Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

Математический факультет

Кафедра математических проблем управления

Л.И. Короткевич

ПРОГРАММИРОВАНИЕ

Тексты лекций
для студентов 1 курса (2 семестр)
специальности 1-31 03 03-01 «Прикладная математика
(научно-производственная деятельность)»

2011

СОДЕРЖАНИЕ

**«…НАУЧИТЬ НЕВОЗМОЖНО…
можно только [помочь] НАУЧИТЬСЯ!»**

**«Программистом можно назвать только того,
кто умеет устранять ошибки, ведь написать
неработающую программу может каждый!»**

1. ЧТО ТАКОЕ ПРОГРАММА НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ

Программу можно представить как набор последовательных команд (алгоритм) для определенного исполнителя, который должен их выполнить для достижения той или иной цели. Например, условно запрограммировать можно человека, если составить ему инструкцию «как сварить суп», и он примется ее исполнять. Очевидно, что инструкция будет на естественном языке (русском, английском или др.). Программисты программируют не людей, а вычислительные машины. Трудность заключается в том, что такие машины не в состоянии понять наш язык. Для «инструктирования» вычислительных машин разработаны и разрабатываются специальные языки, называемые *языками программирования*.

Любую программу выполняет *центральный процессор*. Для того, чтобы процессор мог программу выполнить, она должна быть загружена в *оперативную память*. Т.е. и код программы и ее данные при выполнении программы процессором находятся в оперативной памяти.

*Что такое памят*ь? По сути, это ряд пронумерованных ячеек. Номер ячейки является адресом этой ячейки памяти. Ячейки памяти реального компьютера – это набор из нескольких переключателей, каждый из которых находится в одном из двух состояний: включено (его обозначают 1) или выключено (его обозначают 0). В ячейке памяти таких переключателей, как правило, 8. Каждый переключатель называют битом и говорят, что в ячейке 8 бит или 1 байт. Т.е. одна ячейка памяти является байтом. Содержимое любой ячейки памяти выглядит всегда как последовательность нулей и единиц, независимо от того, что в них находится: число, символ или адрес другой ячейки памяти.

Процессор может выполнять программу, написанную только на *машинном языке*. Машинный язык – это «язык процессора». Программа на машинном языке состоит из машинных команд, записанных в двоичном коде (с помощью 0 и 1). Каждая машинная команда имеет две составляющие: код операции и адресную часть. Код операции определяет, какую команду должен исполнить процессор (элементарное действие, которое может выполнить процессор, например, «переслать байт из одного места в памяти в другое»). Адресная часть указывает, где в памяти компьютера хранятся операнды и куда поместить результат выполнения операции.

2. ОБЩЕЕ ЗНАКОМСТВО С ЯЗЫКОМ С

Язык С (Си) является языком программирования высокого уровня общего назначения, который в то же время позволяет сделать многое из того, что свойственно языкам низкого уровня (ассемблерам). Был разработан в начале 70-х годов Кеном Томпсоном и Денисом Ритчи, сотрудниками компании Bell Labs. Язык С изначально был создан для программирования под операционную систему UNIX. Задумывался как альтернатива ассемблеру для написания системных программ. Сама ОС UNIX написана на С. В последствии был перенесён на множество других операционных систем и стал одним из самых популярных языков программирования.

Язык С имеет массу достоинств. В первую очередь, С ценится за эффективность. Элементы языка С (массивы, функции, указатели) максимально приближены к архитектуре компьютеров. Язык С позволяет программисту полностью контролировать компьютер средствами самого же языка. Изначально язык С был придуман, чтобы заменить ассемблер в написании операционных систем. В настоящее время большинство ОС написано на С. Но применение языка С не ограничивается только написанием операционных систем. Язык С удобен для написания очень и очень многих программ (не web-приложений).

Также язык С широко используется для подготовки специалистов, хотя изначально разрабатывался не для новичков, как тот же Паскаль. Многие языки взяли за основу синтаксис языка С. Язык С не только важен сам по себе, но и открывает дорогу к другим современным и очень популярным языкам, таким как C++, Java, C#, Perl, JavaScript и т.п.

Среды разработки для языка С: в настоящее время используется несколько интегрированных сред разработки программ на языке С. В среде профессиональных разработок наибольшей популярностью пользуются различные версии Visual C++ фирмы Microsoft. В учебных организациях предпочитают продукцию фирмы Borland: Borland C++ и Borland C++ Builder. Эти системы более просты в освоении. Это наиболее часто используемы среды – но есть и другие платные и бесплатные среды, а также компиляторы (например, Dev-C++).

3. СТРУКТУРА ПРОСТОЙ ПРОГРАММЫ НА ЯЗЫКЕ С

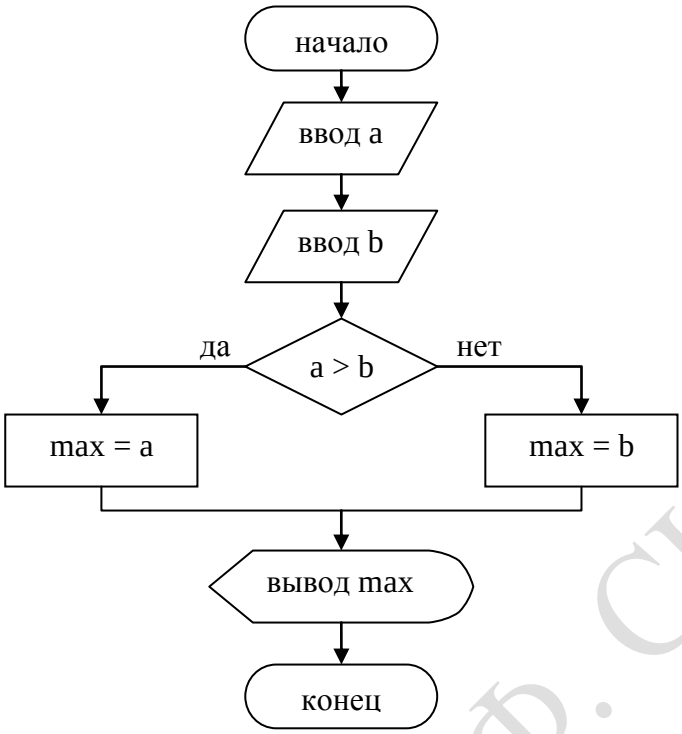
Задача. Ввести с клавиатуры два числа и вывести на экран максимальное из них.

На Паскале (для сдачи на DL):

```
{программа нахождения максимума}
program Task;
var
a, b, max : integer;
begin
  readln(a);
  readln(b);
  if a > b then max := a
```

```
  else max := b;
  writeln(max);
end.
```

Блок-схема решения задачи:



На языке С (в соответствии с правилами оформления):

```
/* Иванов И. ПМ-11 вариант 17
Работа 1.1. Найти максимальное из двух чисел */
//подключить заголовочные файлы для стандартных функций
#include <stdio.h>
#include <conio.h>  // для getch() и clrscr()
#include <bios.h>   // для bioskey(0)
void main() {
  int a, b, max;
  clrscr();       // очистить экран
  printf("a = ");
  scanf("%d", &a);
  printf("b = ");
  scanf("%d", &b);
  printf("Вы ввели a = %d и b = %d\n", a, b);
  if (a > b)
    max = a;
  else
    max = b;
  printf("max = %d\n", max);
  bioskey(0);    // ждать нажатия любой клавиши
```

```
  //getch();      // ждать нажатия любой клавиши
}
```

Про оформление задач:

1. Исходный файл должен начинаться с комментария с указанием автора программы, номера варианта и текста задачи.
2. Текст программы должен быть структурирован (желательный отступ – два символа, если пользуетесь табуляцией, то ставить Options → Environment → Editor → Tab Size = 2, строгая вложенность операторов, скобка { на новой строке или в конце строки.
3. До ввода исходных данных программа должна выводить текст, поясняющий их содержание.
4. Исходные данные после ввода, но не во время него, вывести на экран.
5. Каждый вывод сопровождать текстом, поясняющим содержание данных.
6. После вывода результатов приостанавливать программу до нажатия любой клавиши.
7. Не комментировать каждую строку программы.

```
/* Иванов И. ПМ-11 вариант 17
Работа 1.1. Найти максимальное из двух чисел */
//подключить заголовочные файлы для стандартных функций
```

/* простая программа*/ - комментарий, внутри комментария не может быть символов /* и */
/*, */ - открывающая и закрывающая скобки комментария (возможно несколько строк)
// - комментарий до конца строки

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

Подключение (include – включить) к тексту программы так называемые заголовочных (h от header – заголовок) файлов системы. В этих файлах описаны системные функции и их аргументы (прототипы функций), а также данные (например, константы, описания структур данных), которые можно использовать в программе. Используя эти описания, компилятор проверяет правильность вызова системных функций. В нашем случае программа использует следующие системные функции:

1) функции ввода scanf() и вывода printf(), описания которых находятся в заголовочном файле <stdio.h> («STanDard Input/Output Library»);

2) функцию очистки экрана clrscr() с описанием в файле <conio.h>;

3) функцию ожидания нажатия какой-либо клавиши bioskey(0), описание которой находится в заголовочном файле <bios.h> (или getch() с описанием в файле <conio.h>).

Если программа обращается к каким-либо системным функциям, то в первых ее строках обязательно должно стоять указание о подключении соответствующих заголовочных файлов. Но названия заголовочных файлов совершенно ни к чему запоминать. Чтобы узнать, какой файл надо подключить, надо стать курсором на функцию, нажать Ctrl-F1 и справа посмотреть название файла, который надо подключить.

```
void main() {
```

Программа на языке С состоит из одной или более функций, причем какая-нибудь из них (главная) обязательно должна называться main(). Описание функции состоит из заголовка и тела. Заголовок состоит из имени функции. Отличительным признаком имени функции служат круглые

скобки, а аргумент может и отсутствовать. Тело функции заключено в фигурные скобки и представляет собой набор операторов, каждый из которых оканчивается символом «точка с запятой».

Объявление функции void main() или void main(void), что значит: «функция с именем main, которая ничего не возвращает, и у которой нет аргументов (void)». Слово void можно переводить как «ничто». Далее открываются фигурные скобки, и идёт описание этой функции, в конце фигурные скобки закрываются. Между фигурных скобок находится тело функции, в котором описана последовательность действий, производимых данной функцией — логика функции.

Функция main() – эта главная функция программы, именно она начинает выполняться, когда программа запускается. Функция main() не совсем обычная и на нее накладываются определённые ограничения:

1) в каждой программе может быть только одна функция main();
2) функцию main() нельзя вызывать.

Но при этом, чтобы выполнился код любой функции, её нужно вызвать. Функция main() вызывается операционной системой.

Фигурные скобки { } отмечают начало и конец тела функции. НЕ НАДО СТАВИТЬ ТОЧКУ С ЗАПЯТОЙ ПОСЛЕ MAIN()!!!

```
int a, b, max;
```

Объявление трех переменные с именами a, b и max, которые могут принимать только целочисленные значения (тип – int). В языке С есть разница между маленькими и большими буквами: две разные переменные b и B, операторы нельзя писать большими буквами.

```
clrscr();       // очистить экран

printf("a = ");
```

Функция вывода информации на печать (экран). С ее помощью выводится фраза, заключенная в кавычки: “a = ”.

```
scanf("%d", &a);
```

Функция ввода информации с клавиатуры. Работа программы приостанавливается до тех пор, пока пользователь не наберет на клавиатуре какое-либо число и нажмет клавишу «Enter». Поступившее значение будет направлено в переменную a (не забываем про знак & перед именем переменной при вводе, который означает, что в функцию передается адрес переменной). “%d” означает, что вводится целая переменная. Точно таким же образом в следующих строках будет организован ввод значения числовой переменной b.

```
printf("b = ");
scanf("%d", &b);

printf("Вы ввели a = %d и b = %d\n", a, b);
```

Функция вывода на экран значений двух переменных a и b с поясняющим текстом. Т.к. выводим две целые переменные, то должно быть два раза “%d”. При выводе имена переменных указываются без каких-либо знаков перед ними. Обратите внимание на комбинацию ‘\n’ – она задаёт специальный символ, который является командой: «перейти на следующую строку». Таких специальных символов несколько, все они записываются с помощью символа ‘\’ (символ «backslash»).

```
if (a > b)
  max = a;      // надо ; в отличие от Паскаля
else
  max = b;
```

Сравниваются значения переменных a и b. Если проверяемое условие выполнено, т.е. значение переменной a больше, то оно присваивается переменной max – выполняется действие, записанное после проверки условия. В противном случае (else – иначе) в переменную max заносится значение b.

```
printf("max = %d\n", max);
```

Выводит на экран два сообщения: текстовое (max =) и числовое (значение переменной max).

```
bioskey(0);     // ждать нажатия любой клавиши
//getch();      // ждать нажатия любой клавиши
```

Обращение к функции приводит к задержке на экране сообщения программы до тех пор, пока пользователь не нажмет какую-либо клавишу.

```
}
```

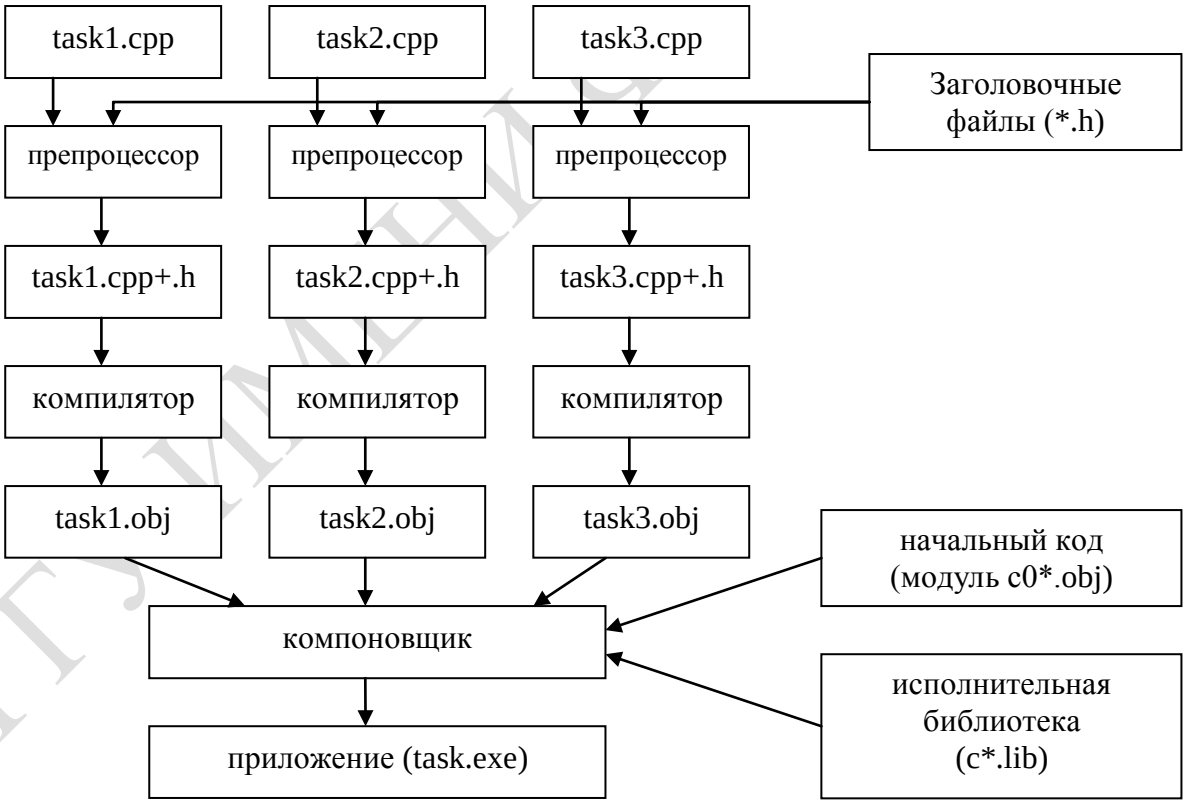
Программа завершается закрывающей фигурной скобкой.

4. ЧТО ТАКОЕ ПРОГРАММА НА ЯЗЫКЕ С

Программы на языке С составляются их исходных файлов (с расширением .c или .cpp) и заголовочных файлов (с расширением .h или .hpp).

Файл с исходным текстом программы на языке С: task.cpp (или task.c). Заголовочные файлы подключаются к файлам с программами с помощью #include.

Для того чтобы выполнить программу, необходимо ее перевести на язык, понятный процессору – в машинные коды. Этот процесс состоит из нескольких этапов.



1. Сначала программа передается *препроцессору*, который выполняет директивы препроцессора, содержащиеся в ее тексте (например, #include – включение файла в текст программы).

2. Получившийся текст передается на вход *компилятора.* Заголовочные файлы не компилируются отдельно, исходные файлы включают их в себя. В результате компиляции программы создается объектный файл: task.obj.

3. Затем выполняется редактирование связей, чтобы из объектных модулей, библиотек и исполнительной библиотеки собрать конечную программу (исполняемый файл task.exe, загрузочный файл). Эту работу выполняет компоновщик (линковщик, редактор связей). Компоновщик физически связывает файлы (например, несколько obj-файлов), в один исполняемый файл и разрешает внешние ссылки. *Внешняя ссылка* создается каждый раз, когда программа из одного файла ссылается на код из другого файла (например, это происходит при вызове системных функций).

Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. Исполняемый модуль имеет расширение **.exe** и запускается на выполнение обычным путем.

Начальный код связан с процедурами инициализации, которые исполняются перед тем, как управление будет передано функции main(), и запуском функции main() на выполнение.

Исполнительная библиотека содержит ряд функций, которые вы можете вызывать из своей программы для выполнения стандартных действий (стандартные функции преобразования данных из одного формата в другой, обработки строк, математические, управления файлами и каталогами и другие).

Про тесты и bat-файлы:

Для каждой программы необходимо подготовить:

- полное множество тестов, размещая каждый тест в отдельном файле в том же каталоге, где находится исходный файл (создание файлов Shift+F4, F4 – редактор, файлы должны быть в DOS-кодировке, что важно для русских букв);
- один командный файл (.bat) вида:

```
task <test1.in
task <test2.in
...
task <testN.in
```

где task – имя загрузочного модуля задачи, `testI.in` (I=1..N) – файлы с тестами в DOS-кодировке.

Работа в bc аналогична работе в bp:

1. Ctrl-F9 – запустить программу на выполнение
2. Alt-F9 – выполнить компиляцию модуля (.cpp → .obj)
3. F9 – создать для программы exe-файл
4. F8 – выполнить строку программы (если есть вызов функции, то вход в нее не выполняется)

5. F7 – выполнить строку программы (если есть вызов функции, то выполняется вход в нее)
6. F4 – выполнить программу да заданной строки и остановиться
7. Ctrl-F8 – установить/снять контрольную точка (после запуска программа останавливается в этой точке)
8. Ctrl-F2 – завершить выполнение программы
9. Alt-F5 – посмотреть результат работы программы (не надо для этого запускать программу заново!!!)
10. Окно watch (Ctrl-F7) – просмотр значений переменных
11. Alt-0 – список окон (удобно переходить между окнами + показывает последние закрытые окна)
12. Alt-N – переход к окну с номером N

Если при компиляции программы будет обнаружена ошибка, она появится в окне сообщений (Message window). Наличие ошибок (errors) не позволяет выполнить программу. Необходимо исправить найденные ошибки и снова скомпилировать программу. Однако даже если в программе нет синтаксических ошибок, некоторые ситуации могут вызвать подозрение у компилятора. Когда компилятор встречается с одной из таких ситуаций, то печатается предупреждение (warning). Чаще всего предупреждение указывает на реальную ошибку в программе. Программист должен проанализировать указанную ситуацию и принять соответствующее решение. В ваших программах предупреждений быть не должно.

Какие могут быть стандартные проблемы:

1) bc не запускается – скорее всего запустили bc из каталога, к которому нет доступа по записи (не из своего каталога) или нет свободного места в вашем каталоге;
2) выдало сообщение *«Linker Error: Unable to open file ‘C0x(L).obj’»* – надо подключить библиотеки (Options → Directories → везде должет быть диск C:);
3) выдало сообщение *«Error: Unable to open include file <stdio.h>»* – надо подключить библиотеки (Options → Directories → везде должен быть диск C:) или неверно написано название h-файла;
4) выдало сообщение *«Error: Function ‘clrscr’ should have a prototype»* – надо написать #include для заголовочного файла, в котором определен прототип этой функции (стать курсором на функцию, нажать Ctrl-F1 и справа посмотреть название файла, который надо подключить).

5. ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ И ТИПЫ ДАННЫХ В ЯЗЫКЕ С

Все программы работают с данными. Данные могут быть переменными и константами. Различие между переменной и константой очевидно: во время выполнения программы значение переменной может быть изменено (например, с помощью присваивания), а значение константы изменить нельзя.

В С можно использовать различные **типы данных.** Данные каждого типа занимают определенное количество байт памяти и могут принимать значения в из-

вестном диапазоне. Размер и допустимый диапазон для них в различных реализациях языка могут отличаться. Вы будете работать в среде DOS, поэтому будем ориентироваться на среду DOS или 16-бит-Windows.

Термины «бит» и «байт» используются для описания как элементов данных, которые обрабатывает компьютер, так и элементов памяти. Наименьшая единица памяти называется бит. Она может принимать одно из двух значений: 0 или 1 (т.е. информация в компьютере кодируется с помощью 0 и 1, а значит, в компьютере используется двоичная система счисления). При хранении информации в памяти компьютера каждый бит информации хранится в одном разряде памяти. 8 бит составляют 1 байт.

Записывать в двоичной системе счисления большие числа неудобно, поэтому кроме двоичной системы счисления используется также 16-ричная система счисления. Дело в том, что 1 байт кодируется в точности двузначным 16-ричным числом, что гораздо более просто и читабельно, чем в двоичной системе.

Цифры 16-ричной системы счисления: 0 1 2 3 4 5 6 7 8 9 A B C D E F.

Десятичное число	**Двоичное число**	**16-ричное число**
0_{10}	0_2	0_{16}
1_{10}	1_2	1_{16}
2_{10}	10_2	2_{16}
3_{10}	11_2	3_{16}
4_{10}	100_2	4_{16}
5_{10}	101_2	5_{16}
6_{10}	110_2	6_{16}
7_{10}	111_2	7_{16}
8_{10}	1000_2	8_{16}
9_{10}	1001_2	9_{16}
10_{10}	1010_2	A_{16}
11_{10}	1011_2	B_{16}
12_{10}	1100_2	C_{16}
13_{10}	1101_2	D_{16}
14_{10}	1110_2	E_{16}
15_{10}	1111_2	F_{16}
16_{10}	10000_2	10_{16}

Алгоритм перевода из 16-ричной системы в двоичную:

1) Каждая цифра 16-ричной системы записывается четырехзначным двоичным числом;
2) Нули, стоящие слева можно отбросить.

Алгоритм перевода из двоичной системы в 16-ричную:

1) Каждые четыре двоичные цифры, считая справа налево, записываются одной 16-ричной цифрой, которые выписываются также справа налево;
2) Если для последней четверки не хватает цифр, слева от двоичного числа дописываются нули.

$C_{16} = 1100_2$

58_{16} = 0101 1000_2 (1 байт)
$B2_{16}$ = 1001 0010_2
1101_2 = D_{16}
101010_2 = $2A_{16}$
1111 1001_2 = $F9_{16}$ (1 байт)

Надо быстро уметь переводить из двоичной системы в десятичную:

$1010_2 = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 8 + 2 = 10_{10}$

И наоборот (можно еще выписывать справа налево остатки от деления числа на 2):

$25_{10} = 16 + 8 + 1 = 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 11001_2$

Ключевые слова для определения основных (фундаментальных) типов данных в языке С:

1) Целые типы: char, int, short (короткое целое не большее, чем int), long (длинное целое не меньшее, чем int), signed, unsigned.
2) Вещественные типы: float, double, long double.

