переменная для хранения дня недели
…
day = Monday;           // В переменную заносится Понедельник
…
if(day == Saturday)…    // Проверка на равенство Субботе
```

Первый элемент перечисления всегда получает значение, равное 0. Каждому последующему элементу перечисления задается значение, на единицу больше, чем у предшественника. Однако, любому элементу перечисления можно явно указать значение, в этом случае последующему элементу будет по-прежнему присвоено значение на 1 больше.

```
     enum COLOR   {Red, Green=2, Blue, White=5, Black};
 // Значения ->    0       2      3       5         6
```

Оператор typedef

С помощью оператора **typedef** в языке Си можно создавать новые имена имеющихся типов данных. Форма использования оператора следующая:

```
typedef имя существующего типа новое_имя;
```

После ключевого слова **typedef** указывается название имеющегося типа данных, после него новое имя в виде одного идентификатора. Используется этот оператор для получения более понятных для программиста имен типов и/или более коротких названий. Например,

```
// Получение понятного названия типа для работы с
логикой typedef int BOOL;
// Получение короткого названия без слова struct
typedef struct _COMPUTER COMPUTER;
…
// Функция, возвращающая логический результат
BOOL sqr(double a, double b, double c,
   double *px1, double *px2);
// Декларация переменной структурного типа данных
COMPUTER c;
```

Оператор может использоваться для создания имени типа данных, являющегося массивом, в этом случае после названия типа указывают размер массива:

```
typedef double MATRIX[3][3] ;
…
   MATRIX a, b, c; // Декларация трех двухмерных массивов
```

При использовании указателей на функции оператором **typedef** можно создать простое название типа данных для таких указателей. Созданный тип значительно упростит декларацию переменных и описание параметров функций:

```
typedef int (*COMPFUNC)(const void *el1, const void *el2) ;
// Описание параметра-указателя на функцию сравнения
// элементов в массиве

// Функция поиска максимального элемента массива с новым типом
void fmax(void *max, const void *mas, int size, int width,
                                              COMPFUNC compare);
// и без
void fmax(void *max, const void *mas, int size, int width, int
          (*compare)(const void *el1, const void *el2)) ;
```

Задачи

1. Создать структурный тип данных для хранения информации о книге.
2. Создать структурный тип данных для хранения информации о фильме.
3. Создать объединение для работы с отдельными байтами целого числа типа **int**.
4. Создать перечисление для работы с месяцами года. Нумерацию констант начать с единицы.
5. Для типов данных, созданных в п. 1 – 4 с помощью оператора **typedef** создать короткие имена.
6. Создать функцию, вычисляющую промежуток времени между двумя отметками времени в виде количества часов, минут и секунд. Отметки времени функция должна получать через параметры структурного типа, а результат возвращать в виде значения структурной переменной.
7. Организовать ввод и хранение информации о книгах. Для проверки правильности хранения в конце ввода вывести всю информацию на экран. Задачи получения информации и вывода на экран реализовать в виде двух функций.
8. Создать структурный тип данных, который будет содержать название месяца, трехбуквенную аббревиатуру месяца, количество дней в месяце и номер месяца.
9. Определить массив, состоящий из двенадцати структур того же типа, что и в п. 8, и проинициализировать его данными месяцев для невисокосного года.
10. Создать функцию, которая получает номер месяца и массив, определенный в п. 9 и возвращает количество дней от начала года до указанного месяца.

11. Работа с файлами

Для работы с файлами язык Си предоставляет большой набор стандартных функций, названия которых начинаются с буквы **f**, описанных в файле **stdio.h**. При работе с этими функциями нужно соблюдать следующую последовательность действий для каждого файла, которая похожа на действия пользователя при работе с файлами в прикладных программах.

1. Попытаться получить доступ к содержимому файла. Называют этот этап открытием файла.
2. Проверить наличие доступа.

3. Если доступ получен, реализовать работу с файлом.
4. Когда все задачи с файлом будут решены, закрыть доступ к файлу.

Реализация этих этапов делается следующим образом.

Открытие файла.

Для открытия файла используется следующая функция:

```
fopen(имя_файла, режим_работы)
```

имя_файла – в виде строки указывается имя файла или путь к файлу, используя системные соглашения, доступ к которому необходимо получить. Для систем DOS и Win-dows эта строка может выглядеть следующим образом:

"my.txt" – указано только имя файла, включая расширение (расширений по умолчанию нет). Такой файл должен находиться в рабочем каталоге программы (рядом с программой). Обычно это каталог, с которого запускается программа.

"data\\my.txt" – указан относительный путь (относительно рабочего каталога) к файлу.

"c:\\windows\\my.txt" – указан абсолютный путь к файлу.

режим_работы – задается в виде строки, в которой может содержаться от одного до трех символов, с помощью которых указывается, что программа намерена с файлом делать. Ниже показаны различные варианты.

"r" – открыть существующий файл только для чтения.

"w" – открыть файл для перезаписи. Если файл не существует, он создается, в противном случае содержимое файла удаляется, и размер файла сразу после открытия будет равен 0.

"a" – открыть файл для добавления в конец. Если файл не существует, он создается, в противном случае содержимое файла не удаляется, а все, что программа будет записывать в файл, будет добавляться в конец файла. Управлять местом записи в этом режиме нельзя.

Каждый из этих режимов разрешает только один вариант работы: чтение информации из файла или запись информации в файл. Для получения сразу двух вариантов работы в конец строки необходимо добавить знак **+**.

"r+" – открыть существующий файл для чтения и записи. **"w+"** – открыть файл для перезаписи и чтения.

"a+" – открыть файл для добавления в конец и чтения с любого места.

Все перечисленные режимы пригодны только для текстовых файлов. Файлы считаются текстовыми, если они содержат только текст, разбитый на строки, конец строки в которых обозначается парой символов **\r\n**. В зависимости от операционных систем, конец строки может обозначаться и одним из указанных символов.

При работе с такими файлами происходит преобразование символов: при чтении пары **\r\n** будет прочитан только символ **\n**, а при записи символа **\n** к нему будет добавлен символ **\r** для получения завершающей последовательности **\r\n**. Кроме того, при чтении информации из файла, если программа встретит символ, являющийся признаком конца файла, дальнейшее чтение информации из файла будет невозможным. В системе DOS и Windows таким символом является символ EOF (end of file), у которого 26 номер в ASCII таблице и который с клавиатуры можно ввести сочетанием клавиш Ctrl+Z.

Остальные файлы называют двоичными или бинарными файлами. Работа с ними должна вестись без всяких преобразований и реакций на управляющие символы, в двоичном режиме. Для этого любой из перечисленных режимов можно дополнить символом **b** от слова binary. Ниже показано несколько вариантов таких режимов.

"rb" – открыть существующий файл только для чтения в двоичном режиме.

"r+b" – открыть существующий файл для чтения и записи в двоичном режиме.

Двоичный режим подходит и для текстовых файлов, но в этом случае программа должна сама решать, что делать с различными управляющими символами.

При получении доступа к файлу, функция **fopen** возвращает адрес структурной переменной типа **FILE**, который в дальнейшем должен использоваться всеми остальными функциями работы с файлами для идентификации открытого файла. В противном случае функция **fopen** возвращает **NULL**, который означает, что по каким-то причинам доступ к содержимому файла невозможен.

Ниже показан пример использования функции **fopen**.

```
FILE *f;
…
f = fopen("my.txt", "r");
```

Проверка наличия доступа.

Проверка реализуется сразу после вызова функции **fopen**, путем проверки содержимого указателя и в случае отсутствия доступа к содержимому файла, последующие этапы должны быть пропущены. При этом обычно всегда выдается на экран сообщение об ошибке, а дальнейшие действия зависят от того, насколько важно содержимое файла для программы.

```
if(f == NULL)
{
  printf("Ошибка при открытии файла.\n");
  exit(1) ; // Если программа не может жить без файла
}
```

Работа с файлом.

Для работы с файлами в языке Си существует много функций, реализующих операции чтения и записи информации и выполняющие другие задачи. Многие функции работы с файлами являются аналогами функций, используемых при организации ввода-вывода.

Функции работы с байтами. Используются для работы с любыми файлами.

fgetc(FILE*) – возвращает очередной байт, прочитанный из файла. В случае ошибки или если достигнут конец файла, функция возвращает EOF, значение, которое выходит за рамки однобайтных переменных целого типа. Параметру функции необходимо указать значение, полученное от функции **fopen**.

Пример чтения содержимого всего файла и выдачи его на экран:

```
int byte;
…
while((byte=fgetc(f))!= EOF)
    putchar(byte);
```

fputc(значение _байта, FILE*) – записывает указанное значение в очередной байт в файле. В случае ошибки функция возвращает EOF.

Пример использования при решении задачи копирования файлов:

```
int byte;
```

```
…
while((byte=fgetc(f1))!=EOF) // Читать байт из одного файла
    fputc(byte, f2) ; // Записать байт в другой файл
```

Первый файл (**f1**) должен быть открыт в режиме чтения, второй файл (**f2**) в режиме перезаписи.

Функции работы со строками. Используются для работы с текстовыми файлами.

fgets(адрес_участка_памяти, размер_участка, FILE*) – читает очередную строку из файла и размещает ее в участке памяти, заданном первым параметром. Для предотвращения переполнения участка памяти, во втором параметре указывается размер участка. Чтение строки заканчивается на завершающей

последовательности **\r\n**, из которой символ новой строки будет помещен в конец, а строка будет завершена 0. Если строка окажется длиннее указанного размера, будет прочитана часть строки. В случае ошибки функция возвращает **NULL**.

Пример чтения содержимого текстового файла и выдачи его на экран:

```
char str[200] ;
…
while(fgets(str,200,f)!= NULL)
    printf("%s", str) ;
```

fputs(строка, FILE*) – записывает указанную строку в файл. В случае ошибки функция возвращает **EOF**. Если в строке не будет содержаться символ новой строки, в файл завершающая строку последовательность записываться не будет. В этом случае строку в файле можно дописать.

Пример записи строк в файл:

```
char str[200] ;
…
while(gets(str)!= NULL)    // Получить строку с клавиатуры
{
   fputs(str, f);          //и записать ее в файл, закончив
   fputs("\n", f);         //завершающей последовательностью
 }
```

Для завершения работы цикла с клавиатуры необходимо ввести символ **EOF** клавишами Ctrl+Z (с новой строки).

Форматные функции работы с файлами. Используются для работы с текстовыми файлами.

fprintf(FILE*, форматная_строка[, список_информации]) – функция осуществляет вывод в файл числовых и символьных данных в разных форматах в виде текста. Параметры у функции совпадают с параметрами функций **printf** и **sprintf**, кроме первого, в котором указывают значение, полученное от функции **fopen**. В случае ошибки функция возвращает **EOF**.

Ниже показаны примеры использования функции **fprintf**.

```
//вывод строки без указания форматов
fprintf(f,"Hello Worldn");
// вывод значения одной переменной a типа float
fprintf(f,"Сторона квадрата равна %f см.\n", a);
// вывод значения двух переменных s и p типа double
fprintf(f,"Площадь=%lf ,периметр=%lf ",s,p);
```

fscanf(FILE*, форматная_строка, список_адресов_переменных) – функция осуществляет чтение числовых (представленных в виде текста) и символьных данных с файла. При успешном чтении информации из файла функция возвращает количество прочитанной информации, а в случае ошибки **EOF**.

Примеры чтения данных из файла функцией **fscanf**:

```
// Чтение целого числа и запись его в переменную m типа
int fscanf(f, "%d", &m);
// Чтение целого числа, представленного в шестнадцатеричном
// формате
fscanf(f, "%x", &h);
// Получение целого и вещественного числа
fscanf(f,"%d%f", &i, &p);
// Получение даты в общепринятом формате: ДД.ММ.ГГГГ.
// Точки между числами функция будет пропускать
scanf("%d.%d.%d", &day, &mon, &year);
```

Блочные функции работы с файлами. Используются для работы с двоичными файлами и могут работать с блоками данных любого типа.

fwrite(адрес_участка_памяти,размер_1-го_элемента, колво_элементов, FILE*) - записывает в файл содержимое указанного участка памяти. Участок памяти задается адресом начала, размером одного элемента, расположенного на участке и количеством элементов. Функция возвращает целое число, равное количеству записанных в файл элементов, которое при нормальной работе должно равняться содержимому третьего параметра. В примере показано, как с помощью функции **fwrite** сохранять содержимое переменных и массивов.

```
double a;
int m[100];
 unsigned n= 0;
 …
 // Запись в файл одной переменной
 n += fwrite(&a, sizeof(double),1,f);
 // Запись в файл массива
 n += fwrite(m, sizeof(int), 100, f);
 if(n != 101) // В файл должна быть записана 101 переменная
    …
```

fread(адрес_участка_памяти, размер_1-го_элемента, колво_элементов, FILE*) – читает из файла указанное количество элементов и заносит их в память начиная с указанного адреса. Функция возвращает целое число, равное количеству прочитанных из файла элементов.

```
…
n = 0;
// Чтение из файла одной переменной
n += fread(&a, sizeof(double),1,f);
// Чтение из файла массива
n += fread(m, sizeof(int), 100, f);
if(n!= 101) // Из файла должна быть прочитана 101 переменная
   …
```

Закрытие файла.

Для закрытия файла используется функция **fclose**, которой необходимо указать адрес структуры **FILE**, полученной от функции **fopen**.