Есть еще **производные типы данных**: указатели, массивы, структуры, объединения и перечисления. Эти типы будут рассмотрены позднее.

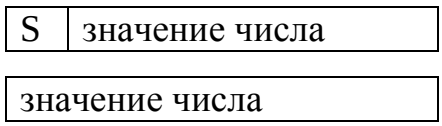
Начнем рассмотрение с **целых типов данных**.

Тип	**Длина**	**Минимум**			**Максимум**		
Целые типы данных							
char, signed char	1 байт (8 бит)	-128	-2^7	0x80	127	2^7-1	0x7F
unsigned char	1 байт (8 бит)	0	0	0x00	255	2^8-1	0xFF
short int, short	2 байта (16 бит)	-32768	-2^{15}	0x8000	32767	2^{15}-1	0x7FFF
unsigned short	2 байта (16 бит)	0	0	0x0000	65535	2^{16}-1	0xFFFF
int, signed int, signed	2 байта (16 бит)	-32768	-2^{15}	0x8000	32767	2^{15}-1	0x7FFF
unsigned int, unsigned	2 байта (16 бит)	0	0	0x0000	65535	2^{16}-1	0xFFFF
long, long int	4 байта (32 бита)	-2147483648	-2^{31}	0x80000000	2147483647	2^{31}-1	0x7FFFFFFF
unsigned long	4 байта (32 бита)	0	0	0x00000000	4294967265	2^{32}-1	0xFFFFFFFF

Ключевые слова signed и unsigned необязательны. Они указывают, как интерпретируется первый бит переменной (если указано ключевое слово unsigned, то первый бит интерпретируется как часть числа, в противном случае первый бит интерпретируется как знаковый). В случае отсутствия ключевого слова unsigned целая переменная считается знаковой.

Если описание типа состоит только из ключевого типа signed или unsigned, то подразумевается тип int.

Представление целых чисел в памяти компьютера (на примере int и unsigned int):



Почему получаются такие максимальные и минимальные значения?

Пусть у нас 4 бита и будем в них хранить число без знака. В 4 битах можно закодировать числа от 0000 до 1111 (от 0 до 15). Т.е. всего чисел, которые можно хранить в 4 битах, $16 = 2^4$. Тогда минимум: 0, а максимум: $2^4-1 = 16-1 = 15$.

Если надо хранить число со знаком, то первый бит будет знаковым (1 – отрицательное число, 0 – не отрицательное число). Тогда для хранения самого числа будут использоваться только 3 бита. Неотрицательных чисел будет $8 = 2^3$ (от 0000 до 0111), оставшиеся 8 чисел из 16 будут начинаться с 1 и будут отрицательными. Тогда минимум: $-8 = -2^3$, а максимум: $2^3-1 = 8-1 = 7$.

Тип char: длина 1 байт = 8 бит.

Всего в байтовом формате можно представить $256 = 2^8$ различных комбинаций из нулей и единиц. Эти комбинации можно использовать для представления целых чисел в диапазоне от 0 до 255 (0000 0000 … 1111 1111).

И если unsigned char (без знака), то максимум: $2^8-1 = 256-1 = 255$.

Если есть знак, то максимальное положительное число: 0111 1111 = 127 (2^7-1). Минимальное отрицательное число: 1000 0000 = -128 (-2^7- всего 128 чисел).

Отрицательные: от $-128_{10} = 1000\ 0000_2 = 80_{16}$ до $-1_{10} = 1111\ 1111_2 = FF_{16}$.

Заметим также, что у четных чисел последний бит всегда равен 0, а у нечетных – всегда 1. Это можно использовать при проверке чисел на четность/нечетность.

Положительное число хранится в памяти в прямом коде, т.е. в обычном двоичном представлении.

Отрицательное число хранится в дополнительном коде:

1) модуль числа записывается в прямом коде (-1 => 0000 0001)

2) в знаковый разряд записывается 1 (1000 0001)

3) для всех битов, кроме знакового, формируется обратный код заменой 0 на 1 и наоборот (1111 1110)

4) к обратному коду прибавляется 1:

```
                                1111 1110
                               +        1
                                ---------
                                1111 1111  => 0xFF  (-1)
-2 => 2 => 0x02 => 1000 0010 => 1111 1101
                               +        1
                                ---------
                                1111 1110  => 0xFE  (-2)
-3 => 3 => 0x03 => 1000 0011 => 1111 1100
                               +        1
                                ---------
                                1111 1101  => 0xFD  (-3)
-4 => 4 => 0x04 => 1000 0100 => 1111 1011
                               +        1
```

```
                                        ---------
                                        1111 1100  => 0xFC  (-4)
-128 => 128 => 0x80 => 1000 0000 => 1111 1111
                                      +        1
                                        ---------
                                        1000 0000 => 0x80  (-128)
```

Можно еще сказать, что **отрицательное число длиной n бит – это *дополнение до 2^n соответствующего положительного числа*.**

Например для переменной из 4-х бит, -4 – это дополнение числа 4 до $16=2^4$, то есть 12 (в двоичном представлении – 1100). Теперь понятно, что сумма положительного числа и равного ему по абсолютной величине отрицательного (например, 5+(-5)) для 4-х бит всегда равна 16. Это число в двоичной записи равно 10000 и занимает 5 бит. Но в нашей переменной четыре бита, и старший разряд «сваливается» с ее левого конца, оставляя четыре нуля, что и требуется.

Для 8 бит: это дополнение числа 4 до $256=2^8$, то есть $252 = 11111100_2$.

Можно использовать макроопределения, находящиеся в файле **<limits.h>,** чтобы определять допустимый диапазон значений для данных различных типов.

```
/*  limits.h */
#define INT_MAX         0x7FFF
#define INT_MIN         ((int)0x8000)
#define UINT_MAX        0xFFFFU
```

Следует отметить, что в языке С строго не определен диапазон значений для типов int и unsigned int. Размер памяти для переменной типа int определяется длиной машинного слова, которое имеет различный размер на разных машинах и в разных средах. Так, в MS DOS (в BC) размер слова равен 2 байтам, а в Windows-приложениях (в C++ Builder) соответственно 4 байтам. Таким образом, одна и та же программа может правильно работать в C++ Builder и неправильно в BC под DOS.

Предпринимаются попытки сделать так, чтобы программа работала в любой среде на любой машине: типы __int8, __int16, __int32, __int64.

Теперь рассмотрим **вещественные типы данных** (типы данных с плавающей точкой, действительные типы данных).

Тип	Длина	Минимум	Максимум
Вещественные типы данных			
float	4 байта (32 бита)	$3.4 * 10^{-38}$	$3.4 * 10^{38}$
double	8 байт (64 бита)	$1.7 * 10^{-308}$	$1.7 * 10^{308}$
long double	10 байт (80 бит)	$3.4 * 10^{-4932}$	$3.4 * 10^{4932}$

Данные вещественного типа представляются в виде мантиссы M и порядка P числа в двоичной системе счисления: $C = M * 2^P$.

Мантисса выражает значение числа без учета порядка. Мантисса имеет знак, который совпадает со знаком самого числа. Порядок выражает степень основания числа (основание системы счисления), на которое умножается мантисса. Порядок также имеет знак, который не имеет отношения к знаку самого числа.

Нормальной формой вещественного числа называется такая форма, в которой мантисса (без учета знака) находится на полуинтервале [0; 1).

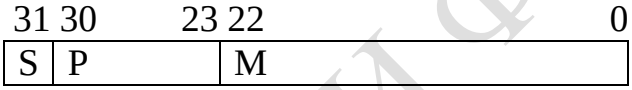
Например, для числа 15.375 нормальной формой будет $1.5375 * 10^1$. Нормальной формой будет и: $0.15375 * 10^2$. Т.е. одно и то же число можно записать разными способами (это недостаток нормальной формы), причем получается, что точка «плавает», отсюда и название «число с плавающей точкой».

Поэтому в информатике используют нормализованную форму записи вещественных чисел, когда мантисса десятичного числа находится на интервале [1;10), а мантисса двоичного числа – на интервале [1;2). В такой форме любое число кроме 0 записывается единственным образом.

Например, для числа 15.375_{10} нормализованной формой будет $1.5375 * 10^1$.

В компьютерах показатель степени принято отделять от мантиссы буквой ‘E’ (exponent). Например, число $1.5375 * 10^1$ записывается 1.5375E1. Это так называемая экспоненциальная форма записи вещественного числа.

Представление вещественных чисел в памяти компьютера (на примере float):



Диапазон чисел, которые можно записать данным способом, зависит от количества бит, отведённых для представления мантиссы и порядка. Точнее, диапазон действительных чисел определяется количеством двоичных разрядов, отведенных под порядок, а их точность – количеством разрядов под мантиссу.

Число бит для хранения мантиссы и порядка зависит от типа данных:

Тип	Размер	Число десятичных цифр	Поля	Знак	Порядок	Мантисса
float	32 бита	7	S-E-F	1	8	23
double	64 бита	15	S-E-F	1	11	52
long double	80 бит	19	S-E-I-F	1	15	1+63

S — знак, E — показатель степени, I — целая часть, F — дробная часть

Вещественное число хранится в памяти компьютера в двоичном виде с нормализованной мантиссой – старшая цифра мантиссы равна 1 (сдвигают влево, один сдвиг увеличивает порядок на 1). Если мантисса всегда нормализована, то старшую 1 можно и не хранить в памяти для экономии бит (это повышает точность представления вещественных чисел). Для long double старшая 1 мантиссы не отбрасывается.

Порядок числа хранится в памяти сдвинутым, т.е. к нему прибавляется число так, чтобы порядок всегда был неотрицательным (для float + 127 = 2^7-1, для double +

1023 = 2^{10}-1, для long double + 16283 = 2^{14}-1). Получаем опять экономию бита – не надо хранить знак порядка.

Рассмотрим число: $15.375_{10} = 1111.011_2$

Алгоритм перевода:

```
1  5 . 3   7  5  =>  1111.011
2^1 2^0 . 2^-1 2^-2 2^-3
(375 * 2) / 1000 = 750  / 1000 <1  =>  0
(750 * 2) / 1000 = 1500 / 1000 ≥1  =>  1 и отнимаем 1
(500 * 2) / 1000 = 1000 / 1000 ≥1  =>  1
```

Нормализованная запись: $1.111011 * 2^{11}$ ($11_2 = 3_{10}$ – точку сдвинули на 3 позиции)

Внутреннее представление для типа float:

S=0

$P = 11_2 = 3_{10} + 127_{10} = 130_{10} = 1000\ 0010_2$

$M = 1110110\ldots0_2$

```
0100 0001 0111 0110 0000 0000 0000 0000
SPPP PPPP PMMM MMMM MMMM MMMM MMMM MMMM
```

Модификатор типа char в языке С является целым типом и используется для представления одного символа. В памяти объект типа char занимает 1 байт и располагается аналогично другим целым типам (1 бит для знака и 7 бит для значения числа для signed char или 8 бит для значения числа для unsigned char). В зависимости от настроек компилятора (или самого компилятора) тип char по умолчанию может быть или signed char, или unsigned char.

Числовым значением объекта типа char является код, соответствующий представляемому символу. Т.е. в char хранятся числовые коды конкретных символов в соответствии с их кодировкой. Закрепление конкретных символов за кодами задается так называемыми кодовыми таблицами.

Система программирования BC 3.1 ориентирована на однобайтовую кодировку символьных данных на базе кодовых таблиц ASCII. Для MS-DOS в нашей стране используется кодовая таблица с номером 866.

В этой таблице в десятичной системе счисления:

маленькие латинские буквы кодируются от ‘a’ = 97 до ‘z’ = 122 (26 букв)
большие латинские: ‘A’ = 65, ‘Z’ = 90
маленькие русские буквы: два диапазона ‘а’ = 160, ‘п’ = 175, ‘р’ = 224, ‘я’ = 239
цифры: ‘0’ = $48_{10} = 30_{16}$, ‘9’ = $57_{10} = 39_{16}$

Для представления символов русского алфавита надо использовать unsigned char, так как коды русских букв превышают величину 127.

Обратить внимание! char = 1 (символ с кодом 1), char = 31 (символ ‘1’).

ASCII

	32	5	53	J	74	_	95	t	116	Й	137	Ю	158	┤	179	╚	200	▌	221	Є	242
!	33	6	54	K	75	`	96	u	117	К	138	Я	159	╡	180	╔	201	▐	222	є	243
"	34	7	55	L	76	a	97	v	118	Л	139	а	160	╢	181	╩	202	▄	223	Ї	244
#	35	8	56	M	77	b	98	w	119	М	140	б	161	╖	182	╦	203	р	224	ї	245
$	36	9	57	N	78	c	99	x	120	Н	141	в	162	╕	183	╠	204	с	225	Ў	246
%	37	:	58	O	79	d	100	y	121	О	142	г	163	╣	184	═	205	т	226	ў	247
&	38	;	59	P	80	e	101	z	122	П	143	д	164	╗	185	╬	206	у	227	°	248
'	39	<	60	Q	81	f	102	{	123	Р	144	е	165	╝	186	╧	207	ф	228	·	249
(	40	=	61	R	82	g	103	|	124	С	145	ж	166	╜	187	╨	208	х	229	·	250
)	41	>	62	S	83	h	104	}	125	Т	146	з	167	╛	188	╤	209	ц	230	√	251
*	42	?	63	T	84	i	105	~	126	У	147	и	168	╙	189	╥	210	ч	231	№	252
+	43	@	64	U	85	j	106	⌂	127	Ф	148	й	169	╘	190	╙	211	ш	232	¤	253
,	44	A	65	V	86	k	107	А	128	Х	149	к	170	╒	191	╘	212	щ	233	■	254
-	45	B	66	W	87	l	108	Б	129	Ц	150	л	171	└	192	╒	213	ъ	234		255
.	46	C	67	X	88	m	109	В	130	Ч	151	м	172	┴	193	╫	214	ы	235		
/	47	D	68	Y	89	n	110	Г	131	Ш	152	н	173	┬	194	╪	215	ь	236		
0	48	E	69	Z	90	o	111	Д	132	Щ	153	о	174	├	195	┘	216	э	237		
1	49	F	70	[	91	p	112	Е	133	Ъ	154	п	175	─	196	┌	217	ю	238		
2	50	G	71	\	92	q	113	Ж	134	Ы	155	▒	176	┼	197	█	218	я	239		
3	51	H	72	]	93	r	114	З	135	Ь	156	▓	177	╞	198	▄	219	Ё	240		
4	52	I	73	^	94	s	115	И	136	Э	157	▒	178	╟	199	▄	220	ё	241		

Состав отображаемых символов ASCII (code page 866)

Есть еще так называемый **«пустой» тип void**. Объектов типа void не существует. Этот тип используется либо для указания на то, что функция не возвращает значения, либо того, что функция не имеет параметров, либо для указателей на объекты неизвестного типа. Более подробно все это мы рассмотрим позднее, когда будем изучать указатели и функции.

6. КОНСТАНТЫ

Термин константа относится к значению, которое не может быть изменено. В языке С константы могут быть четырех типов: целые константы, действительные константы (вещественные, с плавающей точкой), символьные константы и строковыми константы.

Строковая константа – последовательность символов, заключенная в кавычки (например, “123”, “Введите число”).

Символьная константа – символ, заключенный в апострофы (например, ‘a’, ‘!’).

Целая константа – это десятичное, восьмеричное или шестнадцатеричное целое число:

1) **десятичная константа** – последовательность цифр, не начинающаяся с 0 (например, 100, 78).

2) **восьмеричная константа** состоит из обязательного нуля и одной или нескольких восьмеричных цифр. Среди цифр должны отсутствовать восьмерка и девятка, так как эти цифры не входят в восьмеричную систему счисления (например, 077, 05).

3) **шестнадцатеричная константа** начинается с обязательной последовательности 0x или 0X и содержит одну или несколько шестнадцатеричных цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (например, 0xFF, 0x9A).

Вещественная константа – может быть в двух форматах:

1) **с фиксированной точкой (десятичный формат)**:
[цифры].[цифры] (например, 1., 2.0, 7.5).

2) **с плавающей точкой (экспоненциальный формат):**
[цифры]E|e[+|-]цифры (например, 4e-7=$4*10^{-7}$, 5.1e+8=$5.1*10^{8}$, 2e5=$2*10^{5}$).

Если требуется сформировать **отрицательную константу**, то используют знак '-' перед записью константы (который будем называть унарным минусом). Например: -0x2A, -088, -16, -5.7, -2e5.

Для представления констант можно использовать макроопределения. Макроопределение ассоциирует имя с определенным значением.

Типизированные константы – это переменные, значение которых нельзя изменить. Можно создать такую константу, описав переменную с добавлением ключевого слова const перед типом.

```
#define MAX_COST 100 // макроопределение
void main() {
  const int count = 25;  // типизированная константа
  printf("Стоимость всех вещей: :d", MAX_COST * count);
}
```

Лучше применять не макроопределения, а типизованные константы, так как макроопределения являются просто текстовыми подстановками и могут не давать компилятору достаточной информации о представлении данной величины. Кроме этого макроимена нельзя использовать в окне просмотра отладчика (например, в составе выражения). А вот имена типизированных констант во время отладки доступны.

Удобство именованных констант заключается в минимальных переделках программы, связанных с изменением этих констант (например, размерности массивов, константы точности). Если одна и та же константа используется в различных местах программы, то достаточно изменить одну строку программы с объявлением той или иной константы и не менять другие операторы, использующие эту константу.

Кроме естественного представления числовых констант в виде целого или вещественного числа мы уже использовали **«префиксы»** – добавки в начале константы (0x – для 16-ричного числа и 0 для восьмеричного числа).

Также в записи константы можно использовать «суффиксы» – добавки в конце константы. **«Суффиксы»** определяют тип константы:

- 5U,5u – целое число без знака (суффикс – u или U, от Unsigned);
- 5L,5l – длинное целое число (суффикс – l или L, от Long).

Использование «суффиксов» может понадобиться для определения преобразований, которые должны быть выполнены, если константа используется в выражениях.

7. ПЕРЕМЕННЫЕ

Чтобы выделить память для данных конкретного типа, надо определить (описать) переменную. Вначале указывается тип данных, а затем имя переменной (идентификатор).

```
int i = -1, j, k = 0;
char a = 'z';
float t;
```

Определяя переменную, можно присвоить ей начальное значение – **инициализация переменной при описании**.

Можно также определить несколько переменных одного типа, перечислив их через запятую.

Как и любой оператор языка С, определение переменных должно заканчиваться точкой с запятой (;).

Идентификатор переменной – имя переменной. Для обозначения имени переменной разрешается использовать строчные и прописные буквы латинского алфавита, цифры и символ подчеркивания '_'. Первым символом должна быть обязательно буква или символ подчеркивания.

Например, num, cat_1, _total – правильные идентификаторы, а num!, 1cat – неправильные.

Два идентификатора, для образования которых используются совпадающие строчные и прописные буквы, считаются различными. Например: abc, ABC, A128B, a128b.

Ключевые слова – это зарезервированные идентификаторы, которые наделены определенным смыслом. Их можно использовать только в соответствии со значением, известным компилятору языка С. Например, int, while, for, if, else.

Ключевые слова не могут быть использованы в качестве идентификаторов.

В языке С **все переменные должны быть описаны** (нет принципа умолчания). Это означает, что мы должны привести список всех используемых переменных и указать тип каждой из них. Переменные можно описывать по мере необходимости, но обязательно до их использования в тех или иных исполняемых операторах. Повторное объявление переменных с одинаковыми именами считается ошибкой (дублирование имен переменных).

Вам пока лучше для наглядности размещать операторы объявления переменных в начале программы.

Имя переменной нужно давать осмысленно. Следует учитывать, что конкретные реализации компиляторов ограничивают длину имени переменных.

Обратить внимание! При описании переменной под нее выделяется память. Если мы при описании инициируем переменную, то значение переменной становится таким, как надо нам (нужное зам значение записывается в выделенную память). В противном случае переменная принимает случайное значение (то, которое было в памяти, выделенной под переменную).

8. ЭЛЕМЕНТАРНЫЙ ВВОД И ВЫВОД ИНФОРМАЦИИ

Особенностью языка С является отсутствие специальных операторов ввода-вывода. Вместо этого используются библиотечные функции. Эти функции описаны в файле <stdio.h>, который при их использовании надо подключить в начале программы (#include <stdio.h>). Будем рассматривать функции для вывода на экран и ввода с клавиатуры.

Есть две наиболее универсальных функции, которые используются для вывода и ввода: printf() и scanf(). Функции printf() и scanf() работают во многом одинаково – каждая использует форматную (управляющую) строку и список аргументов (функции имеют переменное количество аргументов).

```
printf("управляющая строка", аргумент1, аргумент2, …);
scanf("управляющая строка", аргумент1, аргумент2, …);
```

Для вывода информации используется функция printf(). При вызове функции printf() обязательно передается в качестве первого аргумента форматная строка. Функция просматривает строку и выводит каждый символ так, как он есть, буквально, пока не встретит спецификацию преобразования. Это указание функции printf() типа переменной, которую мы хотим напечатать, и формата ее вывода.

Спецификация преобразования для функции printf() начинается со знака процента (%) и имеет следующий формат:

```
%[флаг][ширина][.точность][размер]тип
```

Каждая спецификация заставляет функцию printf() искать дополнительный аргумент, который затем преобразуется и выводится в соответствии с заданным преобразованием вместо спецификации преобразования. Число дополнительных аргументов в вызове printf() должно соответствовать числу спецификаций.

Вывод: функция printf() использует управляющую строку, чтобы определить, сколько всего аргументов и каковы их типы.

```
printf("Введите число");
printf("Вы ввели два числа: %d и %d", a, b);
```

Основные спецификации преобразования для функции printf():

Элемент	Обязательный	Символ	Значение
флаг	нет	–	прижать число при выводе к левому краю поля
		+	всегда выводить знак (+ или –)
ширина	нет		минимальная ширина поля (если результат больше этой ширины, то он печатается полностью, игнорируя ширину поля)
точность	нет		максимальное число знаков после точки
размер	нет	h	короткое целое (short)
		l	длинное число (long или double)
		L	число типа long double
тип	да	d	представить в виде десятичного целого числа со знаком
		u	представить в виде десятичного целого числа без знака
		x / X	представить в виде 16-ричного целого числа без знака

			(буквы в нижнем/верхнем регистре)
		f	число с плавающей точкой в форме [-]ddd.ddd
		c	вывести одиночный символ
		s	вывести строку
		p	вывести указатель

```
int a = 5, a4 = 10;
unsigned long b = 9;
char c = 'h';
float d = -7.779;
double e = 2222.5555;

printf("\na=%d\na4=%4d\na4=%-4d\na4=%X", a, a4, a4, a4);
// вывод "a=5"
//       "a4=  10"
//       "a4=10  "
//       "a4=A"

printf("\nb=%lu c=%c", b, c);
//       "b=9 c=h"

printf("\nd=%f e=%lf", d, e);
//       "d=-7.779000 e=2222.555500"

printf("\nd=%7.2f  d=%.1f  e=%9.2lf", d, d, e);
//       "d=  -7.78  d=-7.8  e=  2222.56"
```

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

```
printf("%05d", 15);  // результат 00015
```

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

```
printf("%5s", "1234567890");  // результат 1234567890
printf("%.5s", "1234567890"); // результат 12345
```

Выравнивание выдачи производится по правому краю поля. Если мы хотим выравнивать по левому знаку поля, то сразу за знаком % следует поставить знак минуса.