```
fclose(f);
```

После закрытия, доступ к файлу можно получить, только заново открыв его функцией **fopen**.

Последовательный и произвольный доступ

При работе с файлами, место в файле, с которого будет осуществляться чтение или запись информации, определяется положением указателя чтения /записи. При открытии файла, указатель чтения/записи устанавливается в начало файла. Когда программа записывает или читает информацию из файла, указатель чтения/записи перемещается к концу файла на объем прочитанной/записанной информации. Когда указатель дойдет до конца файла дальнейшее чтение информации будут невозможно. Любая функция чтения информации из файла будет возвращать код ошибки, связанной с концом файла (EOF). Запись в файл не ограничивается размером файла. Когда указатель чтения/записи будет находиться в конце файла, запись информации будет перемещать его дальше, за счет этого будет увеличиваться размер файла.

Если не управлять положением указателя чтения/записи, то говорят, что в этом случае реализуется последовательный доступ к файлу, который не всегда может быть приемлемым для решения некоторых задач с файлами. Для реализации произвольного доступа нужно управлять положением указателя чтения/записи, которое реализуется функцией **fseek**, у которой следующая форма использования:

fseek(FILE*, смещение, позиция_отсчета) - устанавливает указатель чте-ния/записи в положение, находящееся на расстоянии указанного смещения, задаваемого в байтах, от указанной позиции отсчета смещения.

позиция_отсчета – задает место в файле, от которого будет отсчитываться смещение. Существует три позиции отсчета смещения, которые задаются следующими константами:

SEEK_SET – начало файла;

SEEK_CUR – текущее положение указателя чтения/записи; **SEEK_END** – конец файла.

смещение – на сколько байт переместить указатель чтения/записи. При указании положительного смещения указатель перемещается к концу файла, при этом возможен выход за рамки файла. За концом файла ничего прочитать невозможно, но возможно записать информацию, при этом размер файла будет увеличен на объем записываемой информации и размер выхода указателя за рамки файла. Какая информация будет содержаться в пропущенном участке файла, зависит от операционной системы. Если смещение отрицательное, то перемещение указателя будет осуществляться к началу файла, за рамки которого выйти нельзя.

Если указатель будет успешно установлен в заданную позицию, функция должна вернуть число 0.

```
fseek(f, 0, SEEK_SET) ; // Возврат указателя в начало файла
fseek(f, 50, SEEK_SET) ;// Установка на 50-й байт в файле
fseek(f, 0, SEEK_END) ; // Установка указателя в конец файла
```

С помощью функции ftell можно узнать, где в файле находится указатель чтения/записи. Форма использования этой функции следующая:

ftell(FILE*) - возвращает положение указателя чтения/записи в файле, отсчитываемое всегда от начала файла.

```
fseek(f, 0, SEEK_END) ; // Установка указателя в конец файла
flength = ftell(f) ; // Получение текущего размера файла
```

Другие функции работы с файлами

feof(FILE*) – проверка на конец файла. Функция возвращает ненулевое значение, если указатель чтения/записи находится в конце файла.

ferror(FILE*) – проверка на наличие ошибок при выполнении операций чтения/записи с файлом. Функция возвращает ненулевое значение, если были ошибки, в противном случае 0.

clearerr(FILE*) – очистка индикатора ошибок. Если при работе с файлом были обнаружены ошибки, устанавливается индикатор ошибок, который проверяет функция **ferror**. Автоматически индикатор ошибок не очищается. Это необходимо делать функцией **clearerr**.

fflush(FILE*) – очищает буфер, используемый файловыми функциями для накопления информации. Если программа записывала информацию в файл, накопленная в буфере информация записывается в файл и в результате буфер очищается. Если программа читала информацию из файла, действия функции не определены.

rewind(FILE*) – перемещает указатель чтения/записи в начало файла.

rename(старое_имя, новое_имя) – переименовывает файл. В зависимости от

реализации, может выполнять переименование каталога и перемещение файла в другой каталог.

remove(имя_файла) – удаляет файл.

tmpfile() – создает временный файл в режиме **w+b**. После закрытия такого файла, он автоматически будет удален. Функция возвращает адрес структуры **FILE** для созданного временного файла.

Потоки и файлы

Функции работы с файлами в языке Си являются универсальными функциями ввода/вывода и могут работать не только с файлами, но и с экраном, клавиатурой, принтером, модемом и другими устройствами, работу с которыми можно представить в виде потоков информации, читаемых или записываемых программой в устройство. Для стандартных потоков информации, которые автоматически открываются при запуске программы, в языке Си имеются следующие имена:

stdin – стандартный поток ввода информации (обычно клавиатура); **stdout** – стандартный поток вывода информации (обычно экран);

stderr - стандартный поток вывода информации об ошибках (обычно экран).

В зависимости от операционной системы, в программе могут существовать и другие стандартные потоки. Так, например, в DOS существует имя **stdprn** для обозначения потока информации, выводимой на принтер.

Файловым функциям ввода/вывода вместо адреса структуры **FILE**, возвращаемой функцией **fopen**, можно указать имя стандартного потока информации. В этом случае файловые функции будут работать с указанными устройствами.