Задание фиксированной ширины полей оказывается полезным при печати данных столбиком (например, матриц):

```
printf("%d %d\n",val1,val2);
printf("%d %d\n",val3,val4);
```

Результат выглядит так:

```
333 22
```

```
4 123
```

Эти же данные можно представить в улучшенном виде, если задать достаточно большую фиксированную ширину поля:

```
printf("%5d %5d\n",val1,val2);
printf("%5d %5d\n",val3,val4);
```

Результат будет выглядеть так:

```
  333    22
    4   123
```

Вторым, третьим и т.д. аргументами функции printf() могут быть не только переменные, но и константы, выражения (вычисляются перед выводом на печать), вызовы функции (перед печатью функция вызывается, затем результат, который она возвращает, печатается).

```
printf("\n%d %d %d %d", a, 55, (a+100)*2, func());
```

Обратить внимание! Надо строго следить за соответствием типа спецификатора и типом данных, выводимых на печать.

```
printf("a=%d b=%u", -100, -100); // вывод: a=-100 b=65436
                                 // 65436 = 2^16-100 = 65536-100
```

Escape-последовательности. Обратная косая черта (\) имеет в языке С специальное значение. Ее называют «escape-символом» и применяют для представления символов, которые нельзя непосредственно ввести с клавиатуры. Несмотря на то, что специальные символы записываются с помощью двух символов, фактически определяется однобайтовая символьная константа.

Последовательность	Название	Функция
\n	новая строка	переход к началу новой строки
\t	табуляция	переход к следующей позиции табуляции
\\	обратная черта	выводит обратную косую черту
\”	кавычка	выводит двойную кавычку
\%	процент	выводит знак процента

Первые две последовательности используются для вывода специальных символов форматирования, которые нельзя ввести с клавиатуры (их коды 10 и 9). Последние три последовательности используются для вывода символов, которые нельзя явно указать в строке вывода функции printf().

Функция printf() будет преобразовывать escape-последовательности, входящие в строку формата, в соответствующие коды, что расширяет возможности управления форматом. Вы уже виделись, как применялась последовательность ‘\n’ в вызовах printf().

Escape-последовательность – это один символ, его можно использовать как символьную константу (char a = ‘\n’).

В кодовой таблице 866 есть группа **символов псевдографики**. С их помощью можно печатать одинарные и двойные контуры таблиц.

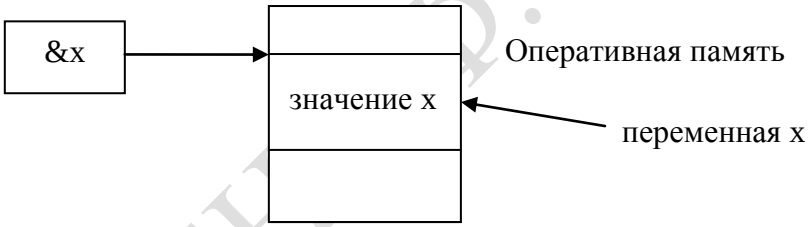
```
printf("\n╔═══╗");
printf("\n║   ║");
printf("\n╚═══╝");
```

Для ввода такого символа надо при нажатой клавише Alt набрать десятичный код символа на правой числовой клавиатуре. Коды символов можно узнать также в DN, нажав клавиши Ctrl-B.

Можно вывести символы псевдографики, как и любой другой символ, и так: например, символ ‘‖’ с кодом 186 (0xBA или 272_8)

```
printf("\xBA");
printf("\272");
```

Для ввода информации используется функция scanf(). Точно так же, как printf(), эта функция ожидает в качестве аргумента форматную строку, содержащую одну или несколько спецификаций формата, указывающих формат и тип данных, которые должны быть прочитаны. Дополнительные аргументы, следующие за строкой формата, должны быть адресами переменных, в которых данные будут сохраняться. Если есть переменная x, то адрес ее можно получить с помощью операции взятия адреса &, т.е. &x.



Управляющая строка функции scanf() может содержать три вида символов: спецификаторы формата, пробелы и другие символы.

Если данные, прочитанные с помощью scanf(), не соответствуют строке формата, функция может вести себя непредсказуемо.

Спецификация преобразования для функции scanf() начинается со знака процента (%) и имеет следующий формат:

```
%[*][ширина][размер]тип
```

Основные спецификации преобразования для функции scanf():

Элемент	Обязательный	Символ	Значение
*	нет	*	подавляет присваивание следующего введенного поля
ширина	нет		максимальная ширина поля
размер	нет	h	короткое целое (short int)
		l	длинное число (long или double)
		L	число типа long double
тип	да	d	вводится целое число со знаком (int *)
		D	вводится длинное целое число со знаком (long *)
		f	вводится число с плавающей точкой (float *)

		u	вводится целое число без знака (unsigned int *)
		U	вводится длинное целое число без знака (unsigned long *)
		c	вводится одиночный символ
		s	вводится строка

```
int a, aa, bb;
unsigned long b;
char c;
float d;
double e;
scanf("%d %lu", &a, &b);
scanf("%2d", &aa);                  // при вводе 125 введется aa = 12
scanf("%d %*d %d", &aa, &bb); // при вводе 1 2 5 введется aa = 1 bb=5
scanf("%c", &c);
scanf("%f %lf", &d, &e);
```

d	int
u	unsigned int
hd	short
hu	unsigned short
ld, D	long
lu, U	unsigned long
f	float
lf	double
Lf	long double
c	char

Разделителями между двумя вводимыми числами являются символы пробела, табуляции или новой строки. Функция scanf() использует введенные при вводе пробелы, символы табуляции (‘\t’) и новой строки (‘\n’) для разбиения входного потока символов на отдельные поля. Она согласует последовательность спецификаций преобразования с последовательностью полей, опуская упомянутые специальные знаки между ними. Т.е. при вводе пробелы, ‘\t’ и ‘\n’ разделяют поля, при этом в управляющей строке эти символы просто игнорируются (поэтому их писать там совсем не обязательно).

Исключением является спецификация %c, обеспечивающая чтение каждого следующего символа, включая пробел и ‘\n’.

```
scanf("%d %d %d", &a, &b, &c);
scanf("%d%d%d", &a, &b, &c);     // = предыдущему
```

Вводить можно по любому: или через пробел в одной строке, или по одному в строке, или по два в строке. Если ввести какой-либо другой символ, то на нем ввод закончится: для ″1 2, 3″ введется только 1 и 2.

После последнего %d не должно быть пробела или ‘\n’, т.к. тогда функция считает, что будет еще ввод. Если любой другой символ (или несколько), все работает нормально (реально символы можно вводить, можно не вводить).

Если в управляющей строке встречаются обычные символы (кроме спецификаций, пробелов и новых строк), то считается, что эти символы должны совпадать с очередными символами во входном потоке.

```
scanf(“%d,%d,%d”, &a, &b, &c);
```

Если ввести одно число или два, то ввод закончится. Если ввести все три через пробел, то введется только первое. Если ввести “1 , 2 , 3”, то введется только первое. Надо вводить все три через запятую, т.е. “1,2,3” (или “1,” + ввод + “2,” + ввод + “3” + ввод).

Знак * после % и перед кодом формата дает команду прочитать данные указанного типа, но не присваивать это значение. Так, scanf("%d%*c%d", &i, &j); при вводе 50+20 присвоит переменной i значение 50, переменной j – значение 20, а символ + будет прочитан и проигнорирован.

Обратить внимание! По спецификации ‘%s’ функция scanf() вводит в строку символы до первого разделителя, в том числе и пробела. Т.е. с помощью этой функции нельзя ввести строку, в которой есть пробелы.

К недостаткам функции scanf() относится невозможность выдачи приглашения к вводу, т. е. приглашение должно быть выдано до обращения к функции scanf().

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

```
char s[10], t[10];
scanf ("%[0123456789]%s", s, t);
```

Введем следующий набор символов: «123abc45». Получим: s = “123”, t = ”abc45”.
Так как ‘a’ не входит в множество поиска (оно состоит только из цифр), то ввод по первому спецификатору формата прерывается и начинается ввод по второму спецификатору формата.

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

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

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

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

Функции print() и scanf() относятся к так называемым функциям **форматированного (форматного) ввода- вывода.**

Функции неформатированного ввода-вывода работают с отдельными символами или строками символов. Описаны эти функции также в файле <stdio.h>.

Для ввода символа используется функция getchar(), которая не имеет аргументов. Функция ожидает, пока не будет нажата клавиша, а затем возвращает значение этой клавиши. Кроме того, при нажатии клавиши на клавиатуре на экране дисплея автоматически отображается соответствующий символ. Эта функция возвращает целое число, соответствующее коду введенного символа. Однако возвращаемое значение можно присвоить переменной типа char, что обычно и делается, так как символ содержится в младшем байте (старший байт при этом обычно обнулен.)

Для вывода символа используется функция putchar(int). Несмотря на то, что эта функция объявлена как принимающая целый параметр, она обычно вызывается с символьным аргументом. На самом деле из ее аргумента на экран выводится только младший байт.

```
int c = getchar();    // char c = getchar();
int cc = 'k';         // char cc = 'k';
putchar(cc);          // putchar(cc);
```

Использование getchar() может быть связано с определенными трудностями. Во многих библиотеках компиляторов эта функция реализуется таким образом, что она заполняет буфер ввода до тех пор, пока не будет нажата клавиша <ENTER>. Это называется построчно буферизованным вводом. Чтобы функция getchar() возвратила какой-либо символ, необходимо нажать клавишу <ENTER>. Кроме того, эта функция при каждом ее вызове вводит только по одному символу. Поэтому сохранение в буфере целой строки может привести к тому, что в очереди на ввод останутся ждать один или несколько символов.

Так как getchar(), может оказаться неподходящей в интерактивной среде, то для чтения символов с клавиатуры может потребоваться другая функция. В стандарте языка С не определяется никаких функций, которые гарантировали бы интерактивный ввод, но их определения имеются буквально в библиотеках всех компиляторов С.

У двух из самых распространенных альтернативных функций getch() и getche() имеются следующие прототипы:

```
int getch(void);
int getche(void);
```

Функция getch() ожидает нажатия клавиши, после которого она немедленно возвращает значение. Причем символ, введенный с клавиатуры, на экране не отображается. Функция getche() отличается от getch() тем, что символ на экране отображает.

Ввод-вывод строк символов выполняют соответственно функции gets() и puts(), в качестве параметра которых указывается адрес строки. С помощью функции gets() можно ввести строку с пробелами.

При смешанном вводе числовых и символьных данных могут возникнуть проблемы: вроде как функции scanf() или gets() ничего не вводят. Это не так, они

вводит то, что осталось в специальном буфере ввода после предыдущего ввода (например, gets() вводит ‘\n’ после ввода числа, или scanf() вводит пробел и остальные символы, которые остались после ввода строки с пробелом по ‘%s’).

Если такое происходит, перед проблемным вызовом функции ввода надо вызвать **функцию очистки буфера ввода**: fflush(stdin); или flushall(); (обе функции описаны в <stdio.h>).

```
int a;
char s[60];
scanf("%d",&a); // вводим 5 и a=5
// fflush(stdin);
// flushall();
gets(s);  // ничего не предлагает вводить, т.к. в буфере есть “\n”,
                что и обрабатывается функцией и в результате s=””.
```

```
int a;
char s[60];
char ss[60];
scanf("%s",s);  // вводим “abc 5dddd”
//fflush(stdin);
scanf("%d",&a); // ничего не предлагает вводить, но a=5
//fflush(stdin);
gets(ss);       // ничего не предлагает вводить, но ss=”dddd”
```

Есть еще интересные **функции**, которые используются **при выводе информации на экран**:

void clrscr(void);	осуществляет очистку экрана
void gotoxy(int x, int y);	перемещает курсор в позицию x строки y
void cprintf(char *format, ...);	выполняет то же самое, что и printf(), но выводит информацию, используя установленный цвет фона и цвет символа
void textcolor(int color);	установка цвета символа с кодом color
void textbackground(int color);	установка цвета фона с кодом color

Функции не изменяют цвет уже выведенных символов. Их влияние распространяется на все последующие выводы с помощью функции cprintf().

При установке цвета допускается использовать шестнадцать цветов символа с кодами 0...15, и восемь цветов фона с кодами 0...7.

Для удобства работы с цветами в <conio.h> определены мнемонические имена для цветов:

```
enum COLORS {
  /* цвета для символов и фона */
    BLACK         /* черный */,          BLUE      /* синий */,
    GREEN         /* зеленый */,         CYAN      /* салатовый */,
```

```
    RED           /* красный */,         MAGENTA   /* малиновый */,
    BROWN         /* коричневый */,      LIGHTGRAY /* светло-серый */,
  /* цвета только для символов */
    DARKGRAY      /* темно-серый */,     LIGHTBLUE /* ярко-синий */,
    LIGHTGREEN    /* ярко-зеленый */,    LIGHTCYAN /* ярко-салатовый */,
    LIGHTRED      /* ярко-красный */,    YELLOW    /* желтый */,
    LIGHTMAGENTA /* ярко-малиновый */, WHITE     /* белый */
};
```

9. ВЫРАЖЕНИЯ И ОПЕРАЦИИ

Выражение – это последовательность операндов и операций. Значения, участвующие в операциях, называются операндами. Операнды – это переменные, константы либо другие выражения. Выражение может состоять из одной или более операций. Когда выражение содержит более чем одну операцию, порядок их выполнения определяется соотношением приоритетов – операция с более высоким приоритетом выполняется раньше. Операции с одинаковым приоритетом обрабатываются в соответствии с их ассоциативностью (порядком выполнения). Ассоциативность бывает двух видов: слева направо и справа налево:

`a=b+c*d` => умножение имеет более высокий приоритет, чем сложение, у присваивания самый низкий приоритет => `a=(b+(c*d))`

`a=b+c-d` => присваивание последнее, сначала сложение, потом – вычитание, т.е. у них приоритет одинаковый, а ассоциативность (порядок выполнения) – слева направо => `a=((b+c)-d)`

Каждая **операция возвращает какое-то значение**. Например, операция 5+2 вернёт семь.

Операция	Описание	Пример	Приоритет/ Ассоциативность
Первичные и постфиксные			
[]	индекс массива	mas[5]	16, слева направо
()	вызов функции	puts(msg)	16, слева направо
.	элемент структуры	time.hour	16, слева направо
->	элемент структуры	time->hour	16, слева направо
++	постфиксное приращение	num++	15, слева направо
--	постфиксное уменьшение	num--	15, слева направо
Унарные операции			
++	префиксное приращение	++num	14, справа налево
--	префиксное уменьшение	--num	14, справа налево
sizeof	размер в байтах	sizeof(num)	14, справа налево
(тип)	преобразование типа	(float)i	14, справа налево
~	побитовое НЕ	~visible	14, справа налево
!	логическое НЕ	!EOF	14, справа налево
-	унарный минус	-i	14, справа налево
&	адрес	&num	14, справа налево
*	разыменование	*ptrNum	14, справа налево

Бинарные и тернарные операции			
Мультипликативные			
*	умножение	a*10	13, слева направо
/	деление	a/10	13, слева направо
%	взятие по модулю	a%10	13, слева направо
Аддитивные			
+	сложение	a+10	12, слева направо
-	вычитание	a-10	12, слева направо
Побитовый сдвиг			
<<	сдвиг влево	a<<1	11, слева направо
>>	сдвиг вправо	a>>1	11, слева направо
Операции отношения			
<	меньше	i<n	10, слева направо
<=	меньше или равно	i<=10	10, слева направо
>	больше	i>0	10, слева направо
>=	больше или равно	i>=0	10, слева направо
Равенство			
==	равно		9, слева направо
!=	не равно		9, слева направо
Битовые			
&	побитовое И	a&b	8, слева направо
^	побитовое исключающее ИЛИ	a^b	7, слева направо
|	побитовое ИЛИ	a|b	6, слева направо
Логические			
&&	логическое И	a>0 && b>0	5, слева направо
||	логическое ИЛИ	a!=0 || b==0	4, слева направо
Условия			
?:	при условии	a>b ? 1: 0	3, справа налево
Присваивание			
=	присваивание	x=10	2, справа налево
*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	присваивание произведения, частного, остатка и т.д.	x+=10 x%=10	2, справа налево
,	запятая	x=2, y=3;	1, слева направо

По числу участвующих операндов, операции делятся на три группы: унарные (у операции один операнд), бинарные (в операции участвуют два операнда), тернарные (в операции участвуют три операнда). В языке С всего одна операция, которая принимает три операнда – условная операция.

По типу выполняемой операции различают: арифметические, поразрядные логические, присваивания, операции отношения и др.

Рассмотрим основные и специфические операции языка С более подробно.

9.1. Арифметические операции

В языке С имеется стандартный набор арифметических операций: +, -, /, *. Ассоциативность арифметических операций принята слева направо.

Обратить внимание! Деление целых чисел в С дает всегда целое число. Если же хотя бы один из операндов вещественный, результат также будет вещественным. В языке С принято правило, согласно которому дробная часть у результата деления целых чисел отбрасывается. Это действие называется «усечением», т.е. результат деления целых чисел **округляется не до ближайшего целого, а всегда до меньшего целого числа.**

```
3/2   // результат = 1
3./2  // результат = 1.5
```

```
int a = 2, b = 1;
float sr = (a+b) / 2;   // 1
      sr = (a+b) / 2.;  // 1.5
```

К арифметическим операциям относится и операция взятия остатка %. Данная операция делит первый операнд на второй и берёт остаток от деления:

```
int x=25;
int y=10;
int z;
z=x%y;   // z = 5
```

Операция % определена только над целыми операндами. Это так называемое деление по модулю: результат – остаток от деления. Аналогом в Паскале была операция mod.

9.2. Операция изменения знака

Знак минус используется для указания или изменения алгебраического знака некоторой величины.

```
a = -15;
b = -a;   // b = 15
```

Когда знак используется подобным образом, данная операция называется «унарный минус».

`x = -x; //` Операция изменяет алгебраический знак x. Не надо для изменения знака умножать на -1: x*(-1) или вычитать из 0: 0-x.

9.3. Операции инкремента и декремента

Специфическими для С являются операции инкремента ++ и декремента --.

Инкремент – увеличение переменной на единицу. Новое значение сохраняется в переменной.

Декремент – уменьшение переменной на единицу. Новое значение сохраняется в переменной.

Эти операции могут применяться только к переменным. Операнд инкремента и декремента может иметь целый или вещественный тип или быть указателем.

Операции инкремента и декремента могут записываться как перед своим операндом (*префиксная* форма записи), так и после него (*постфиксная* запись).

В префиксной форме (плюсы стоят до переменной) сначала происходит увеличение переменной на единицу, и потом это увеличенное значение участвует в выражении. При постфиксной форме (плюсы стоят после переменной) переменная сначала участвует в выражении, а только затем она увеличивается.

```
int x = 5;
int y;
y = ++x;    // y = 6, x = 6
y = --x;    // y = 5, x = 5
y = x++;    // y = 5, x = 6
y = x--;    // y = 6, x = 5
```

9.4. Операция присваивания

В языке С знак равенства не означает «равно». Он означает операцию присваивания некоторого значения. Операнду, находящемуся в левой части, присваивается значение операнда, стоящего в правой части операции: x = 5; (теперь в переменной x хранится значение 5).

Левым операндом должна быть переменная. Нельзя записать: 5 = x;

В языке С можно писать в одном предложении сразу несколько операций присваивания. Например, обнуление трех переменных можно сделать так:

```
int x, y, z;
x = y = z = 0;
```

Присваивания выполняются справа налево: сначала переменная z получает значение 0, затем переменная y и наконец x.

В языке С принято следующее правило: любое выражение с оператором присваивания, заключенное в круглые скобки, имеет значение, равное присваиваемому. Например, выражение (a = 2 + 5) имеет значение 9. После этого можно записать другое выражение, например ((a = 2 + 5) < 10), которое всегда будет истинным.

Можно писать даже так:

```
x = (a = 2 + 5) * (b = 1 + 1);      // a=7 b=2 x=14
```

В языке С имеется еще несколько других операций присваивания, которые отличаются от описанной операции – это так называемые комбинированные операции присваивания.

9.5. Арифметические операции с присваиванием: +=, -=, *=, /=, %=

Данные операции позволяют сократить код:

```
int x = 5;
x += 5;             // две последние строки эквивалентны
x = x + 5;
```

Арифметические операции с присваиванием используются, когда необходимо внести изменения в переменную, используя при этом значение, которое хранится в этой переменной в данный момент.

Такие операции присваивания есть еще и для операций сдвига и поразрядных логических операций: <<=, >>=, &=, ^=, |=.

9.6. Поразрядные логические операции

Поразрядные логические операции в порядке увеличения приоритета: | ^ & ~.

|	поразрядное логическое ИЛИ (OR)
^	поразрядное сложение по модулю 2 (XOR – исключающее ИЛИ)
&	поразрядное логическое И (AND)
~	поразрядная инверсия

Эти операции возвращают значения такого же типа, как и их операнды : если x и y целые, то и результат x&y тоже будет целый.

При выполнении этих операций вычисления ведутся над двоичным представлением операндов. Каждый бит результата определяется по битам операндов так:

Операнд 1	Операнд 2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Унарная инверсия требует единственного операнда справа от знака ~. Результат образуется поразрядной инверсией всех битов операнда.

При выполнении данных операций не имеет смысла смотреть на десятичное представление чисел.

```
int  i = 0x45FF,      //    i= 0100 0101 1111 1111
     j = 0x00FF;      //    j= 0000 0000 1111 1111
int  k;
k = i ^ j;       // k=0x4500 = 0100 0101 0000 0000
k = i | j;       // k=0x45FF = 0100 0101 1111 1111
k = i & j        // k=0x00FF = 0000 0000 1111 1111
k = ~i;          // k=0xBA00 = 1011 1010 0000 0000
```

Данные операции используются тогда, когда необходимо обрабатывать биты, а не числа: например, при кодировании или сжатии информации. Также они используются для проверки значений конкретных битов числа: самая быстрая проверка на четность/нечетность числа – проверяем последний бит, битовые поля признаков.

```
int x = 11;          // x & 1  ≡  x & 0x1  ≡  x & 0x0001

   …0000 1011
 & …0000 0001
   ----------
   …0000 0001
```

Вывод: x&1, если =1, то число нечетное

```
int x = 12;

   …0000 1100
 & …0000 0001
   ----------
```

…0000 0000

Вывод: x&1, если =0, то число четное

9.7. Операции сдвига: >> и <<

Операции сдвига осуществляют смещение операнда влево << или вправо >> на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами.

При сдвиге влево << правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо >> метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип unsigned, то свободные левые биты устанавливаются в нуль. В противном случае они заполняются копией знакового бита (это так называемое размножение знака).

Число двоичных разрядов для сдвига может быть задано только константой или константным выражением, т.е. выражением, целиком состоящим из констант. Нельзя написать: int x, y = 2; x = x >> y;

Результат операции сдвига не определен, если второй операнд отрицательный.

Операции сдвига – это наиболее быстрые способы умножения и деления на 2, 4, 8 и т. д. (т.е. на степень двойки).

Сдвиг влево << соответствует умножению первого операнда на степень числа 2, равную второму операнду, а сдвиг вправо >> соответствует делению первого операнда на 2 в степени, равной второму операнду.

```
unsigned char i=6, k;   // i = 0000 0110
k = i<<1;               // k = 0000 1100 = 12 = 6 * 2^1
k = i<<2;               // k = 0001 1000 = 24 = 6 * 2^2
k = i>>1;               // k = 0000 0011 = 3 = 6 / 2^1
char  j=-4, n;          // j = 1111 1100
n = j<<1;               // n = 1111 1000 = -8 = -4 * 2^1
n = j>>2;               // n = 1111 1111 = -1 = -4 / 2^2
unsigned char a=252;    // a = 1111 1100
k = j>>2;               // k = 0011 1111 = 63 = 252 / 2^2
```

9.8. Логические операции и операции отношения

Данные операции используются при формировании логических выражений. Логическое значение в языке С может иметь только два значения: 1, если оно ИСТИННО; и 0, если оно ЛОЖНО (т.е. тип результата логической операции – int).

Строго говоря, в языке С значению ИСТИНА соответствует не только значение 1, но и любое другое ненулевое значение: 0 – ЛОЖЬ, все что <> 0 – ИСТИНА.

Операции отношения		
>	больше	1, если операнд слева больше, чем справа, иначе 0
<	меньше	1, если операнд слева меньше, чем справа, иначе 0

==	равно	1, если операнд слева равен операнду справа, иначе 0
>=	больше или равно	1, если операнд слева больше или равен, чем справа, иначе 0
<=	меньше или равно	1, если операнд слева меньше или равен, чем справа, иначе 0
!=	не равно	1, если операнд слева не равен операнду справа, иначе 0
Логические операции		
&&	логическое И	1, если оба операнда равны 1, иначе 0
||	логическое ИЛИ	1, если один из операндов равен 1, иначе 0
!	логическое НЕ	унарная: 1, если операнд равен 0, иначе 0

Приоритет операций по убыванию: !, (>, <, >=, <=), (==, !=), &&, ||.

```
if (x!=y && x==z) {
  // выполняется, если x не равно y и x равно z
}
if (!x || y<z) {            // if (x==0 || y<z) {
  // выполняется, если не x или y меньше z
}
if (x && y) {               // if (x!=0 && y!=0) {
  // выполняется, если x и y
}
```

Обратить внимание! Сравнение на равно это ‘= =’, а не ‘=’. Если напишите ‘=’ при сравнении, ошибки не будет, а будет простое присваивание.

```
if (x = 10) { // x = 10 и войдем в if, т.к. результат условия 10 (ИСТИНА)
  операторы
}
if (x = 0) {  // x = 0 и не войдем в if, т.к. результат условия 0 (ЛОЖЬ)
  операторы
}
```

9.9. Условная операция «? :»

Условная операция ? : – единственная тернарная (то есть принимающая три операнда) операция в языке С. Это специфичная операция для языка С. Формат операции:

```
операнд1 ? операнд2 : операнд3;
```

В первом операнде записывается логическое выражение. Оно вычисляется. Если получается ИСТИНА, результатом всей операции условия является второй операнд, если ЛОЖЬ – результат операции есть третий операнд. Использовать эту операцию можно вместо простого ветвления if else.

```
flag = x > y ? 1 : 0;
```

Приведённый код эквивалентен следующему:

```
if (x>y)
```

```
  flag = 1;
else
  flag = 0;
```

или можно и так записать:

```
x > y ? (flag = 1) : (flag = 0);
```

Можно и так писать:

```
d = x > y ? (flag = 1) : (flag = 0); // d будет таким же как flag
```

9.10. Операция последовательного вычисления

Операция последовательного вычисления обозначается запятой (,) и используется для вычисления двух и более выражений там, где по синтаксису допустимо только одно выражение. Эта операция вычисляет два операнда слева направо.

```
x=0, a=b;
```

Операнды могут быть любых типов. Результат операции имеет значения и тип второго операнда.

9.11. Операция определения требуемой памяти в байтах sizeof

Результатом операции sizeof является размер в байтах типа или переменной. Тип операнда – значение любого типа или имени типа. Тип результата – unsigned int. Используется как sizeof (выражение) или sizeof (имя типа).

```
int a, b, c, d;
char z;
a = sizeof(int);           // получим 2
b = sizeof(unsigned long); // получим 4
c = sizeof(short);         // получим 2
```

С помощью этой операции можно определить размер не только типа, но и любой переменной (в том числе и массива).

```
d = sizeof(a);             // получим 2
d = sizeof(z);             // получим 1
```

Число элементов в массиве целых чисел, определяемое как число байт в массиве, поделенное на число байт, занимаемых одним элементом массива.

Для целого одномерного массива число элементов определяется так:

```
 n = sizeof(mas_name) / sizeof(int);
```

9.12. Операция приведения типа (type)

При вычислении выражений тип каждого операнда может быть преобразован к другому типу.

Преобразование одного типа в другой называется неявным, если оно должно выполняться компилятором автоматически. Во многих языках программирования неявное преобразование типов ограничено ситуациями, где в принципе не происхо-

дит потери информации (например, целое число может быть преобразовано в действительное, но не наоборот).

Преобразование называется явным, если для его задания программист должен специально что-то написать.

Неявное преобразование типов выполняется автоматически при смешивании в одном выражении операндов разных типов.

Также язык С позволяет выполнять и явное приведение типов, используя операцию приведения типа (type), type – любой допустимый тип языка С.

Приведение типов это изменение (преобразование) типа объекта. Для выполнения преобразования необходимо перед объектом записать в скобках нужный тип:

```
(имя-типа) операнд
```

```
int i=5, j=2;
float x=0;
x = (float)i + 2;     // x = 7
x = i / j;            // x = 2
x = (float)i / j;     // x = 2.5
```

В этом примере целая переменная i с помощью операции приведения типов приводится к плавающему типу, а затем уже участвует в вычислении выражения.

Выражению с приведением типа не может быть присвоено другое значение.

10. ОПЕРАТОРЫ УПРАВЛЕНИЯ ВЫЧИСЛИТЕЛЬНЫМ ПРОЦЕССОМ

Язык С предусматривает различные конструкции, позволяющие нам управлять потоком исполнения операторов в программе, т.е. выполнять ветвление, циклическое выполнение одного или нескольких операторов, передачу управления в нужное место кода программы.

Возможности управления расширяются тем, что можно составить блок из нескольких операторов и обращаться с ним как с одиночным оператором. Блок начинается с открывающей фигурной скобки { и заканчивается закрывающей скобкой }. Функция, собственно, является блоком.

Другими словами, операторы программы могут быть простыми и составными. Простой оператор – это оператор, не содержащий другие операторы. Каждый оператор в языке С должен заканчиваться точкой с запятой (;).

Специальным случаем простого оператора является пустой оператор – одинокая точка с запятой. Этот оператор используется там, где по синтаксису языка требуется оператор, а по смыслу программы никакие действия не выполняются.

Составной оператор (или блок) – любая совокупность простых операторов, заключенная в фигурные скобки {}. Составной оператор идентичен простому оператору и, в свою очередь, может входить в состав другого составного оператора.

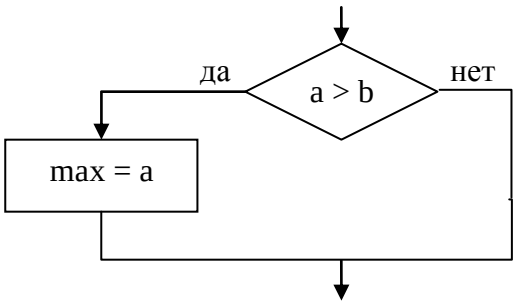
10.1. Операторы ветвления if и else

Операторы ветвления позволяют выполнить какой-то один фрагмент кода из нескольких, в зависимости от условия.

Оператор if осуществляет условное ветвление программы, проверяя истинность выражения. Он имеет следующий вид:

```
if (условное_выражение)
  оператор_исполняемый_если_выражение_истинно;
```

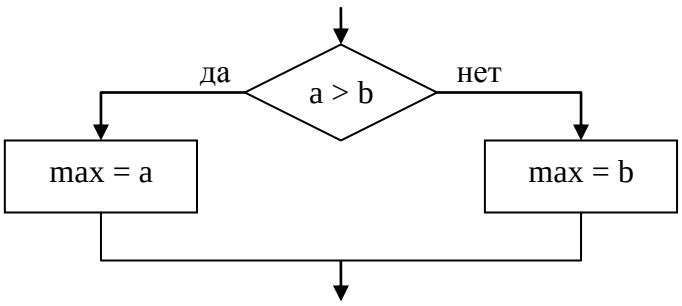
Внутри оператора if может быть как простой, так и составной оператор (т.е. несколько операторов). Если несколько операторов, то надо брать их в скобки {}, если один оператор – то можно брать, можно и не брать.




```
if (a > b)
  max = a;                  // простой оператор
if (a > 0 && b > 0) {
  x = a - b;                // составной оператор
  y = a + b;
}
```

При необходимости в комбинации с if можно использовать ключевое слово else, позволяющее выполнить альтернативный оператор, если выражение в условии ложно.

```
if (условное_выражение)
  оператор_исполняемый_если_выражение_истинно;
else
  оператор_исполняемый_если_выражение_ложно;
```




```
if (a > b)
  max = a;                  // простой оператор
else
```

```
  max = b;
if (a > 0 && b > 0) {
  x = a - b;              // составной оператор
  y = a + b;
}
else {
  x = a * b;
  y = a / b;
}

if (a > 0 && b > 0) {
  x = a - b;               // составной оператор
  y = a + b;
}
else
  y = x = a * b;           // простой оператор
```

Операторы if и else могут быть вложенными. Совместное использование обеих форм оператора if приводит к неоднозначности, называемой «проблемой висящего else».

Например, вложенный оператор if

```
if(e1) if( e2) оператор1; else оператор2;
```

может быть интерпретирован так:

```
if (e1)                   // верно
  if(e2)
    оператор1;
  else
    оператор2;
```

или так:

```
if(e1)                    // неверно
  if(e2) оператор1;
else
  оператор2;
```

Если такая конструкция является двусмысленной, компилятор ставит каждое else в соответствие ближайшему предшествующему if без else. Соответствие ищется в пределах блока, в которых заключено else. Следовательно, первая интерпретация является интерпретацией, принятой в языке С. Для ликвидации таких неоднозначностей и указания явного намерения программиста можно использовать скобки {}.

Задача. *Найти первое по порядку положительное число из трех чисел a, b, c.*

```
#include <stdio.h>
#include <bios.h>
void main() {
  int a, b, c, k = -1;
```

```
  printf("\nВведите три числа\n");
  scanf("%d %d %d", &a, &b, &c);
  if (a > 0)
    k = a;
  else {      // эти скобки можно не ставить, т.к. в else один оператор if…else
    if (b > 0)
      k = b;
    else
      k = c;
  }
  if (k > 0)
    printf("\nПервое положительное число = %d", k);
  else
    printf("\nПоложительных чисел нет");
  bioskey(0);
}
```

Обратить внимание! Отступы для демонстрации уровней вложенности операторов и где ставить скобки {}.

Нет такого как в Паскале – в С обязательно перед else должна стоять точка с запятой или скобка }.

Обратить внимание! Вынесение операторов до if и после else, если они повторяются в if и else.

10.2. Оператор switch

В некоторой точке вашей программы может оказаться несколько (более двух) возможных путей ветвления. Если необходимо выбрать один вариант из многих, то это можно сделать с помощью вложенных операторов if…else. Но это не очень удобно. В качестве альтернативы, для сложного условного ветвления язык С предоставляет конструкцию switch. Синтаксис ее следующий:

```
switch (выражение) {
  case константа_1:
    оператор_1;
    [break;]
  case константа_2:
    оператор_2;
    [break;]
  ...
  case константа_N:
    оператор_N;
    [break;]
  [default:
    оператор_N+1;]
}
```

где константа_i – константа или константное выражение, оператор_i – одни оператор или группа операторов. Скобки [] означают, что данная часть оператора может отсутствовать.

Результат вычисления выражения сравнивается с каждой из констант_i. Если находится совпадение, то управление передается оператору или операторам, связанным с данным case, т.е. идущим сразу после него. Исполнение оператора switch продолжается до конца оператора switch или пока не встретится оператор break, который передает управление из тела switch вовне.

Операторы, связанные с ключевым словом default, выполняются, если выражение не соответствует ни одному из константных выражений в case. Ключевое слово default необязательно должно располагаться в конце оператора switch.

Задача. Сколько дней в месяце (високосным будем считать только год, который делится на 4)?

```
void main() {
  int y, m, d;
  printf("Введите год и номер месяца\n");
  scanf("%d %d", &y, &m);
  switch(m) {
    case 2:
      d= 28 + !(y%4);
      break;
    case 4:
    case 6:
    case 9:
    case 11:
      d= 30;
      break;
    default:
      d= 31;
  }
  printf("Количество дней %d\n", d);
  bioskey(0);
}
```

10.3. Оператор цикла while

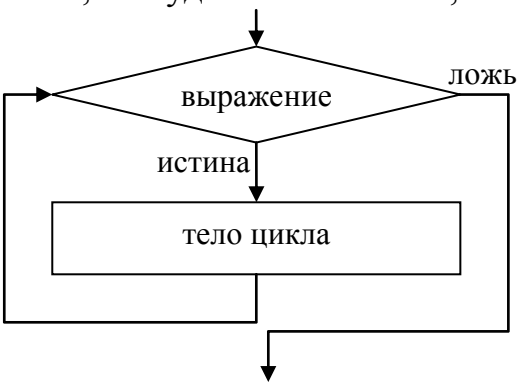
В языке С существуют три вида циклов: **while**, **for**, **do...while**.

Ключевое слово while позволяет выполнять тело цикла до тех пор, пока условие цикла не перестанет быть истинным. Синтаксис его следующий:

```
while (выражение)
  тело цикла;
```

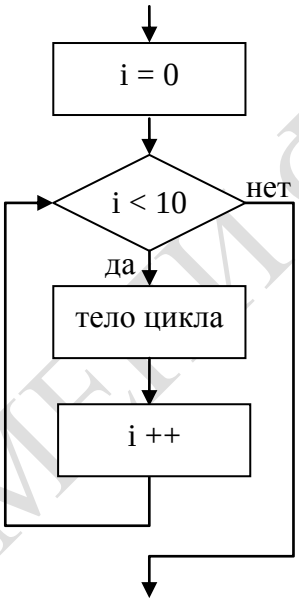
Если тело цикла состоит более чем из одного оператора (из составного оператора), то эти операторы надо заключать в фигурные скобки {}.

Тело цикла while, не будет выполняться, если выражение изначально ложно.



Обычно инициализирующее выражение расположено за пределами цикла (до цикла). Выражение, которое влияет на условие цикла, обычно находится в конце тела цикла.

```
int i = 0;          // инициализация счётчика
while (i < 10) {
  // тело цикла
  i++;              // операция инкремента
}
```



<u>Задача.</u> Подсчитать количество отрицательных и сумму положительных чисел, введенных с клавиатуры. Окончание ввода – число 0.

```
void main() {
  int a, k = 0, s = 0;
  printf("Введите последовательность чисел\n");
  scanf("%d", &a);
  while (a != 0) {
    if (a < 0) k++;
    else s += a;
    scanf("%d", &a);
  }
  printf("Количество отрицательных чисел = %d\n", k);
```

```
  printf("Сумма      положительных чисел = %d\n", s);
  bioskey(0);
}
```

Задача. Подсчитать количество чисел, введенных с клавиатуры. Окончание ввода – превышение суммы положительных чисел числа 100.

```
void main() {
  int a, k = 0, s = 0;
  printf("Введите последовательность чисел\n");
  scanf("%d", &a);
  if (a > 0) s += a;
  while (s <= 100) {
    k++;
    scanf("%d", &a);
    if (a > 0) s += a;
  }
  printf("Количество чисел = %d\n", k);
}
```

Задача. Подсчитать количество совпадающих пар чисел, введенных с клавиатуры. Окончание ввода – число, кратное 5.

```
void main() {
  int a, k = 0, b = 5;
  printf("Введите последовательность чисел\n");
  scanf("%d", &a);
  while (a%5 != 0) {
    if (a == b) k++;
    b = a;          // сохраняем предыдущее введенное число
    scanf("%d", &a);
  }
  printf("Количество пар совпадающих чисел = %d\n", k);
}
```

Задача. Ввести число и вывести все его цифры по одной в строке.

```
void main(){
  unsigned long x, y;
  scanf("%lu", &x);
  while(x){           // while(x!=0)
    y = x % 10;       // очередная цифра числа
    printf("y = %lu\n", y);
    x /= 10;          // убираем последнюю цифру числа
  }
}
```

Задача. Сформировать число по его цифрам. Окончание ввода – цифра 9.

```
void main(){
```

```
  unsigned long x, y = 0;
  scanf("%lu", &x);
  while(x != 9){
    y *= 10;   // y = y*10 + x;
    y += x;    //
    scanf("%lu", &x);
  }
  printf("y = %lu\n", y);
}
```

Задача. *Записать число в обратном порядке цифр (125 => 521).*

```
void main(){
  unsigned long x, y = 0, c;
  scanf("%lu", &x);
  while(x){
    c = x % 10;
    y = y*10 + c;
    x /= 10;
  }
  printf("y = %lu\n", y);
}
```

Задача. *Удалить старший разряд числа (725 => 25).*

```
void main(){
  unsigned long x, y = 0, p = 1;
  scanf("%lu", &x);
  while(x > 9){
    y += (x%10 * p);
    x /= 10;
    p *= 10;
  }
  printf("y = %lu\n", y);
}
```

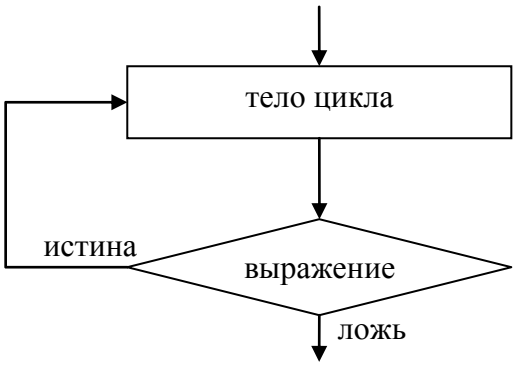
Второй вариант:

```
void main(){
  unsigned long x, y, xx, k = 1;
  scanf("%lu", &x);
  xx = x;
  while(x > 9){
    k *= 10;
    x /= 10;
  }
  y = xx % k;
  printf("y = %lu\n", y);
}
```

10.4. Оператор цикла do…while

В цикле do…while оценка условия производится после исполнения тела цикла. Это означает, что тело цикла будет обязательно исполняться хотя бы один раз. Синтаксис оператора имеет следующий вид:

```
do
  тело цикла;
while (выражение);
```

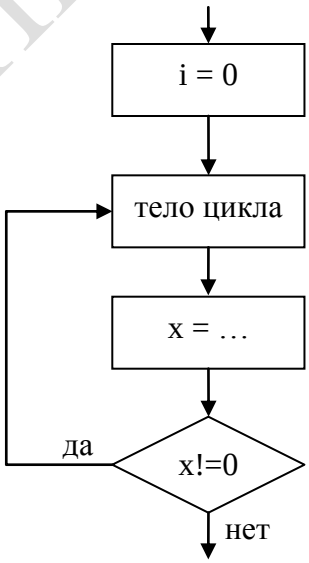


Если тело цикла состоит более чем из одного оператора, то эти операторы надо заключать в фигурные скобки {}.

Тело цикла do…while будет выполняться не менее 1 раза, даже если выражение изначально ложно.

В таком виде тело цикла выполнится один раз точно, а дальнейшее выполнение цикла будет зависеть от полученного значения x:

```
int x;                  // описание x – значение случайное
do {
  // тело цикла
  x = ...;              // вычисление x для условия
}
while (x != 0);
```



Задача. Ввод положительного числа с проверкой.

```
void main(){
  int x;
  do {
    printf("Введите положительное число\n");
    scanf("%d",&x);
  }
  while(x <= 0);
  printf("x = %d\n", x);
}
```

10.5. Оператор цикла for

Наиболее сложная форма оператора цикла – это оператор for.

Синтаксис оператора следующий:

```
for (выражение_1; выражение_2; выражение_3)
  тело цикла;
```

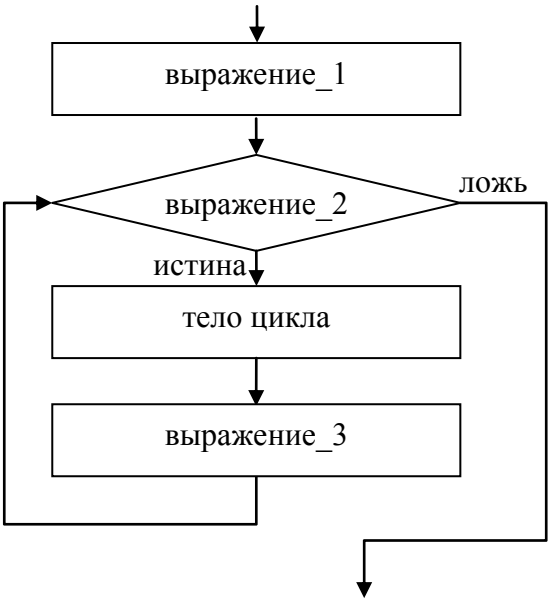
Цикл for похож на цикл while в следующей интерпретации:

```
выражение_1;
while (выражение_2) {
  тело цикла;
  выражение_3;
}
```

Сейчас самое важное – понять, как работает цикл for:

1) Выполнение выражения_1 (инициализация)
2) Выполнение выражения_2 (проверка условия)
3) Если выражение_2 ложно, то выход из цикла
4) Выполнение тела цикла
5) Выполнение выражения_3 (приращение)
6) Переход на выполнение выражения_2, т.е. к пункту 2

Эти шаги повторяются до тех пор, пока выражение_2 не станет ложным, и программа продолжит работу, начиная со следующего оператора, стоящего после тела цикла.



Если тело цикла состоит более чем из одного оператора, то эти операторы надо заключать в фигурные скобки {}.

Любое выражение их трех в заголовке цикла for может быть последовательностью простых операторов, разделяемых оператором запятая.

Все три выражения в скобках () цикла for являются необязательными. Тем не менее, в скобках () цикла for обязательно должны быть все точки с запятой (две), даже если какое-то выражение опущено.

Выражение_1, если есть, всегда будет выполняться. Вычисление выражения_3 не производится, если выражение_2 ложно с самого начала.

```
void main(){
  int x, s;
  for (s = 0, x = 1; x < 11; ) {
    s += x;
    x++;
  }
  printf("s = %d\n", s);
}
```

Второй вариант:

```
void main(){
  int x = 1, s = 0;
  for (; x < 11; x++)
    s += x;
  printf("s = %d\n", s);
}
```

Отличие цикла for в С от цикла for в Паскале: 1) индексную переменную можно изменять внутри цикла; 2) условие выполнения цикла не фиксируется изначально, а вычисляется на каждом проходе цикла (поэтому лучше не использовать в условии вычисляемые выражения, в том числе с вызовами функций, если они дают на каждом проходе цикла одно и то же значение).

Различные операторы циклов могут выражаться друг через друга.

Задача. Найти сумму чисел от 1 до 10.

```
void main(){
  int x, s = 0;
  for (x = 1; x < 11; x++)
    s += x;
  printf("s = %d\n", s);
}
```

Второй вариант:

```
void main(){
  int x = 1, s = 0;
  while (x < 11) {
    s += x;
    x++;
  }
  printf("s = %d\n", s);
}
```

Третий вариант:

```
void main(){
  int x = 1, s = 0;
  do {
    s += x;
    x++;
  }
  while (x < 11);
  printf("s = %d\n", s);
}
```

10.6. Бесконечные циклы

Одна из самых неприятных особенностей операторов цикла заключается в возможности образования бесконечного цикла (в зацикливании программы) там, где программист этого не ожидает. Причина зацикливания – условие выполнения цикла никогда не станет ложным из-за ошибок в программе.

```
void main(){
  int x = 1, s = 0;
  while (x < 11);  // !!!!!!!!!
  {
    s += x;
    x++;
  }
  printf("s = %d\n", s);
}
```

В этом примере причина зацикливания – незаметная точка с запятой после while. Компилятор считает, что в цикле повторяется не составной оператор в {}, а

пустой оператор. А так как в цикле x никогда не изменится, то и условие никогда не станет ложным. Но это не значит, что не может быть в цикле пустого оператора. Далее приведен пример правильного цикла с пустым оператором цикла.

```
void main(){
  int x, s;
  for (s = 0, x = 1; x < 11; s+= x, x++);
  printf("s = %d\n", s);
}
```

Если все же вам в программе потребуется бесконечный цикл (а это иногда бывает нужно сделать), вы можете написать цикл, который никогда не кончается, опустив все три выражения в операторе for.

```
for (;;)
  printf("Бесконечный цикл\n");
```

Можно также написать бесконечный цикл, используя в операторе while в качестве условия выражение, которое всегда истинно. Конечно, программу, выполнение которой не завершается, вроде как нельзя назвать корректной. Но, как вы увидите в дальнейшем, существуют способы прекратить исполнение и бесконечных циклов.

```
while (1)
  printf("Бесконечный цикл\n");
```

10.7. Другие управляющие средства языка С

Бывают ситуации, когда необходимо прервать выполнение блока операторов независимо от каких-либо условий. Язык С предусматривает для этих целей четыре оператора: break, continue и return. Это так называемые операторы безусловной передачи управления.