```
fgets(str, 200, stdin);   // Получить строку с клавиатуры
fputs(str, stdout);       // Выдать строку на экран
 // Форматный вывод на экран
fprintf(stdout, "Сторона квадрата равна %f см.\n", a);
…
fscanf(stdin, "%d", &m);  // Получить с клавиатуры целое число
```

Для получения доступа к другим устройствам, функции **fopen** вместо имени файла необходимо указать системное имя нужного устройства или порта, к которому подключено устройство. Если устройство или порт может быть представлен в виде потока информации, с которым могут работать файловые функции, к нему будет предоставлен доступ. Ниже показан пример открытия доступа к устройству, подключенному к порту **COM1:**, который можно использовать в системах DOS и Windows.

```
   f = fopen("COM1:", "r+");
```

Стандартные функции ввода/вывода (**getchar**, **putchar**, **gets**, …) тоже поддерживают потоковый ввод/вывод, который можно перенаправить в файл или другое устройство, используя системные команды. Например, в системе DOS или Windows, используя следующую команду запуска программы с именем Hello.exe можно заставить программу осуществить вывод информации в файл с именем My.txt.

```
Hello.exe > My.txt
```

Перенаправление ввода/вывода поддерживают и другие системы: UNIX, LINUX, OS/2 и др.

Можно и программно перенаправить открытый поток, включая стандартные потоки (**stdin**, **stdout** и **stderr**) на другое устройство или файл, используя функцию **freopen**, форма использования которой показана ниже:

freopen(имя_файла, режим_работы, FILE*) – перенаправляет существую-щий поток в новый файл или на другое устройство.

имя_нового_файла – имя нового файла или устройства, доступ к которому будет открыт и куда будет перенаправлен поток, указываемый в последнем параметре.

Пример перенаправления вывода на экран в файл с именем My.txt:

```
freopen("my.txt", "w", stdout);
printf("Hello World!"); //Вывод в файл
```

Для отмены вывода информации в файл, можно поток **stdout** перенаправить на консоль:

```
freopen("con", "w", stdout);     //Для DOS и Windows
printf("Hello World!\n");        //Вывод на экран
```

В системах DOS и Windows имя CON обозначает стандартное устройство ввода/вывода, называемое консолью (экран и клавиатура).

Задачи

1. Напишите программу, которая подсчитывает в текстовом файле количество латинских букв.
2. Напишите программу, которая подсчитывает в текстовом файле количество строк.
3. Напишите программу, которая подсчитывает количество слов в текстовом файле. Слова в файле должны разделяться пробельными символами.
4. Напишите программу, которая создает файл numbers.txt и записывает в него 5 введенных пользователем целых чисел. Просмотрите при помощи редактора текста созданный файл. Убедитесь, что каждое число находится в отдельной строке.
5. Напишите программу, которая дописывает в файл numbers.txt пять введенных пользователем целых чисел. Убедитесь при помощи редактора текста, что количество чисел в файле увеличивается каждый раз на 5 чисел.
6. Напишите программу, которая вычисляет среднее арифметическое чисел, находящихся в файле numbers.txt. Каждое число должно находиться на отдельной строке.
7. Напишите программу, которая четные числа из файла numbers.txt копирует в файл numbers2.txt, а нечетные в файл nembers1.txt.
8. Напишите программу, которая копирует текстовые файлы, используя функции **fgets** и **fputs**.
9. Напишите программу, которая копирует файлы, используя функции **fread** и **fwrite**.
10. Напишите программу для обработки чисел. Программа должна занести вводимые пользователем числа в массив, определить сумму чисел, среднее арифметическое, максимальное и минимальное и вывести результат на экран и в текстовый файл, включая введенные числа.
11. Напишите программу для обработки чисел. Программа должна занести вводимые пользователем числа в массив, определить сумму чисел, среднее арифметическое, максимальное и минимальное и вывести результат на экран и в двоичный файл, включая введенные числа.
12. Сделать программу чтения содержимого текстового файла, создаваемого программой в задаче 10. Все прочитанные числа должны быть размещены в переменных и массиве.
13. Сделать программу чтения содержимого двоичного файла, создаваемого программой в задаче 11. Все прочитанные числа должны быть размещены в переменных и массиве.