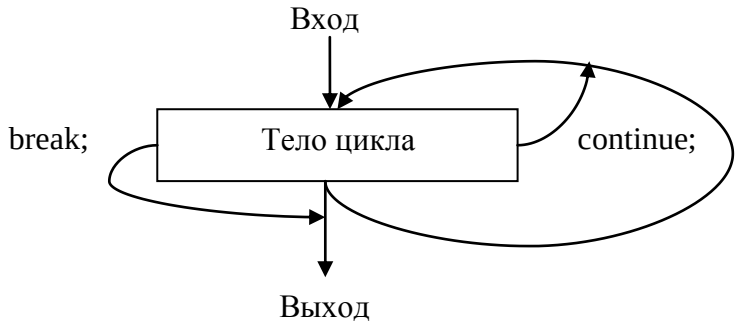
Оператор break прекращает выполнение оператора цикла while, do…while, for и switch, в котором он непосредственно находится. Для выхода из группы вложенных циклов, надо использовать несколько операторов break.

```
int x;
while (1) {
  printf("Введите положительное число\n");
  scanf("%d", &x);
  if (x > 0) break;
}
printf("x = %d\n", x);
```

Оператор continue можно использовать только внутри тела цикла. Данный оператор возвращает управление к началу цикла, пропуская оставшуюся его часть. В результате его выполнения управление переносится:

1) для цикла for на секцию выражения_3;

2) для циклов while и do…while на вычисление выражения и определения необходимости завершения цикла.



Задача. *Найти сумму четных чисел из диапазона от 1 до 10.*

```
int x = 2, s = 0;
while (x < 11) {
  s += x;
  x += 2;
}
printf("s = %d\n", s);
```

==========

```
int x = 1, s = 0;
while (x < 11) {
  if (x%2 == 1) {
    x++;
    continue;
  }
  s += x;
  x++;
}
printf("s = %d\n", s);
```

==========

```
int x, s = 0;
for (x = 2; x < 11; x+=2)
  s += x;
printf("s = %d\n", s);
```

==========

```
int x, s = 0;
for (x = 1; x < 11; x++) {
  if (x%2 == 1) continue;
  s += x;
}
printf("s = %d\n", s);
```

Оператор return переносит управление из текущей функции в точку ее вызова, т.е. прерывает выполнение функции. Его мы рассмотрим позднее.

Оператор goto осуществляет безусловную передачу управления на метку в пределах текущей функции. Так использовать этот оператор уж очень не рекомендуется из-за запутанности и сложности понимания программы с ним, рассматривать его мы не будем, т.к. любую программе с goto можно написать без goto.

<u>Задача.</u> Вычислить значение функции на интервале [a;b] в n равноудаленных точках. Обеспечить контроль введенных и полученных данных.

$$f = \begin{cases} \cos(x) + \dfrac{7 - x^2}{|x|}, & x < 2, \\ \ln(x - u) - \dfrac{\sqrt{x}}{u}, & x \geq 2. \end{cases}$$

```
#include <stdio.h>
#include <math.h>
void main(){
  int n, err;
  float a, b, u, x, f, h;
  printf("Введите a, b, u, n\n");
  scanf("%f%f%f%d", &a, &b, &u, &n);
  if(n < 2) n = 10;
  if(a == b) { a -= 0.5; b += 0.5; }
  else
    if(a > b) { x = a; a = b; b = x; }
  h = (b-a) / (n-1);
  for(x = a; x < b+0.5*h; x += h){
    err=0;
    if (x < 2) {
      if (x == 0)
        err = 1;
      else
        f = cos(M_PI*x)+(7-pow(x,2))/fabs(x);  // M_PI  x*x
    }
    else {
      if (u == 0 || x-u <= 0)
        err = 1;
      else
        f = log(x-u)-pow(x,0.5)/u;   // sqrt(x)
    }
    printf("|%10.4f  |", x);
    if(err)
       printf("  не определена!  |\n");
    else
      printf("%16.4f  |\n", f);
  }
}
```

10.8. Стандартные математические функции

В любых арифметических выражениях можно использовать стандартные математические функции, которые можно применять к любым числовым операндам. При использовании этих функций в программу необходимо включить файл <math.h>. При этом будут определены, например, следующие функции:

sin(x) – синус (аргумент в радианах);
cos(x) – косинус (аргумент в радианах);
tan(x) – тангенс (аргумент в радианах);
exp(x) – экспонента e^x;
log(x) – натуральный логарифм ln(x);
log10(x) – десятичный логарифм $\log_{10}(x)$;
sqrt(x) – квадратный корень;
pow10(x) – возведение числа 10 в степень x, т.е. 10^x;
pow(x,y) – возведение в степень x^y;
fabs(x) – абсолютная величина для double;
abs(x) – абсолютная величина для int.

11. ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЙ И ПОБОЧНЫЕ ЭФФЕКТЫ

11.1. Преобразования типов при вычислении выражений

При выполнении операций происходят неявные (автоматические) преобразования типов в следующих случаях:

- при выполнении операций, если операнды разных типов;
- при выполнении операций присваивания, если значение одного типа присваивается переменной другого типа;
- при передаче аргументов функции.

При выполнении операций автоматическое преобразование типов выполняется, чтобы привести операнды выражений к общему типу так, чтобы не было потери точности. Затем осуществляется сама операция. В таком случае всегда происходит расширение коротких величин до размера больших величин (другими словами, операнды приводятся к «старшему» типу, т.е. более длинному типу). Типы по убыванию старшинства: long double, double, float, unsigned long, long, unsigned int, int, short, char.

Правила неявного преобразования типов в выражения:

1) char и short преобразуется в int, float – в double (в С все действия с вещественными числами производятся с двойной точностью);

2) если один из операндов double, то другие тоже преобразуются к double, и результат тоже double;

3) если один из операндов long, то другие тоже преобразуются к long, и результат тоже long;

4) если один из операндов unsigned, то другие тоже преобразуются к unsigned, и результат тоже unsigned;

5) если все операнды типа int, то результат будет тоже int.

В операциях присваивания тип значения, которое присваивается, преобразуется к типу переменной, получающей это значение. Допускается преобразования целых и плавающих типов, даже если такое преобразование ведет к потере информации. Т.е. в этом случае возможно преобразование старшего типа к младшему.

```
int x, y = 7;
float z = 5.7;
x = z;       // x = 5
z = y;       // z = 7.0
```

Преобразования при вызове функции: если задан прототип функции, и он включает объявление типов аргументов, то над аргументами в вызове функции выполняются обычные арифметические преобразования. Эти преобразования выполняются независимо для каждого аргумента. Например, тип параметра float, а задали при вызове функции в качестве параметра целое 7. Тогда перед выполнением функции целое число 7 преобразуется к вещественному числу 7.0.

Обратить внимание! Арифметические операции над операндами выполняются корректно при условии, что результат не выходит за пределы разрешенного диапазона. Преобразования, выполняемые при операциях, не обеспечивают обработку ситуаций переполнения. При этом сообщение об ошибке не выдается.

Например, если к максимальному int числу (2 байта) прибавить 1, то вместо положительного числа +32768 в компьютере окажется отрицательное число –32768. И никакого предупреждения о нарушении допустимого интервала система не выдаст. Считается, что программист сам должен позаботиться о соответствующих проверках.

```
int i=30000, j=30000, k;  // max_int = 32767
long y =30000;
long x;
k = i + j;         // k = -5536  =>  60000 - 65536 (2^16)
x = i + j;         // x = -5536
x = i + y;         // x = 60000
x = (long)i + j;   // x = 60000
```

Почему на работает эта программа (зацикливается)?

```
#include <stdio.h>
main(){
  unsigned char i;
  for(i=0;i<=255;i++)
    printf("%d\n",i);
}
```

По замыслу эта программа должна вывести на экран все числа от нуля до 255 и затем прекратить работу. На самом же деле она будет работать вечно, выводя на экран все числа от 0 до 255 вновь и вновь. Весь фокус в том, что переменная i объявлена как unsigned char, а диапазон ее значений – от нуля до 255. Казалось бы, условие i<=255 показывает, что i меняется в допустимых пределах. Но давайте посмотрим, что будет с циклом, когда i достигнет 255. Условие i <= 255 в этом случае выполнится, printf() покажет число 255, а дальше i увеличится на единицу и снова будет проверено условие i < =255. Чему будет равно i в этот момент, мы уже знаем: 255+1 означает для переменной unsigned char ноль! Условие i <= 255 будет выполнено, и цикл будет прокручен снова 256 раз, затем i снова станет равна нулю и так до бесконечности.

Очень важно понимать, что компилятор *не может и не хочет обнаруживать такие ошибки*. Язык С не вмешивается в замысел программиста, считая, что тот хорошо знает, что делает. С создавался для программистов, которым не нужно мешать надоедливыми проверками и предупреждениями.

Любые операнды типа char и short преобразуются к типу int, а любые операнды unsigned char или unsigned short преобразуются к типу unsigned int.

```
unsigned char a=150, b=150, c; // max_uchar = 255
int d;
c = a + b;  // c = 44, но потеря информации произойдет при приведении
                типов при присваивании, а не при сложении => 300 – 256 (2^8)
d = a + b;  // d = 300

int i = 30000, j = 30000;
long y = 30000, x;
x = i + j + y;     // x = 24464, т.к. при i+j уже информация потеряется
x = i + (j + y);   // x = 90000
```

Про такие эффекты надо помнить. Но это не единственные подводные камни при вычислении выражений.

11.2. Побочные эффекты при вычислении выражений

1) *Операции присваивания, инкремента и декремента в сложных выражениях могут вызывать побочные эффекты*, так как они не только возвращают значение, но еще и изменяют операнд. Эти побочные эффекты могут вызвать трудности из-за того, что получение значения и обновление переменной могут проходить не тогда, когда ожидается. Кроме этого, порядок вычисления операндов некоторых операций зависит от реализации (от компилятора).

Другими словами, необходимо с осторожностью использовать выражения, при вычислении которых возможны побочные результаты, так как результаты вычисления таких выражений часто проявляются не сразу и, кроме того, зависят от используемого компилятора.

```
j = 3;
i = (k = j + 1) + (j = 5);
```

Значение переменной i будет равно 9 или 11 в зависимости от того, какое выражение в скобках будет вычислено первым. Таким образом, с использованием разных компиляторов можно получить различные результаты. Кроме побочных эффектов, использование вложенных операторов присваивания усложняет чтение и восприятие программного кода.

При применении операций инкремента или декремента к переменной эта переменная не должна появляться в выражении более одного раза, так как итог и в этом случае будет зависеть от компилятора. Не следует писать код, который полагается на порядок обработки или особенности компилятора:

```
m[i++] = m[i++] = 0;
```

Смысл выражения – записать 0 в следующие две позиции массива m. Однако в зависимости от того, когда будет обновляться i, позиция в m может быть пропущена, и в результате переменная i будет увеличена только на 1.

В BC, например, получим: i = 0, m = {1, 2, 3, 4, 5} => i = 2, m = {0, 2, 3, 4, 5}, а не ожидаемые: i = 2, m = {0, 0, 3, 4, 5}.

Надо разбить выражение на 2 выражения: `m[i++] = 0; m[i++] = 0;`

Даже если в выражении содержится только один инкремент, результат может быть неоднозначным:

```
m[i++] = i;
```

Если изначально i = 1, то элемент массива может принять значение 1 или 2.

2) *Операнды логических выражений вычисляются слева направо*. Если значения первого операнда достаточно, чтобы определить результат операции, то второй операнд не вычисляется. Это опять же зависит от компилятора. Например, в BC есть ускоренное вычисление логических выражений.

```
int a = 2, b = 3, c = 4, i = 0, j = 0;
if (a < i++ && b < j++)
  c = 10;
```

Т.к. a=2 уже не меньше i=0, то i++ выполнится, а второе выражение даже не будет проверяться и вычисляться и после выполнения оператора: j=0, а не j=1.

3) *Побочные эффекты могут возникать и при вызове функции*, если он содержит прямое или косвенное присваивание (через указатель). Это связано с тем, что аргументы функции могут вычисляться в любом порядке. Например, побочный эффект имеет место в следующем вызове функции:

```
prog(a, a = k * 2); // в зависимости от того, какой аргумент вычисляется
                    первым, в функцию могут быть переданы различные значения
```

Ввод-вывод – еще один потенциальный источник возникновения закулисных действий.

В примере осуществляется попытка прочитать два взаимосвязанных числа из стандартного ввода:

```
scanf (“%d %d”, &n, &m[n]);
```

Выражение неверно, поскольку одна его часть изменяет n, а другая использует ее. Значение m[n] будет правильным только в том случае, если новое значение n будет таким же, как и старое. Проблема здесь не в порядке вычисления аргументов, а в том, что все аргументы scanf() вычисляются до того, как эта функция вызывается на выполнение, так что &m[n] всегда будет вычисляться с использованием старого значения n.

Следующий пример показывает, что аргументы при вызове функции вычисляются не в том порядке, в каком ожидалось:

```
int a = 2;
printf("%d %d", a+=3, a+=5); // a = 10   на экране 10 и 7 (а не 5 и 10)
```

или

```
printf("%d %d", a+=3, a);     // a = 5    на экране 5 и 2 (а не 5 и 5)
```

Чтобы избежать недоразумений при выполнении побочных эффектов, необходимо придерживаться следующих правил:

1. Не использовать операции присваивания переменной в выражении, если эта переменная используется в выражении более одного раза.

2. Не применять операции увеличения или уменьшения к переменной, которая входит в выражение более одного раза.

3. В логических выражениях не проводить никаких дополнительных вычислений.

4. Не использовать операции присваивания переменной в вызове функции, если эта переменная участвует в формировании других аргументов функции.

5. Не применять операции увеличения или уменьшения к переменной, присутствующей в более чем одном аргументе функции.

6. Использовать естественную форму выражений. Записывать выражения в том виде, в котором вы произносили бы их вслух.

7. Разбивать сложные выражения. Язык С имеет богатый и разнообразный синтаксис, поэтому многие увлекаются втискиванием всего подряд в одну конструкцию.

```
x += (xp = (2*k < (n-m) ? c[k+1] : d[k --]) );
```

Лучше написать так:

```
xp = 2*k < (n-m) ? c[k+1] : d[k --];
x += xp;
```

или даже так:

```
if (2*k < n-m)
  xp = c[k+1];
else
```

```
  xp = d[k--];
x += xp;
```

8. И как вывод: необходимо быть проще. Это убережет и от ошибок, и упростит восприятие вашего программного кода.

12. МАССИВЫ

Очень часто в программе требуется сохранять и обрабатывать некоторое множество значений одного и того же типа. Как вы должны уже знать, в таких случаях используют массивы.

Массив – это совокупность элементов данных одного и того же типа, объединенных общим именем и расположенных в непрерывной области памяти вплотную друг за другом так, что к каждому элементу массива можно получить доступ, зная его порядковый номер или индекс.

12.1. Описание массива

Описание массива производится с помощью обычного оператора описания, при этом за именем массива в квадратных скобках должна быть записана целая положительная константа или константное выражение, равное размеру этого массива, то есть максимально возможному числу элементов.

```
#define N_MAX 100
const int N =10;

int a[5], b[2*40], K = 10;
// int mas[K];  - ошибка
double c[N];
char d[N_MAX];
```

В таком виде, все элементы массива ещё не инициализированы, т.е. содержат случайные значения.

Имя массива с квадратными скобками, в которых записано индексное выражение целого типа, обозначает значение соответствующего элемента массива. В языке С нумерация элементов массива начинается с нуля, то есть в массиве a из пяти элементов есть следующие элементы: a[0], a[1], a[2], a[3], a[4].

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

индексы массива

При работе с индексированными переменными необходимо внимательно следить за тем, чтобы индексы не вышли за пределы допустимого диапазона, определяемого при описании массива (от 0 до n-1). Дело в том, что компилятор С не проверяет факт выхода индексов за границы массива, а при ошибочном занесении данных за пределы массива может запортиться нужная информация и, скорее всего, программа зависнет.

12.2. Инициализация массива

Инициализация – это присвоение значений вместе с описанием данных. Ранее мы уже рассматривали инициализацию простых переменных: int a = 5;

Для инициализации массива за его именем располагают знак присваивания и список инициализации, который представляет собой заключенные в фигурные скобки и разделенные запятыми инициализирующие значения.

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

Но нельзя написать по аналогии с переменными:

```
int x;  x=5;
int a[5]; // a = {1, 2, 3, 4, 5}; - ошибка
```

Констант в списке инициализации должно быть не больше, чем объявленный размер массива. Если их меньше, то элементы, для которых нет констант, обнуляются.

```
int a[5] = {1, 2, 3};  // a[3] = a[4] = 0
```

Для инициализируемого массива допускается вообще не указывать размер массива. В этом случае размер массива определяется компилятором по количеству констант.

```
int a[] = {1, 2, 3, 4, 5}; //
```
компилятор выделит 10 байт для хранения массива из 5 двухбайтовых целых чисел

Как в программе вычислить реальную длину массива, задаваемого в виде: *TYPE* mas[] = { ..... }; , т.е. без явного указания размера:

Количество элементов в таком массиве можно вычислить так:

```
N_MAS = sizeof(mas) / sizeof(mas[0]);
```

или

```
N_MAS = sizeof(mas) / sizeof(TYPE);
```

Оба способа выдадут число, равное N (*sizeof(mas)* выдает размер всего массива в байтах, а *sizeof(mas[0])* или *sizeof*(*TYPE*) выдают размер одного элемента).

12.3. Ввод-вывод массива

Язык С не имеет встроенных средств для ввода-вывода массива целиком, поэтому массив вводят и выводят поэлементно с помощью циклов. Также нельзя один массив напрямую присвоить другому (присвоение значений тоже надо выполнять по одному элементу).

Задача. Найти сумму элементов массива.

```
void main() {
  int a[100], n, i, n_max;
  int s = 0;
  n_max = 100;        // n_max = sizeof(a) / sizeof(int);
  do {
    printf("Введите количество элементов массива n = ");
```

```
    scanf("%d", &n);
  }
  while (n < 0 || n > n_max);
  printf("Введите элементы массива\n");
  for (i = 0; i < n; i++) {
    printf("a[%d] = ", i);
    scanf("%d", &a[i]);
  }
  printf("Вы ввели массив\n");
  for (i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");
  for (i = 0; i < n; i++)
    s += a[i];
  printf("Сумма элементов массива = %d\n", s);
}
```

Задача. Отсортировать массив по возрастанию.

```
void main(void) {
  int a[10], n;
  int i, j, b;
  scanf("%d", &n);
  for (i=0; i<n; i++)
  scanf("%d", &a[i]);
  for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
      if (a[i] > a[j]) {
        b = a[i]; a[i] = a[j]; a[j] = b;
      }
  for (i=0; i<n; i++)
    printf("%d ", a[i]);
  printf("\n");
}
```

Задача. Сформировать вектор B из четных элементов вектора A.

```
void main() {
  int a[100], b[100], na, nb, i, j;
  printf("Введите количество элементов в векторе A: ");
  scanf("%d", &na);
  printf("Введите элементы вектора A\n");
  for (i = 0; i < na; i++)
    scanf("%d", &a[i]);
  j = 0;    // индекс очередного формируемого элемента = 0
  for (i = 0; i < na; i++)
    if (!(a[i]&1)) {
      b[j] = a[i];  // формирование вектора B
```

```
      j++;  // индекс очередного формируемого элемента
  }
  nb = j;  // количество элементов в сформированном B
  printf("Сформирован вектор B\n");
  for (j = 0; j < nb; j++)
    printf("%d ", b[j]);
}
```

12.4. Двумерные массивы (массивы массивов)

Элементом массива может быть в свою очередь тоже массив. Таким образом, мы приходим к понятию двумерного массива или матрицы. Описание двумерного массива строится из описания одномерного путем добавления второй размерности:

```
int a[4][3];
```

Анализ подобного описания необходимо проводить в направлении выполнения операций [], то есть слева направо. Таким образом, переменная a является массивом из четырех элементов, что следует из первой части описания a[4]. Каждый элемент a[i] этого массива в свою очередь является массивом из трех элементов типа int, что следует из второй части описания.

Имя двумерного массива с двумя индексными выражениями в квадратных скобках за ним обозначает соответствующий элемент двумерного массива и имеет тот же тип. Например, a[2][1] является величиной типа int, а именно ячейкой, в которой находится число 8, и может использоваться везде, где допускается использование величины типа int.

Для наглядности двумерный массив можно представить в виде таблицы с числом строк, равным первому размеру массива, и числом столбцов, равным второму размеру массива*:*

Массив a	**Столбец 0**	**Столбец 1**	**Столбец 2**
Строка 0	1	2	3
Строка 1	4	5	6
Строка 2	7	8	9
Строка 3	10	11	12

Но память компьютера, одномерна, в ней нет ничего кроме идущих подряд ячеек памяти, поэтому компилятор вынужден будет «вытянуть» двухмерный массив в линейку по *строкам*: сначала в памяти разместится первая строка, потом – вторая, за ней третья и т.д.

В соответствии с интерпретацией описания двумерного массива (слева-направо) элементы двумерного массива располагаются в памяти компьютера по строкам, одна строка за другой. Для массива int a[2][3] (2 строки, 3 столбца) расположение в памяти будет таким:

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	
0 0	0 1	0 2	1 0	1 1	1 2	индексы массива

Инициализация двумерного массива также проводится по строкам.

```
int a[4][3] = { { 18, 21,  5 },   // внутренние {} можно опустить
                {  6,  7, 11 },
                { 30, 52, 34 },
                { 24,  4, 67 }
              };
int a[][3] = {  { 18, 21,  5 },
                {  6,  7, 11 },
                { 30, 52, 34 },
                { 24,  4, 67 }
             };

int a[4][3] = { { 18 },         // пропущенные элементы будут = 0
                {  6,  7 },     // внутренние {} нельзя опустить
                { 30, 52 },
                { 24,  4, 67 }
              };
```

Во втором случае первый размер массива будет определен компилятором. Следует отметить, что второй размер массива должен быть всегда указан. Это необходимо для того, чтобы сообщить компилятору размер строки массива, без которого он не может правильно разместить двумерный массив в памяти компьютера.

Ввод двумерного массива осуществляется поэлементно с помощью двух вложенных циклов. Следующий фрагмент программы предназначен для ввода по строкам двумерного массива размером n строк на m столбцов.

```
int a[5][10], n = 5, m = 10;
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++) {
    printf("a[%d][%d] = ", i, j);
    scanf ("%d", &a[i][j]);
  }
```

Вывод такого же двумерного массива также осуществляется с помощью двух вложенных циклов (после вывода очередной строки массива осуществляется переход на следующую строку).

```
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++)
    printf ("%5d ", a[i][j]);
  printf("\n");
}
```

<u>Задача.</u> Найти сумму элементов массива A[n][m]. В заданной строке найти максимальный элемент.

```
void main() {
  int a[10][20], n, m, i, j, k, s = 0, max;
  printf("Введите размерность массива n и m\n");
```

```
  scanf("%d %d", &n, &m);
  printf("Введите элементы массива\n");
  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      scanf("%d", &a[i][j]);
  printf("Вы ввели массив\n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
      printf ("%5d ", a[i][j]);
    printf("\n");
  }
  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      s += a[i][j];
  printf("Сумма элементов массива = %d\n", s);
  printf("Введите номер строки k = ");
  scanf("%d", &k);
  max = a[k][0];
  for (j = 1; j < m; j++)
    if (a[k][j] > max)
      max = a[k][j];
  printf("Максимальный элемент в %d строке = %d\n", k, max);
}
```

Обратить внимание! В двумерном массиве строка, столбец и диагонали являются по своей сути одномерными массивами. Поэтому при обработке строки, столбца или диагоналей достаточно использовать один цикл, а не два вложенных.

В языке С допускается использовать не только двумерные, но и трехмерные, четырехмерные и т. д. массивы. Их использование ничем принципиально не отличается от использования двумерных массивов, однако на практике они применяются значительно реже.

13. УКАЗАТЕЛИ

Самая эффективная программа – программа, написанная на машинном языке (что не делается из-за сложности) или ассемблере (в ОС все, что критично по скорости выполнения, пишут на нем). И машинная программа, и ассемблер хороши тем, что работают непосредственно с адресами и участками памяти.

Память компьютера представляет собой массив байт. Когда мы описываем некоторую переменную или массив, в памяти выделяется непрерывная область для хранения этой переменной. Все байты памяти компьютера пронумерованы. Номер байта, с которого начинается в памяти наша переменная, называется адресом этой переменной (смещение этой ячейки от начала памяти).

Данное может занимать несколько подряд идущих байт. Размер в байтах участка памяти, требуемого для хранения значения типа `TYPE`, можно узнать при

помощи операции `sizeof(TYPE)`, а размер переменной `var` – при помощи `sizeof(var)`.

Язык С хорош тем, что он аккумулирует в себе свойства языков низкого и высокого уровня, т.е. позволяет писать низкоуровневые задачи (т.е. работать прямо с областями памяти) на языке высокого уровня. Кроме этого:

1) Во-первых, код на языке С можно легко писать на низком уровне абстракции, почти как на ассемблере (только писать и отлаживать такой код намного проще, чем на ассемблере). Поэтому С называют «универсальным ассемблером» или «ассемблером высокого уровня». В то же время, С часто называют языком среднего уровня или даже низкого уровня, учитывая то, как близко он работает к реальным устройствам.

2) Во-вторых, язык С позволяет программистам довольно точно представлять, как выполняются их программы. Благодаря этому программы, написанные на С, эффективнее написанных на многих других языках. Как правило, лишь оптимизированный вручную код на ассемблере может работать ещё быстрее, потому что он даёт полный контроль над машиной.

Язык С имеет средства работы непосредственно с областями оперативной памяти компьютера, задаваемыми их адресами. Т.е. имеется два способа доступа к переменной: ссылка на переменную по имени и непосредственный доступ к памяти компьютера через использование указателей. Именно указатели дают возможность писать низкоуровневые и эффективные программы. А еще указатели дают возможность писать не просто эффективные, но и компактные программы. Поэтому указатели в языке С – одна из наиболее привлекательных для профессиональных программистов особенность языка.

Но существует и противоположное мнение: указатели вместе с с операторами goto позволяют писать программы, которые невозможно понять, а также программы с трудно находимыми ошибками программирования. Это, безусловно, справедливо, если указатели используются программистом без должной квалификации. Однако, при должном уровне знаний, использование указателей помогает достичь ясности и простоты.

Задача. *Переслать данные из одного массива в другой.*

```
void main() {
  int xx[3][3] = { {1,2,3},
                   {4,5,6},
                   {7,8,9}
                 };
  int yy[3][3] = { {5,5,5},
                   {7,7,7},
                   {9,9,9}
                 };
  int i, j;
  for (i=0; i<3; i++)
    for (j=0; j<3; i++)
      xx[i][j] = yy[i][j];
```

```
// с помощью указателей
  memcpy(xx, yy, sizeof(yy));
//  memcpy(xx,yy,10);    // переслать первые 5 элементов массива
```

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

Указатель является переменной, которая содержит адрес в памяти (переменная, содержащая адрес в памяти другой переменной). Указатель-константа – это значение адреса оперативной памяти.

Отличие указателей от машинных адресов состоит в том, что указатель может содержать адреса данных только определенного типа. Другими словами, в языке С указатели строго типизированы, т. е. различают указатели (адреса) символьных, целых, вещественных величин, а также типов данных, создаваемых программистом. Для описания указателя на какой-либо тип данных перед именем переменной ставится *. Объявление указателя `int *ptr` говорит о том, что выражение `*ptr` имеет тип `int`.

```
int *a, *b, c, d;
double *bc;
char *s;
```

Никогда не следует писать знак * слитно с типом данных: `int* a,b;`. В этой строке создается впечатление о том, что описаны два указателя на тип int, в то время как на самом деле описан один указатель на `int`, а именно `a`, и одна переменная `b` типа `int`.

Описание переменных заставляет компилятор выделять память для хранения этих переменных. Описание указателя выделяет память лишь для хранения адреса. В этом смысле указатели на `int` и на `double` будут занимать в компьютере одинаковое количество байт памяти, зависящее от модели памяти, на которую настроен компилятор. Например, в Small-модели длина указателя равна двум байтам, а в Large-модели – четырем. Конкретную длину указателя можно определить с помощью операции `sizeof`.

При описании указателей в качестве имени типа данных можно использовать ключевое слово `void`: `void *vd;`. При таком описании с указателем не связывается никакой тип данных, т. е. получаем указатель на данные произвольного типа. Такой указатель обычно называют пустым указателем. Указатель на `void` применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Ключевое слово `void` говорит об отсутствии данных о размере объекта в памяти. Но компилятору для корректной интерпретации ссылки на память через указа-

тели нужна информация о числе байтов, участвующих в операции. Поэтому, во всех случаях использования указателя, описанного как `void *`, необходимо выполнить операцию явного приведения типа указателя.

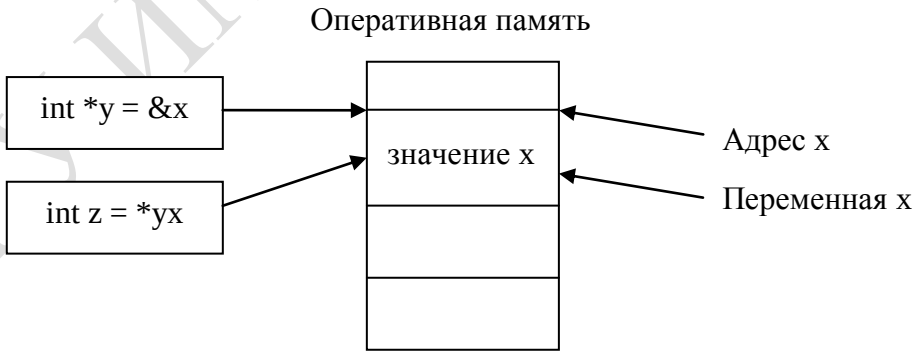
Для поддержки адресной арифметики в языке С имеются две специальные операции – операция взятия адреса & и операция получения значения по заданному адресу * (операция разименования, разадресации).

Унарная операция & выдает адрес объекта. Оператор `px = &x;` присваивает переменной `px` (указателю) адрес первой ячейки памяти, начиная с которой хранится значение переменной `x` (говорят, что теперь `px` *указывает на* `x`, т.е px содержит адрес `x`). Различия между двумя формами записи, `px` и `&x`, заключается в том, что `px` – это переменная, в то время как `&x` – константа. Операция & применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Ее операндом не может быть ни выражение, ни константа (например, нельзя определить с помощью этой операции адрес константы `&100` или результата выражения `&(a+2)`). Если эту операцию применить к указателю, то результатом будет адрес ячейки памяти, в которой хранится значение указателя.

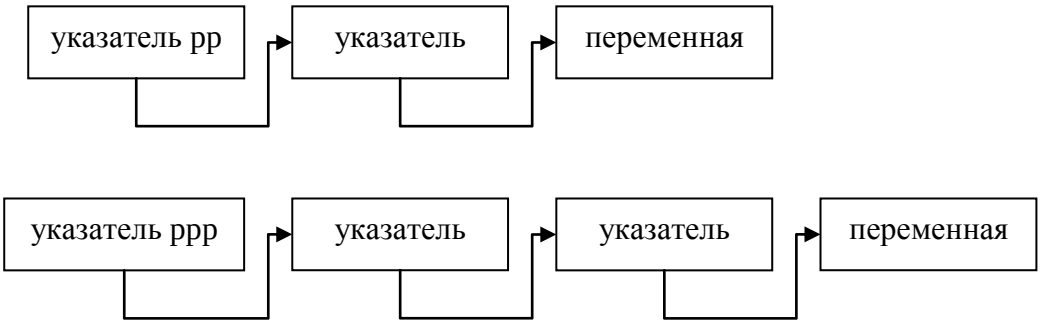
Результат операции & можно использовать в любом выражении, где допускается использование указателя соответствующего типа.

Унарная операция * называется операцией *косвенного доступа*. Примененная к указателю она выдает объект, на который данный указатель указывает. Операцию * можно словами выразить так: «взять содержимое памяти по адресу, равному значению указателя». Результатом операции * является значение того объекта, к адресу которого применялась операция *, тип результата совпадает с типом объекта. Операция * может применяться только к указателям и только в том случае, когда они типизированы. При необходимости применить эту операцию к указателю типа `void*` следует использовать явное преобразование типов.

Результат операции * можно использовать в любом выражении, где допускается использование объекта соответствующего типа.




```
int *p, a, b;
int **pp;    // указатель на указатель на тип int
//int ***ppp;  // указатель на указатель на указатель на тип int
```




```
p = &a;
*p = 10;    // a = 10
p = &b;
*p = 20;    // b = 20
pp = &p;
**pp = 30; // b = 30
```

Поясним присваивания рисунком, в котором прямоугольники изображают ячейки памяти для хранения величин типа `int` и указателей, внутри которых проставлены значения величин, а над ними записаны их названия и адреса. Будем считать, что переменные располагаются в памяти последовательно по мере их объявления и первая переменная начинается в памяти с адреса 100. Длина указателя – 4 байта, длина `int` – 2 байта.

Состояние ячеек до первого присваивания после описания переменных:

p, адрес 100	a, адрес 104	b, адрес 106	pp, адрес 108
мусор	мусор	мусор	мусор

Состояние ячеек после присваивания: `p = &a`

p, адрес 100	a, адрес 104	b, адрес 106	pp, адрес 108
104	мусор	мусор	мусор

Состояние ячеек после присваивания `*p = 10`

p, адрес 100	a, адрес 104	b, адрес 106	pp, адрес 108
104	10	мусор	мусор

Состояние ячеек после присваивания `p = &b`

p, адрес 100	a, адрес 104	b, адрес 106	pp, адрес 108
106	10	мусор	мусор

Состояние ячеек после присваивания `*p = 20`

p, адрес 100	a, адрес 104	b, адрес 106	pp, адрес 108
106	10	20	мусор

Состояние ячеек после присваивания `pp = &p`

p, адрес 100	a, адрес 104	b, адрес 106	pp, адрес 108
106	10	20	100

Состояние ячеек после присваивания `**p = 30`

p, адрес 100	a, адрес 104	b, адрес 106	pp, адрес 108
106	10	30	100

Операции взятия адреса объекта и разыменования указателя – взаимно обратны:

```
int x;
int *px = &x;        // инициализируем адресом x
*(&x) = x;
&(*px) = px;
```

Для указателей одного и того же типа допустимой является операция присваивания. Кроме того, указателю типа `void` может быть присвоено значение адреса данного любого типа, но не наоборот.

```
int *a, *b, c, cc;
double *d;
void *v;
a = b;           // Правильно
v = d;           // Правильно
a = &c;
v = a;           // Правильно
// cc = *v;      // Ошибка !!!!!!!
cc = *(int *)v; // Правильно и cc = c
// b = v;        // Ошибка !!!!!!!
// d = a;        // Ошибка !!!!!!!
```

В случае неправильного присваивания указателей компиляторы обычно выдают предупреждающие сообщения, которыми никогда не следует пренебрегать. В частности и поэтому, в ваших программах не должно быть никаких предупреждений компилятора, т.к. такие предупреждения чаще всего указывают на скрытые ошибки.

Если по какой-либо причине необходимо выполнить операцию присваивания между указателями разного типа, то следует использовать явное преобразование типов:

```
int *a, *b;
double *d;
void *v;
b = (int *)v;     // нормально
d = (double *)a; //  возможны проблемы, т.к. sizeof(double) != sizeof(int)
```

При этом ответственность за корректность подобных операций целиком ложится на программиста. Действительно, в предыдущем примере `a` является указателем на ячейку памяти для хранения величины типа `int`. Обычно это ячейка размером 2 байта. После присваивания указателей с явным преобразования типов, делается возможным обращение к этой ячейке посредством указателя `d`, как к ячейке с величиной типа `double`. Размер этого типа обычно 8 байт, да и внутреннее представление данных в корне отличается от типа `int`. Никакого преобразования самих данных не делается, ведь речь идет только об указателях. Дальнейшая работа с указателем `d` скорее всего заденет байты, соседние с байтами, на которые указывает `a`. Результат интерпретации этих байт будет тоже неверным.

Нельзя указателю присвоить число: `int *a = 100` (это бессмысленно + выдаст ошибку «нельзя преобразовать int к int *»). Указатели и целочисленные переменные не являются взаимозаменяемыми объектами, т.е. хотя адрес – это число, но в тоже время целое число – это не адрес.

С другой стороны, с помощью явных преобразований можно получить указатель на произвольную ячейку памяти. Например, указатель на ячейку памяти 0777000 можно получить с помощью следующей записи: `a = (int *)0777000;`

Обращение к конкретным ячейкам памяти часто бывает необходимо в программах, взаимодействующих с оборудованием, например в драйверах устройств, когда для управления устройствами нужно иметь доступ к таким ячейкам памяти, как регистры состояния или ячейки буфера устройства. Хотя такие возможности полезны и даже необходимы для некоторых приложений, пользоваться ими следует с большой осторожностью.

Есть одно исключение. Значение 0 может быть присвоено указателям любого типа. Гарантируется, что нет объектов с нулевым адресом. Следовательно, указатель, равный нулю, можно интерпретировать как указатель, который ни на что не ссылается. Попытка использовать это значение для обращения к объекту может привести к ошибке. Если делается попытка присвоить какое-либо значение по адресу указателя, значение которого равно нулю, то многие компиляторы выдают сообщение «Null pointer assingment». В языке С определен макрос NULL для представления такого нулевого указателя, описание которого находится в библиотеке <stdio.h> и является системозависимым.

В операциях с указателями участвуют два объекта: сам указатель и объект, на который он указывает. Помещение ключевого слова `const` перед объявлением указателя делает константой объект, а не указатель. Для объявления указателя в качестве константы используется оператор объявления `*const`, а не просто `*`.

```
void main() {
  int a = 1;
  const int c = 22;
  int *p1, *p2;
  int const *pc1;       // указатель на константу типа int
  const int *pc2;       // указатель на константу типа int
  int *const cp = &a;   // указатель-константа на int
```

```
                                  //  (при описании требуется инициализация)
//p1 = &c;      //  нельзя присвоить адрес константы обычному указателю
  pc1 = &c;     //  но можно присвоить указателю на константу
//*pc1 = 100;  // cannot modify a const object
//cp = &c;     // cannot modify a const object
  p1 = &a;
  *p1 = 100;   // a = 100
  pc2 = &a;
//*pc2 = 200;  // cannot modify a const object
  p2 = (int *)pc2;
  *p2 = 200;   // a = 200
}
```

Если указатель указывает на константу, то нельзя изменить то, на что он указвает. Но можно присвоить адрес переменной указателю на константу, т.к. это безопасная операция. Однако, нельзя присвоить адрес константы произвольному указателю, т.к. в этом случае можно будет изменить значение самой константы.

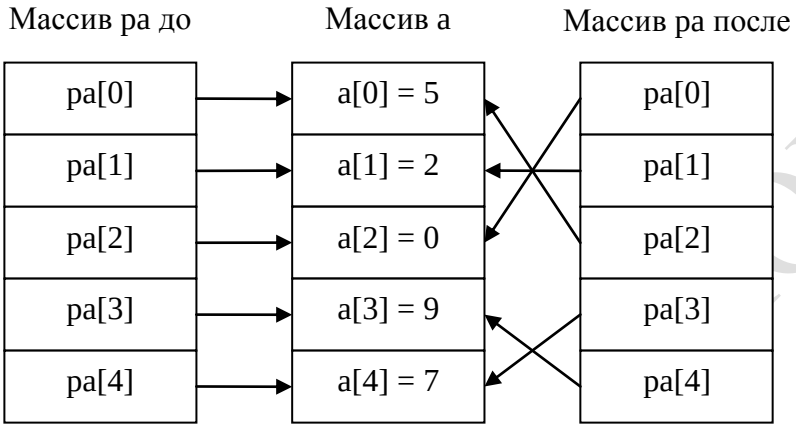
Если указатель сам является константой, то нельзя изменить его значение.

Так как указатели сами являются переменными, то можно также описывать и использовать массивы указателей: `int *a[10]`.

Задача. *Вывести на экран отсортированный массив. Исходный массив не должен измениться. Можно использовать потом отсортированный массив указателей для двоичного поиска в исходном массиве.*

```
void main() {
  int a[10], i, j, n=10;
  int *pa[10], *p;
  printf("Введите массив\n");
  for (i=0; i<n; i++)
    scanf("%d", &a[i]);
  for (i=0; i<n; i++)
    pa[i] = &a[i];          // a+i
  for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
      if (*pa[i] > *pa[j]) {  // сравниваем значения
        p = pa[i];            // меняем местами указатели
        pa[i] = pa[j];
        pa[j] = p;
      }
// === второй способ сортировки ======================
//  for (i=0; i<n; i++)
//    for (j=0; j<n-1; j++)
//      if (*pa[j] > *pa[j+1]) {
//        p = pa[j];
//        pa[j] = pa[j+1];
```

```
//         pa[j+1] = p;
//       }
  printf("Исходный массив\n");
  for (i=0; i<n; i++)
    printf("%d ", a[i]);
  printf("\n");
  printf("Отсортированный массив\n");
  for (i=0; i<n; i++)
    printf("%d ", *pa[i]);
  printf("\n");
}
```



Трудности при работе с указателями.

Ничто не может доставить больше неприятностей, чем ошибочный указатель! Ошибочный указатель трудно найти потому, что ошибка в самом указателе никак себя не проявляет. Проблемы возникают при попытке обратиться к объекту с помощью этого указателя. Если значение указателя неправильное, то программа с его помощью обращается к произвольной ячейке памяти. При чтении в программу попадают неправильные данные, а при записи искажаются другие данные, хранящиеся в памяти, или портится участок программы, не имеющий никакого отношения к ошибочному указателю. В обоих случаях ошибка может не проявиться вовсе или проявиться позже в форме, никак не указывающей на ее причину.

Поскольку ошибки, связанные с указателями, особенно трудно обезвредить, при работе с указателями следует соблюдать особую осторожность. Рассмотрим некоторые ошибки, наиболее часто возникающие при работе с указателями.

1) ***Неинициализированный указатель.*** Описание указателя не является требованием на выделение памяти для хранения данных. Память выделяется только для хранения адреса. Поэтому прежде, чем использовать указатель, ему нужно присвоить значение адреса реально существующего объекта. В противном случае результат работы программы непредсказуем.

```
int *p, a;
a = *p;
*p = 100;
```

В этой последовательности используется указатель, которому предварительно не присвоено никакого значения. В переменной `p` находится произвольное значение. Первая операция присваивания приведет к тому, что переменная a получит значение из двух ячеек памяти с непредсказуемым адресом. Вторая – к тому, что по непредсказуемому адресу будут записаны 2 байта с числом 100. Если эти байты попадут на область данных программы, то программа, скорее всего, выдаст неправильный результат. Если они попадут на область кода программы или на системную область ОС, то в лучшем случае программа аварийно завершится, а в худшем компьютер полностью зависнет. В небольших программах такая ошибка часто остается незамеченной, так как программа и данные занимают немного места в памяти. Самое плохое в затирании памяти – программа может работать случайно, а нет ничего хуже, чем искать ошибку, для которой нет теста, при запуске на котором ошибка появляется всегда.

2) ***Непонятные данные или недоразумения при работе с указателями.*** Еще одна потенциально опасная ситуация: в указатель занесли адрес участка памяти, содержащего данные не того типа, который задан в описании указателя, либо вообще содержащего неизвестно что:

```
int i = 2, j, *pi;
double x = 12.76;
pi = (int *) &x;  // представление double в памяти <> целому
j = *pi;          // чему равно j?
pi = &i;
pi += 7;          // адрес чего находится в pi?
j = *pi;          // чему равно j?
pi = i;           // если компилятор не отругает, то
j = *pi;          // чему равно j?
```

Само присваивание указателю некорректного значения еще не является ошибкой. Ошибка возникнет лишь при обращении к данным по этому указателю (такие ошибки довольно тяжело искать!).

3) ***Неправильное понимание принципов расположения переменных в памяти.*** Программисту ничего не известно о том, как используемые им данные располагаются в памяти, будут ли они расположены так же при следующем выполнении программы или как их расположат другие компиляторы. Поэтому сравнивать одни указатели с другими в общем случае недопустимо (если только это не указатели на элементы одного массива). Похожая ошибка возникает, когда делается необоснованное предположение о расположении массивов (предполагая, что массивы `int a[10], b[10];` расположены рядом, пытаются обращаться к ним с помощью одного и того же указателя).

4) ***Адрес несуществующего объекта.*** Следует также опасаться случая, когда указатель содержит адрес объекта программы, завершившего свое существование.

```
void main(){
  int *p, a, i;
  for (i=0; i < 10; i++) {
```

```
    int b = i + 10;
    p = &b;
  }
  a = *p;  // некорректное присваивание
}
```

Хотя и станет a=19, но если посмотреть в отладчике, то на этой строчке увидим «Undefined symbol b». Т.е. переменной уже нет, а ее значение мы используем. В такой ситуации может 100 раз повезти, а на 101 раз программа выдаст неверный результат. Искать такие ошибки опять же очень долго и очень трудно.

То, что неправильные указатели могут быть очень коварными, не может служить причиной отказа от их использования. Следует лишь быть осторожным и внимательно проанализировать каждое применение указателя в программе.

Задача. *Используя указатели, найти максимальное из двух чисел. Затем увеличить первое число на 1, а второе – умножить на 2.*

```
void main() {
  int a, b, max;
  int *pa, *pb, *pmax;
  pa = &a; pb = &b; pmax = &max;
  printf("Введите a и b");
  scanf("%d %d", pa, pb);
  *pmax = *pa > *pb ? *pa : *pb;
  (*pa)++;                          // про приоритет операций
  *pb *= 2;
  printf("Максимальное число = %d\n", *pmax);
  printf("a = %d\n", *pa);
  printf("b = %d\n", *pb);
}
```

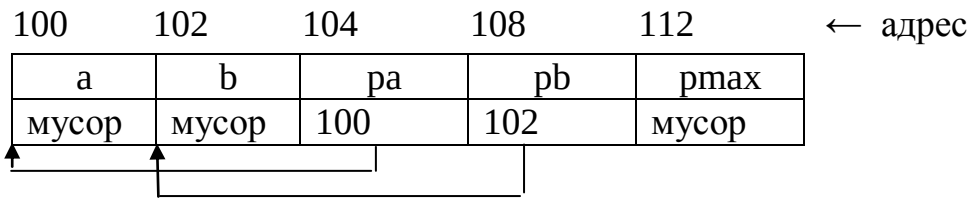
или так можно (без переменной для максимума):

```
void main() {
  int a, b;
  int *pa, *pb, *pmax;
  pa = &a; pb = &b;
  // рисунок после выполнения этих операторов (1)
  printf("Введите a и b");
  scanf("%d %d", pa, pb);
  // рисунок после выполнения этих операторов (вводим 10 и 20) (2)
  pmax = *pa > *pb ? pa : pb;
  // рисунок после выполнения этого оператора (3)
  printf("Максимальное число = %d\n", *pmax);
  // на экране будет "Максимальное число = 20"
  (*pa)++;
  *pb *= 2;
```

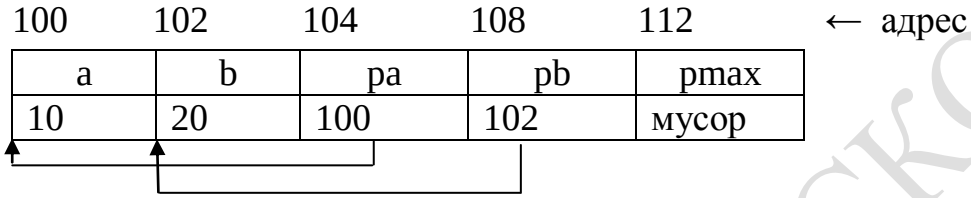
```
  // рисунок после выполнения этих операторов (4)
  printf("a = %d\n", *pa);
  printf("b = %d\n", *pb);
  // на экране будет "a = 11", "b = 40"
}
```

Рисунки для второго случая:

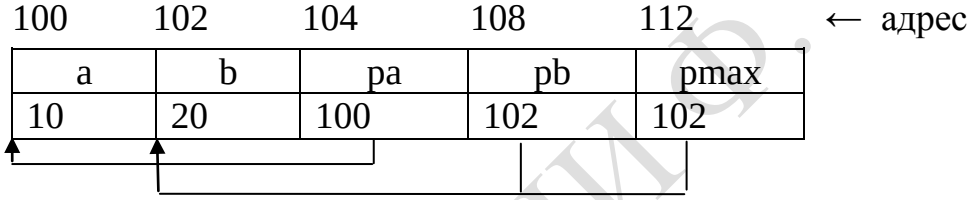
(1)



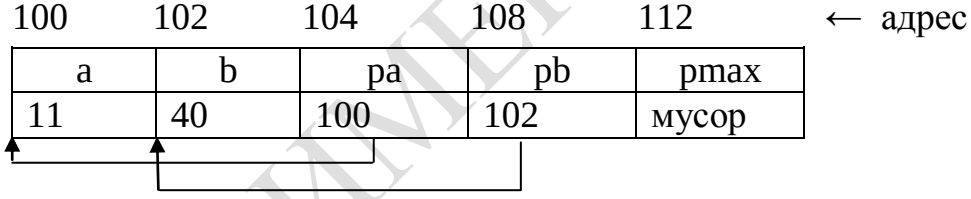
(2)



(3)



(4)



14. АДРЕСНАЯ АРИФМЕТИКА

Адресная арифметика – это способ вычисления адреса какого-либо объекта при помощи арифметических операций над указателями, а также использование указателей в операциях. Адресную арифметику также называют арифметикой над указателями.

Арифметика с указателями дает программисту возможность работать с разными типами одинаковым способом. В то же время мощная арифметика с указателями может стать причиной ошибок. Из-за сложностей использования указателей многие современные языки программирования высокого уровня (например, Java или C#) не разрешают прямой доступ к памяти с использованием адресов. В то же время указатели в С – достоинство этого языка, и не зря ОС пишут в основном на С.

Для указателей-переменных разрешены следующие операции: присваивания, сравнение, сложение, вычитание, инкремент ++ и декремент --.

Присваивание мы уже рассмотрели. Вспомним:

1) указателю можно присвоить адрес переменной (не константы и не выражения);
2) указателю можно присвоить указатель того же типа (или другого с приведением типа);
3) указателю типа `void *` можно присвоить указатель любого типа;
4) любому указателю можно присвоить `NULL`;
5) указателю нельзя присвоить число. Константа ноль – единственное исключение из этого правила: ее можно присвоить указателю.

Язык С разрешает операцию сравнения указателей одинакового типа, при этом, по сути, сравниваются адреса в памяти.

Сравнение указателей в общем случае некорректно! Это связано с тем, что одним и тем же физическим адресам памяти могут соответствовать различные пары значений «сегмент-смещение» (при страничной (сегментной) организации памяти разные страницы (сегменты) виртуальной памяти могут загружать в один и тот же физический блок оперативной памяти).

Но при определенных условиях указатели все же можно сравнивать. Если `p` и `q` указывают на элементы одного и того же массива, то такие отношения, как <, >= и т.д., работают надлежащим образом. Например, `p < q` истинно, если `p` указывает на более ранний элемент массива, чем `q`. Отношения == и != тоже работают.

Также любой указатель можно осмысленным образом сравнить на равенство или неравенство с `NULL`.

Но ни за что нельзя ручаться, если вы используете сравнения при работе с указателями, указывающими на разные массивы. Если вам повезет, то на всех машинах вы получите очевидную бессмыслицу. Если же нет, то ваша программа будет правильно работать на одной машине и давать непонятные результаты на другой.

Задача. *Какое число в массиве находится раньше: ноль или любое положительное?*

```
void main() {
  int a[7] = { -3, 5, 2, 0, -3, 7, 0};
  int *p0 = NULL, *p1 = NULL;
  int i;
  for (i = 0; i < 7; i++) {
    if (a[i] == 0 && p0 == NULL)
      p0 = &a[i];              // можно находить индексы i0 и i1
    if (a[i] > 0 && p1 == NULL)
      p1 = &a[i];
  }
  if (p0 != NULL && p1 != NULL)
    printf("Раньше находится %s\n", p0 < p1 ? "0" : ">0");
  else {
    if (p0 == NULL)
```

```
      printf("Нулей в массиве нет\n");
    if (p1 == NULL)
      printf("Положительных в массиве нет\n");
  }
}
```

Над указателями определены операции сложения и вычитания с целым числом (константой или переменной). Все действия с указателями автоматически учитывают размер объектов, на которые они указывают. Т.е. важной особенностью арифметических операций с указателем является то, что физическое увеличение или уменьшение его значения зависит от типа указателя, т.е. от размера того объекта, на который указатель ссылается.

Если к указателю, описанному как `type *p;`, прибавляется или отнимается константа `N`, значение `p` изменится на `N*sizeof(type)`.

```
int *p, i = 2;
int a[5] = {1, 2, 3, 4, 5};
p = &a[1];  // *p = 2
p++;        // *p = 3, значение p увеличится на 2
p += i;     // *p = 5, значение p увеличится на 4
p--;        // *p = 4, значение p уменьшится на 2
p += 2;     // *p = 2, значение p уменьшится на 4
```

В этом примере унарная операция `p++` увеличивает `p` так, что он указывает на следующий элемент набора объектов того типа, что задан при определении указателя, а именно `int`. Здесь операция `p += i` увеличивает `p` так, чтобы он указывал на элемент, отстоящий на `i` элементов от текущего элемента.

Именно подобное уменьшение или увеличение указателя дает возможность сканировать такие объекты, как строки и массивы. Эти операции применяются чаще всего к указателям, адресуемым элементы одного и того же массива (но никто не запрещает их использовать для любых указателей).

Унарные операции * и & имеют более высокий приоритет, чем арифметические операции, так что присваивание

```
y = *ip + 1;
```

берет объект, на который указывает ip, и добавляет к нему 1, а результат присваивает переменной y. Аналогично

```
*ip += 1;
```

увеличивает на единицу объект, на который указывает ip; те же действия выполняют выражения

```
++*ip;
```

и

```
(*ip)++;
```

В последней записи скобки необходимы, поскольку если их не будет, увеличится значение самого указателя, а не то, на что он указывает. Это обусловлено тем, что унарные операции * и ++ имеют одинаковый приоритет и выполняются справа налево.

Специальное применение имеют указатели на тип `void`. Указатель на `void` может указывать на значения любого типа. Однако для выполнения операций над указателем на `void` либо над указуемым объектом необходимо явно привести тип указателя к типу, отличному от `void`.

```
int i[5] = {1, 2, 3, 4, 5};
void *p;
p = &i[1];
//*p = 10;          // ошибка!!!
*(int *)p = 10;     // правильно и i[1] = 10
//p++;              // ошибка!!!
((int *)p)++;       // правильно и p->i[2]
```

Эти и аналогичные конструкции представляют собой самые простые и самые распространенные формы арифметики указателей или адресной арифметики.

Два указателя на элементы одного и того же массива можно вычитать. Разность двух указателей `type *p1, *p2;` – это разность их значений, поделенная на `sizeof(type)`, т.е. разность двух указателей – целое число, модуль которого определяет число на 1 большее, чем число элементов массива, расположенных между этими указателями (другими словами, получается разность индексов двух элементов массива). Понятно, что результат может быть отрицательным.

Задача. *Какое число в массиве находится раньше: ноль или любое положительное? И сколько элементов массива между ними находится?*

```
if (p0 != NULL && p1 != NULL)
  printf("Раньше находится %s\n", p0 < p1 ? "0" : ">0");
  int k = p0 < p1 ? p1-p0 : p0-p1;
  printf("Между ними %d элементов\n", --k);
else {
...
}
```

За исключением упомянутых выше операций (присваивание, сложение и вычитание указателя и целого, вычитание и сравнение двух указателей), вся остальная арифметика указателей является незаконной. Запрещено складывать два указателя, умножать, делить, сдвигать или маскировать их, а также прибавлять к ним переменные типа `float` или `double`.

Объединение в одно целое указателей, массивов и адресной арифметики является одной из наиболее сильных сторон языка С.

15. МАССИВЫ И УКАЗАТЕЛИ

В языке С массивы и указатели тесно связаны друг с другом. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей. Т.е. доступ к элементам массива можно осуществлять не только с помощью операции `[]`, но и используя механиз указателей. Вариант с ука-

зателями обычно оказывается более быстрым, но и несколько более трудным для непосредственного понимания, по крайней мере, для начинающего.

15.1. Указатели и одномерные массивы

В языке С имя массива – это указатель-константа на первый байт нулевого элемента массива. Другими словами, имя массива имеет значение, равное адресу нулевого элемента массива или равное адресу начала самого массива. Этот указатель отличается от обычных указателей тем, что его нельзя изменить (например, установить на другую переменную), поскольку он сам хранится не в переменной, а является просто некоторым постоянным адресом.

Следствием такой интерпретации имен массивов является то, что для того чтобы установить указатель `p` на начало массива, надо написать:

```
int *p, mas[10];
p = mas;
p = &mas[0];
```

но не

```
p = &mas; // операция & перед одиноким именем массива не нужна
          // и недопустима (нельзя получить адрес константы)!
```

Вывод: когда в программе объявляется массив `int mas[10]`, то этим определяется не только выделение памяти для десяти элементов массива, но еще и определяется указатель-константа с именем `mas`, значение которого равно адресу первого по счету (нулевого) элемента массива (или адресу начала самого массива).

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Если указатель-константа `a` указывает на нулевой элемент массива, то по определению `a+1` указывает на следующий (первый элемент) элемент, `a+i` указывает на `i`-й элемент массива. Значит, если `a+1` указывает на `a[1]`, то `*(a+1)` есть содержимое `a[1]`, если `a+i` – адрес `a[i]`, то `*(a+i)` есть содержимое `a[i]`. Эти замечания справедливы независимо от типа элементов в массиве `a`.

```
int *pa, i = 5, x;
pa = a;             // или так можно написать: pa = &a[0];
x = *a;             // x = 0, т.е. x = a[0]
x = *(a+i);         // x = 5, т.е. x = a[i] i=5
*(a+9) = 99;        // a[9] = 99
x = *pa;            // x = 0, т.е. x = a[0]
x = *(pa+3);        // x = 3, т.е. x = a[3]
*(pa+7) = *(a+2); // a[7] = a[2];
```

Обратить внимание! Выражение `a[i]` в языке С всегда преобразуется к виду `*(a+i)`: `a[i]=*(a+i)`. Т.е. доступ к элементу одномерного массива в языке С физически реализуется через адрес начала массива в памяти и смещения элемента

(индекс элемента) от начала массива. Вот поэтому имя массива и должно быть адресом начала массива.

Если указатель `p` является указателем на элемент массива, то в выражениях его также можно использовать с индексом: `p[i]` идентично `*(p+i)`.

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *p = a+2;   // устанавливаем p на a[2]

int x = p[3];      // x = 5 или x = a[5]
x = *(p+7);        // x = 9 или x = a[9]
```

Вывод: Любое выражение, включающее массивы и индексы, может быть записано через указатели и смещения и наоборот, причем даже в одном и том же утверждении. А элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

Задача. Найти сумму элементов массива. Для доступа к элементам массива использовать указатели.

```
void main() {
  int a[100], n, i;
  int s = 0;
  do {
    printf("Введите количество элементов массива n = ");
    scanf("%d", &n);
  }
  while (n < 0 || n > 100);
  printf("Введите элементы массива\n");
  for (i = 0; i < n; i++)
    scanf("%d", a + i);

  printf("Вы ввели массив\n");
  for (i = 0; i < n; i++)
    printf("%d ", *(a+i));
  printf("\n");
  for (i = 0; i < n; i++)
    s += *(a+i);
  printf("Сумма элементов массива = %d\n", s);
}
```

```
==== второй вариант ====================================

void main() {
  int *pa;
  ...
  printf("Введите элементы массива\n");
  for (pa = a, i = 0; i < n; i++, pa++)
    scanf("%d", pa);

  printf("Вы ввели массив\n");
```

```
  for (pa = a, i = 0; i < n; i++, pa++)
    printf("%d ", *pa);
  printf("\n");
  for (pa = a, i = 0; i < n; i++, pa++)
    s += *pa;
  printf("Сумма элементов массива = %d\n", s);
}
```

==== третий вариант ====================================

```
void main() {
  int *pa, *pa_n;
  ...
  printf("Введите элементы массива\n");
  for (pa = a, pa_n = a + n; pa < pa_n; pa++)
    scanf("%d", pa);
  printf("Вы ввели массив\n");
  for (pa = a, pa_n = a + n; pa < pa_n; pa++)
    printf("%d ", *pa);
  printf("\n");
  for (pa = a, pa_n = a + n; pa < pa_n; pa++)
    s += *pa;
  printf("Сумма элементов массива = %d\n", s);
}
```

==== четвертый вариант ==================================

```
void main() {
  int *pa;
  ...
  pa = a;
  printf("Введите элементы массива\n");
  for (i = 0; i < n; i++)
    scanf("%d", &pa[i]);
  printf("Вы ввели массив\n");
  for (i = 0; i < n; i++)
    printf("%d ", pa[i]);
  printf("\n");
  for (i = 0; i < n; i++)
    s += pa[i];
  printf("Сумма элементов массива = %d\n", s);
}
```

<u>Задача.</u> Что выведется в результате работы программы?

```
void main () {
  char arr[] = {'Ш', 'К', 'О', 'Л', 'А'};
  char *pt;
```

```
  int i;
  pt = arr + sizeof(arr) - 1;
  for(i = 0; i < 5; i++, pt--)
    printf("%c %c\n", arr[i], *pt);
}
```

Может сложиться мнение, что массив и указатель полностью эквивалентны. Однако имеется два существенных отличия массива от указателя:

- массиву при описании выделяется память для хранения всех его элементов, а указателю только для хранения адреса;
- адрес массива навсегда закреплен за именем, то есть имя массива является адресной константой. Указатель является переменной, и операции `p=a` и `p++` имеют смысл; имя массива является константой, и конструкции типа `a=p` или `a++`, или `p=&a` будут ошибочными.

15.2. Указатели и двумерные массивы

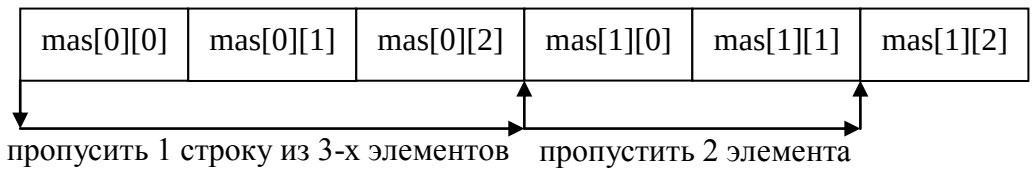
Как и для одномерных массивов, доступ к элементам двумерного массива в программах может осуществляться как по индексу, так и с помощью механизма указателей.

Обратить внимание! Как и для одномерных массивов, выражение `a[i][j]` в языке С всегда преобразуется к виду: *(адрес начала массива + смещение), где смещение определяется уже двумя индексами `i` и `j`. Т.е. доступ к элементу двумерного массива в языке С физически реализуется опять же через адрес начала массива в памяти и смещения элемента (индексы элемента) от начала массива.

Как уже говорилось, двумерный массив – это массив массивов, т.е. такой массив, элементами которого являются массивы. Можно считать, что двумерный массив `mas[n][m]` – это `n` последовательно расположенных одномерных массивов размерностью `m`.

Как определить смещение элемента `mas[i][j]`? Надо пропусть `i` раз по `m` элементов + еще `j` элементов. Как узнать адрес начала массива? Его должно быть можно определить по имени массива. Тогда самый очевидный способ доступа к элементу двумерного массива с помощью указателей такой:

```
int mas[2][3];
int *p = (int *)mas; //  необходимо преобразование типа, т.к. по смыслу
                     //  определения двумерного массива как массива
                     //  массивов mas имеет тип int[3] *
// p = &mas[0][0]; -  можно и так определить адрес начала массива
int x = *(p + 1*3 + 2);  // x = mas[1][2]
```



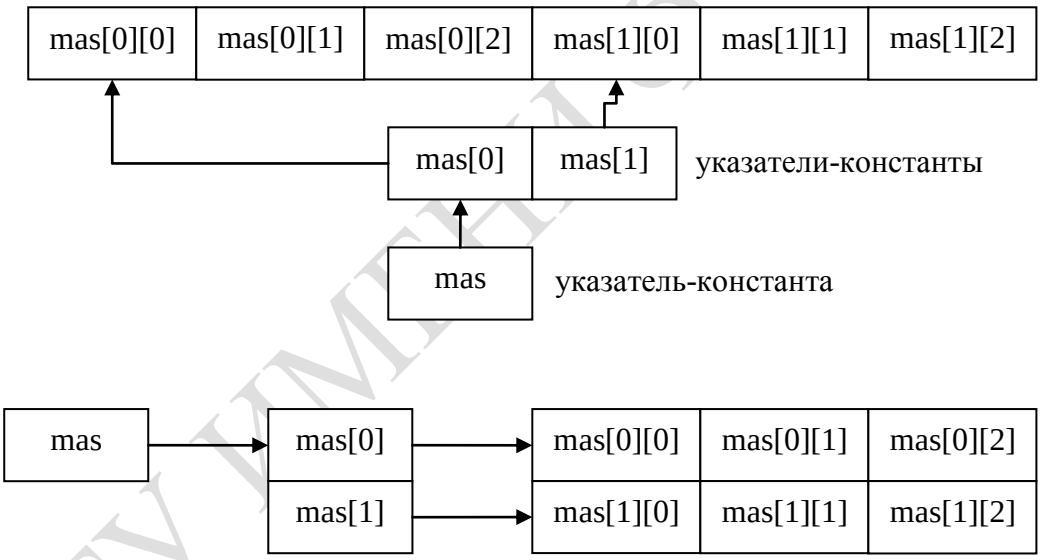
Но так получать доступ к элементам двумерного массива не очень удобно.

С другой стороны, по определению каждая строка двумерного массива – одномерный массив. И, скорее всего, можно стать на нужный элемент, отталкиваясь от адреса соответствующей строки (и это было бы очень удобно). Логично предположить, что `mas[i]` будет адресом начала `i`-ой строки двумерного массива. Но тогда `mas` должно быть адресом массива указателей на строки двумерного массива.

Получаем, что с одной стороны `mas` должно быть указателем-константой на начало массива, а с другой стороны имя двумерного массива должно быть указателем-константой на массив указателей-констант, а элементами массива указателей должны быть указатели-константы на начало каждой из строк массива (одномерный массив).

Вывод: когда в программе объявляется массив в виде `int mas[2][3]`, то этим определяется не только выделение памяти для шести элементов массива, но еще и определяется массив указателей-констант из 2-х элементов `mas[0]` и `mas[1]`, значениями которых являются адреса соответствующих строк массива, а также определяется указатель-константа с именем `mas`, значение которого равно адресу массива указателей-констант с адресами строк массива.

Пусть у нас есть двумерный массив: `int mas[2][3];`



Для нашего двумерного массива `int mas[2][3];` указателями-константами на нулевую и первую строки будут `mas[0]` и `mas[1]`, а следующие выражения будут тождественными:

```
mas[0] == &mas[0][0]
mas[1] == &mas[1][0]
```

Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме: `mas[1]` или `*(mas+1)` – и получим сразу адрес первой строки (т.е. сместимся от начала массива на одну строку `== mas + 1*3`).

Для доступа к элементам двумерного массива используются два индексных выражения в форме: `mas[1][2]`, `*(mas[1]+2)` или `*(*(mas+1)+2)` (можно и так написать `(*(mas+1))[2]`). Т.е сместимся от начала массива на одну строку, взяв адрес этой строки из указателя-константы, а затем сместимся уже в нужной строке на нужное количество элементов. Так мы получим адрес нужного элемента, а операция * даст нам значение этого элемента.

Получить адрес самого первого элемента массива (с индексами [0][0]) можно так:

```
int mas[2][3];
int *p = (int *)mas;
int *p1 = &mas[0][0];
int *p2 = mas[0];
```

Получить значение самого первого элемента массива (с индексами [0][0]) можно так:

```
int xx;
xx = *p;
xx = **mas;       // *(*(m+0)+0)
xx = mas[0][0];   // xx = *p1;
xx = *mas[0];     // xx = *p2;
```

Рассмотрим вывод двумерного массива разными способами.

```
  int i, j;
  int x[2][3] = { {1,2,3},
                  {4,5,6}
                };
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++)
      printf("%4d ", *(x[i]+j));
    printf("\n");
  }
=== или можно так ========================
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++)
      printf("%4d ", *(*(x+i)+j));
    printf("\n");
  }
=== или можно так ========================
  int *px = (int *)x;     // имя массива – адрес его самой первой строки
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++)
      printf("%4d ", *(px + 3*i + j));
    printf("\n");
```

```
  }
```

Задача. Найти максимальные элементы в квадратной матрице и на ее главной диагонали. Для доступа к элементам массива использовать указатели.

```
void main() {
  int a[10][10], n, i, j;
  int max, max_d;
  do {
    printf("Введите размерность матрицы n = ");
    scanf("%d", &n);
  }
  while (n < 0 || n > 10);
  printf("Введите матрицу\n");
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", *(a + i) + j);
  printf("Вы ввели матрицу\n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      printf("%4d ", *(*(a + i) + j));
    printf("\n");
  }
  max = **a;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (max < *(*(a + i) + j))
        max = *(*(a + i) + j);
  printf("Максимальный элемент матрицы = %d\n", max);
  max_d = **a;
  for (i = 0; i < n; i++)
    if (max_d < *(*(a + i) + i))
      max_d = *(*(a + i) + i);
  printf("Максимум на диагонали матрицы = %d\n", max_d);
}
```

Есть ли разница между двумерным массивом и массивом указателей? Для двух следующих определений:

```
int a[10][20];
int *b[10];
```

записи `a[5][7]` и `b[5][7]` будут синтаксически правильным обращением к некоторому значению типа `int`. Однако только `a` является классическим двумерным массивом: для двухсот элементов типа `int` будет выделена память, а вычисление смещения элемента `a`[*строка*][*столбец*] от начала массива будет вестись по формуле 20 * *строка* + *столбец*, учитывающей его прямоугольную природу. Для `b` же определено только 10 указателей, причем без инициализации. Инициализация

должна задаваться явно – либо статически, либо в программе. Предположим, что каждый элемент b указывает на массив из 20 элементов, в результате где-то будет выделено пространство, в котором разместятся 200 значений типа int, и еще 10 ячеек будет выделено для указателей.

Важное преимущество массива указателей в том, что элементы такого массива могут иметь разные длины. Таким образом, каждый элемент массива b не обязательно указывает на массив из 20 элементов; один может указывать на два элемента, другой — на пятьдесят, а некоторые и вовсе могут ни на что не указывать.

Задача. Что увидим на экране?

```
void main() {
  int xx[3][3] = { {1,1,1},     // 1 строка вывода
                   {2,2,2},     // 3 строка вывода
                   {3,3,3}
                 };
  int yy[3][3] = { {5,5,5},     // 4 строка вывода
                   {7,7,7},
                   {9,9,9}      // 2 строка вывода
                 };
// ===  вроде как массив [4][3]  ====================
  int *x[] = {xx[0], yy[2], xx[1], yy[0], NULL};
  int i=0;
  while (x[i]) {
    printf("%d %d %d\n", *x[i], *(x[i]+1), *(x[i]+2));
//  printf("%d %d %d\n", x[i][0], x[i][1], x[i][2]);
    i++;
  }
}
```

Задача: найти максимум в двух массивах xx и yy. Можно искать два максимума и выбирать больший из них. А можно аналогичным образом составить один массив и искать максимум в одном массиве.

16. СТРОКИ

В языке С нет отдельного типа для строк. Работа со строками реализована через массивы. Хотя в других языках программирования имеется такой тип данных как string – строки.

Строка в Паскале – специальный тип данных string. Тип string – это, по существу, массив array [0..255] of char. Первый его элемент (байт с номером 0) задает динамическую длину строки (реальную длину строки), которая может принимать значения от 0 до 255 символов. Символы, составляющие строку, занимают места от 1 до 255.

```
var
```

```
s : string;
begin
  s:= '1234567';
  writeln(Ord(s[0]));  // увидим длину строки 7
end.
```

В других средах программирования на Паскале могут быть типы строковых данных с другими максимальными длинами (например, тип `AnsiString`). Переменная типа `AnsiString` – это динамически распределяемые массивы символов, максимальная длина которых ограничивается только наличием памяти. При определении таких переменных память для них не выделяется, а выделяется память только под указатель, который хранит адрес самой строки. Под строку память будет выделена при присваивании или процедурой `SetLength`.

В языке С символьная строка – это одномерный массив типа `char`, заканчивающийся нулем (символом с кодом 0, ноль-символом, символьной константой `'\0'`).

Если объявить строку `s` в виде `char s[10]`, то для хранения этой строки будет отведено 10 байт памяти. К каждому символу строки можно обращаться, как к элементу массива: `s[0]`, `s[1]` и т.д. При этом реальная длина строки может быть и меньше 10 символов. Признаком окончания строки является символ с нулевым ASCII-кодом. Если в строке `s` записано слово `"C++"`, то `s[0]='C'`, `s[1]='+'`, `s[2]='+'`, `s[3]=0`, а во всех остальных элементах массива `s` могут быть записаны произвольные символы.

C	+	+	\0	-	-	…	-

Длина строки (т.е. число символов, предшествующих `'\0'`) нигде явно не хранится. Длина строки ограничена лишь размером массива, в котором сохранена строка, и может изменяться в процессе работы программы в пределах от **0** до **длины массива-1**. Таким образом, максимальная длина строки, объявленной как char s[n] может быть `n-1` символ, поскольку один символ требуется для хранения нулевого символа, завершающего строку. При работе со строками надо быть аккуратными и не вылезать за границы массива-строки, поскольку язык С не контролирует подобные ошибки. Ошибки, связанные с переполнением строк, наряду с ошибками при работе с указателями, являются основными причинами затирания памяти.

Язык С допускает строковые константы. Строковая константа – это набор символов в двойных кавычках. В конце строковой константы не нужно ставить символ `'\0'`. Это сделает компилятор автоматически. Например, константа `"Borland C++"` будет выглядеть в памяти как вот такой массив символов:

B	o	r	l	a	n	d		C	+	+	\0

Адрес строки (этот адрес указывает на первый символ строки)

Значение строковой константы – это адрес ее первого символа.

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

Пустая строка – это строка с нулевой длиной. Чтобы получить пустую строку, надо в нулевой символ строки занести 0. Пустая строка – константа это константа "" (не надо писать так "\0").

```
char s[80];
s[0] = 0;     // или *s = 0     или *s = '\0'
```

Ввод и вывод строк осуществляется так же, как и ввод-вывод других типов данных: с помощью scanf() и printf():

```
char s[80];
scanf("%s",s); // & перед s писать не надо, т.к. s и есть адрес строки,
               // как указатель-константа на начало массива
printf("%s",s);
```

Есть еще специальные функции для ввода и вывода только строк:

```
gets(s);        // ввод строки
puts(s);        // вывод строки
```

Между printf() и puts() отличий нет, кроме того, что в printf() помимо строки можно еще вывести какой-нибудь поясняющий текст, например. Отличие между scanf() и gets(): scanf() вводит строку или до конца строки (\n), или до первого пробела; gets() – вводит всю набранную строку до конца. Если ввести, например, строку «мама мыла» и нажать Enter, то scanf() введет только « Ама», а gets() – введет всю строку «мама мыла».

При выводе на экран строки выводятся все символы строки вплоть до завершающего нулевого символа. Нулевой символ на экран не выводится.

При вводе строки считываются все введенные символы (для scanf() или до конца или до пробела, причем пробел не считывается). При этом строка автоматически будет дополнена завершающим нулевым символом.

При описании строки можно инициализировать.

```
char s1[] = "123456";              // строго 7 символов
char s2[10] = "123456";            // последние 4 символа будут 0
char s3[] = {'1','2','3','4','5','6','\0'};
char s4[10] = {'1','2','3','4','5','6'}; // последние символы
                                   // будут 0
char s5[] = {'1','2','3','4','5','6'};   // это не строка!!!

char s6[6] = "";                   // это пустая строка
```

1	2	3	4	5	6	\0			
1	2	3	4	5	6	\0	\0	\0	\0
1	2	3	4	5	6	\0			
1	2	3	4	5	6	\0	\0	\0	\0
1	2	3	4	5	6				
\0	\0	\0	\0	\0	\0				

В языке С не предусмотрены какие-либо операции для обработки всей строки символов целиком. Есть функции для работы со строками, их мы рассмотрим позже. Но так как строка, это массив символов, то мы можем обрабатывать строки посимвольно, обращаясь к отдельным элементам строки через их индексы.

Указатели в сочетании с операцией инкремента естественным образом используются для сканирования строк. В таком контексте динамически меняющийся указатель на строку часто называют просто строкой, хотя реально это не так.

Задача. Подсчитать длину строки.

```
void main(){
  char s[80];
  int n = 0;
  gets(s);
  while(s[n] != 0)    // while(s[n])    while(s[n]!='\0')
    n++;
  printf("n = %d\n", n);
}
```

Задача. Подсчитать длину строки, используя указатель.

```
void main(){
  char s[80], *ps;
  int n = 0;
  gets(s);
  ps = s;
  while(*ps) {
    n++;
    ps++;
  }
  printf("n = %d\n", n);
}
```

```
=======  второй вариант ============================
```

```
void main(){
  char s[80], *ps;
  gets(s);
  ps = s;
  while(*ps)
```

```
    ps++;
  printf("n = %d\n", ps-s);
}
```

Задача. Сколько цифр в слове?

```
void main(){
  char s[80];
  int i = 0, n = 0;
  gets(s);
  while(s[i] != '\0') {
    if (s[i] >= '0' && s[i] <= '9') // символ цифра?
      n++;
    i++;
  }
  printf("В слове %s цифр %d\n", s, n);
}
```

Задача. К строке 1 присоединить строку 2.

```
void main(){
  char s1[80],s2[80];
  int i = 0,j = 0;
  gets(s1);
  gets(s2);
  while(s1[i])       // становимся в конец строки s1
    i++;
  while(s2[j]) {     // добавляем к s1 строку s2
    s1[i]=s2[j];     // s1[i++]=s2[j++];
    i++;             //
    j++;             //
  }
  s1[i]=0;    // не забывать ставить символ конца строки!!!
  puts(s1);
}
```

```
=======  второй вариант ============================
```

```
void main(){
  char s1[80],s2[80];
  int i,j;
  gets(s1);
  gets(s2);
  for (i=0; s1[i]; i++);
  for (j=0; s2[j]; i++, j++)
    s1[i]=s2[j];
  s1[i]=0;
  puts(s1);
}
```

Задача. *К строке 1 присоединить строку 2, используя указатели.*

```
void main(){
  char s1[80],s2[80];
  char *ps1,*ps2;
  gets(s1);
  gets(s2);
  ps1 = s1;
  ps2 = s2;
  while(*ps1)
    ps1++;
  while(*ps2) {
    *ps1 = *ps2;
    ps1++;
    ps2++;
  }
  *ps1 = 0;
  puts(s1);
}
```

Задача. *Удалить из строки все вхождения заданного символа.*

```
void main() {
  char s[80], ss[80], c;
  int i = 0, j = 0;
  gets(s);
  c = 'a';
  while (s[i]) {
    if (s[i] != c) {
      ss[j]=s[i];    // создаем новую строку
      j++;
    }
    i++;
  }
  ss[j]=0;
  puts(ss);
}
```

```
=======  второй вариант ============================
void main() {
  char s[80], c;
  int i = 0, j;
  gets(s);
  c = 'a';
  while (s[i]) {
    if (s[i] == c) {
      j = i;
      while (s[j])   // смещаем хвост
```

```
        s[j++] = s[j+1];
    }
    else i++;
  }
  puts(s);
}
```

======= третий вариант ============================

```
void main() {
  char s[80], c;
  char *ptr, *ptr_rez;
  ptr = ptr_rez = s;
  gets(s);
  c = 'a';
  while (*ptr) {
    if (*ptr!=c) {
      *ptr_rez=*ptr;  // используем два указателя
      ptr_rez++;
    }
    ptr++;
  }
  *ptr_rez=0;
  puts(s);
}
```

Задача. Дана строка. Создать подстроку длиной n>0, начиная от символа с номером k>=0.

```
void main(){
  char s[80], ss[80];
  int k, n, i;
  gets(s);
  scanf("%d%d", &n, &k);
  for(i = 0; i < k; i++)
    if(!s[i]) break;
  if (i < k)
    *ss = 0;          //ss[0] = 0;
  else {
    for(i = 0; i < n && s[i+k]; i++)
      ss[i] = s[i+k];
    ss[i]=0;
  }
  puts(ss);
}
```

Задача. Дана строка не нулевой длины. Добавить в начало строки n раз последний символ строки и в конец строки n раз первый символ строки.

```
void main(){
  char s[80], c_end;
  int i = 0, j, n;
  gets(s);
  scanf("%d", &n);
  while(s[i])
    i++;
  c_end = s[i-1];
  for (j = 0; j < n; j++)
    s[i+j] = s[0];         // s[i+j] = *s
  s[i+j] = 0;
  j += i;
  for (i = j; i >= 0; i--)
    s[i+n] = s[i];
  for (i = 0; i < n; i++)
    s[i] = c_end;
  puts(s);
}
```

======= второй вариант ============================

```
void main(){
  char s[80], c_end;
  int i = 0, j, n;
  gets(s);
  scanf("%d", &n);
  while(*(s+i))
    i++;
  c_end = *(s+i-1);
  for (j=0; j<n; j++)
    *(s+i+j) = *s;
  *(s+i+j) = 0;
  j += i;
  for (i = j; i >= 0; i--)
    *(s+i+n) = *(s+i);
  for (i = 0; i < n; i++)
    *(s+i) = c_end;
  puts(s);
}
```

======= третий вариант ============================

```
void main(){
  char s[80], *ps, *p, c_end;
  int n;
  gets(s);
  scanf("%d", &n);
  ps = s;
  while(*ps)
```

```
    ps++;
  c_end = *(ps-1);
  p = ps + n;
  for (; ps != p; ps++)
    *ps = *s;
  *ps = 0;
  p = ps;
  ps += n;
  for (; p >= s; p--, ps--)
    *ps = *p;
  for (; ps >= s; ps--)
    *ps = c_end;
  puts(s);
}
```

<u>Задача.</u> Дано предложение, слова в котором разделены произвольным количеством пробелов. Найти количество слов в предложении.

```
void main(){
  char s[80];
  int n = 0, i = 0;
  gets(s);
  while (s[i]) {
    while (s[i]==' ')
      i++;
    if (!s[i]) break;
    n++;
    while (s[i]!=' ' && s[i])
      i++;
  }
  printf("n = %d\n", n);
}
```

```
=======  второй вариант ============================
```

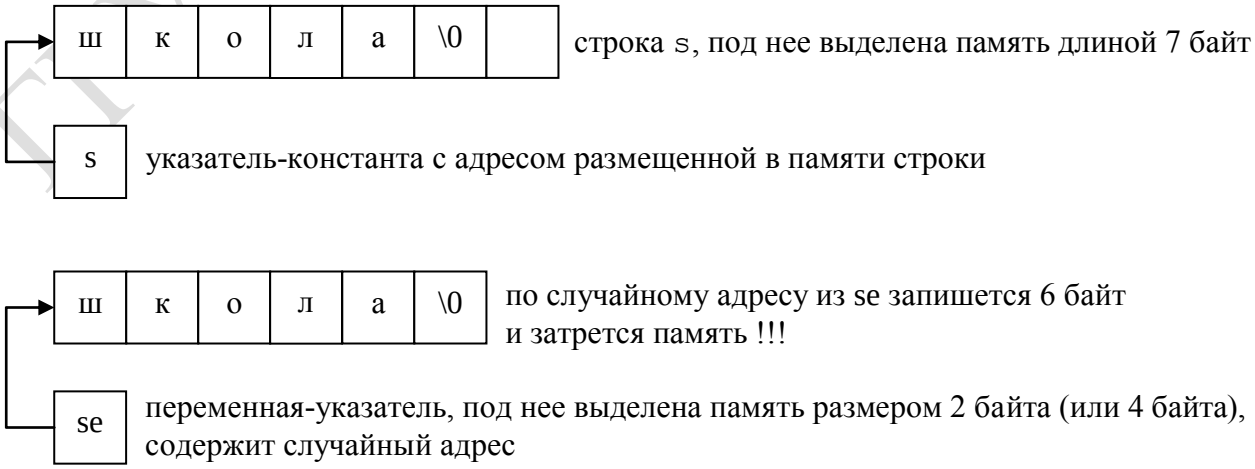
```
void main(){
  char s[80], c = ' ';
  int n = 0, i;
  gets(s);
  for (i=0; s[i]; i++) {
    if (s[i]!=' ' && c==' ')
      n++;
    c = s[i];          // сохраняем предыдущий символ
  }
  printf("n = %d\n", n);
}
```

<u>Задача.</u> Дано предложение. Оставить от каждого слова только первую букву, поставив после нее длину слова (“заяц” => “з4”).

```
#include <stdlib.h>    // для itoa()
void main(){
  char s[80], ss[80], c = ' ', t[3];
  int i, j=0, n, k;
  gets(s);
  for (i=0; s[i]!=0; i++) {
    if (s[i]!=' ' && c==' ') {
      ss[j] = s[i];
      j++;
      n = i;
      while (s[i]!=' ' && s[i])
        i++;
      n = i - n;
      itoa(n,t,10);
      for (k=0; t[k]; ss[j++]=t[k++]);
      ss[j++] = ' ';
      i--;
    }
    c = s[i];
  }
  if (j > 0)
    j--;
  ss[j] = 0;
  puts(ss);
}
```

Поскольку имя массива фактически является указателем на первый элемент массива, переменные типа строк могут также рассматриваться, как имеющие тип `char *`. Но это не значит, что, написав `char *s`, мы описываем строку. Мы описываем указатель на строку. В этот указатель можно занести адрес реальной строки (размещенной в памяти).

```
char s[7];        char *se;
gets(s);          gets(se);
```



```
char s[80];  // описали строку
char *ps;    // описали указатель на строку
ps = s;      // указатель ps стал указывать на строку s
gets(ps);    // введенная строка сохранится в памяти,
             // выделенной для переменной s
char *p = "12345"; // описали указатель и в него занесли
                   // адрес константы "12345"
                   // Изменять такую строку нельзя!!!
```

Обратить внимание! Важно осознать, что при выполнении операции присваивания в ячейки памяти, отведенные для указателя, пересылается не массив символов, а только указатель на его начала, т.е. адрес первого символа строковой константы.

```
//s = p;     // ошибка!!!!!!!!!
ps = p;      // в ps занесли адрес константы "12345"
             // Это не копирование строки!!!
```

Как определить, что символ цифра: `s[i]>='0' && s[i]<='9'`

17. МАССИВЫ СТРОК

Массив строк представляет собой обычный двумерный массив символов. При этом обязательно должно выполняться одно условие – в каждой строке такого массива должен быть ноль-символ, который определяет конец соответствующей строки массива.

Описать массив строк можно так же, как и любой другой двумерный массив.

```
char mas[5][10]; // будет выделена память для хранения 5 строк, каждая
                 // из которых может иметь длину до 9 символов.
                 // Первоначально содержится случайная информация
```

200	?	?	?	?	?	?	?	?	?	?
210	?	?	?	?	?	?	?	?	?	?
220	?	?	?	?	?	?	?	?	?	?
230	?	?	?	?	?	?	?	?	?	?
240	?	?	?	?	?	?	?	?	?	?

Чтобы этот массив стал массивом строк, в каждой строке должен появиться ноль-символ. Тогда строкой будет все то, что находится в соответствующей строке массива до ноль-символа.

```
mas[0][0]='G'; mas[0][1]='G'; mas[0][2]='U'; mas[0][3]=0;

for (i=1; i<5; i++)
  mas[i][0]=0;
```

200	G	G	U	0	?	?	?	?	?	?
210	0	?	?	?	?	?	?	?	?	?
220	0	?	?	?	?	?	?	?	?	?

230	0	?	?	?	?	?	?	?	?	?
240	0	?	?	?	?	?	?	?	?	?

Имеем одну строку "GGU" (mas[0]) и четыре пустые строки (mas[1], mas[2], mas[3], mas[4]). Т.е. имя массива с одним индексом является строкой символов.

Массив строк можно инициализировать при описании:

```
char mas [5][10] = {“май”, “июль”, “август”};
```

200	м	а	й	0	0	0	0	0	0	0
210	и	ю	л	ь	0	0	0	0	0	0
220	а	в	г	у	с	т	0	0	0	0
230	0	0	0	0	0	0	0	0	0	0
240	0	0	0	0	0	0	0	0	0	0

Или так (количество строк определится автоматически):

```
char mas[][10]={"май", "июль", "август"};
```

200	м	а	й	0	0	0	0	0	0	0
210	и	ю	л	ь	0	0	0	0	0	0
220	а	в	г	у	с	т	0	0	0	0

При работе с массивом строк удобно использовать массивы указателей. Память выделяется для 3 указателей (по 4 байта на каждый, например). В каждый указатель занесется адрес соответствующей константы. Правда такие строки нельзя изменять, иначе запортятся константы.

```
char *mas[3]={"май", "июль", "август"};
```

200	500				504				509			
212												

500	м	а	й	0			
504	и	ю	л	ь	0		
509	а	в	г	у	с	т	0

В этом случае каждый элемент массива представляет собой адрес соответствующей строки символов, а сами строки располагаются компилятором в области констант (в сегменте данных загрузочного модуля программы). Никакой лишней памяти, связанной с различной длиной строк, при этом не расходуется.

Задача. *Ввести и вывести массив строк. Конец ввода – пустая строка.*

```
void main() {
  int n = 0, i;
  char s[10][80];
  gets(s[n]);
  while (s[n][0] != 0) { //*s[n]
    n++;
    gets(s[n]);
```

```
  }
  for (i=0; i<n; i++)
    puts(s[i]);
}
```

======= второй вариант ============================

```
void main() {
  int n = 0, i;
  char s[10][80];
  while (*gets(s[n]))  // gets возвращает адрес строки s[n]
    n++;
  for (i=0; i<n; i++)
    puts(*(s+i));      //адрес строки содержится в s[i]
}
```

Задача. Дано предложение. Сформировать массив слов из слов предложения.

```
void main() {
  int n=0, i=0, j=0;
  char s[80], w[10][80];
  gets(s);
  while (s[i]!=0) {
    while(s[i]==' ')
      i++;
    if (s[i]==0) break;  // чтобы не пропустить 0-символ
    while (s[i]!=' ' && s[i]!=0) {
      w[n][j]=s[i];
      j++;
      i++;
    }
    w[n][j]=0;
    n++;
    j=0;
  }
  for (i=0; i<n; i++)
    puts(w[i]);
}
```

======= второй вариант ============================

```
void main(void){
  char s[80],w[20][80];
  int i=0,j,n=0;
  gets(s);
  while(s[i]){
    while(s[i]==' ') i++;
    if(!s[i])
      break;
    for(j=0; s[i]!=' ' && s[i]; w[n][j++]=s[i++]);
```

```
      w[n++][j]=0;
    }
}
```

18. ФУНКЦИИ

В большинстве языков программирования предусмотрены средства, позволяющие оформлять вспомогательный алгоритм как подпрограмму. Это бывает необходимо тогда, когда какой-либо алгоритм неоднократно повторяется в программе или имеется возможность использовать некоторые фрагменты уже разработанных ранее алгоритмов. Кроме того, подпрограммы применяются для разбиения крупных программ на отдельные смысловые части в соответствии с модульным принципом в программировании (считается хорошим тоном, если размер подпрограммы не превышает размера экрана).

Подпрограмма – это последовательность операторов, которые определены и записаны только в одном месте программы, однако их можно вызвать для выполнения из одной или нескольких точек программы. Каждая подпрограмма определяется уникальным именем.

Самое важное, что нужно усвоить: подпрограмма – это группа операторов, у которой есть имя. Подпрограммы используются в трех основных случаях:

1. Если операторы, составляющие тело подпрограммы, встречаются в программе много раз. Тогда имеет смысл создать для них отдельную подпрограмму. При этом произойдёт сокращение кода и, при необходимости, изменения надо будет вносить только в одно место.

2. Для организации кода, если несколько операторов программы выполняют какую-то конкретную задачу. При этом размер кода может и не уменьшится, но программа станет более читабельной.

3. Если подпрограмма может потребоваться в дальнейшем.

18.1. Определение функции в языке С

Принципы программирования на языке С основаны на понятии функции. В языке С все подпрограммы являются функциями. Любая программа на языке С состоит из одной или нескольких функций, причем обязательно должна быть функция с именем `main()` (или `WinMain()`). Выполнение программы всегда начинается с команд, содержащихся в функции `main()`, затем последняя вызывает другие функции.

Формат ***определения*** функции:

```
[тип_возвращаемого_значения] имя_функции (список_параметров) {
  описание данных
  операторы
  [return выражение;]
}
```

Рассмотрим, например, функцию, которая вычисляет сумму двух целых чисел:

```
int sum(int a, int b) {
```

```
  int s;
  s = a+b;
  return s;
}
```

Сначала указан тип значения, которое функция возвращает, – `int`. Затем после пробела следует имя функции – идентификатор, составленный по тем же правилам, что и для имен переменных. *Имя функции должно отражать то, что функция делает!* После имени функции в круглых скобках перечислены формальные параметры функции с указанием их типов. В нашей функции это `a` типа `int` и `b` типа `int`.

После круглых скобок со списком формальных параметров в фигурных скобках следует блок с операторами функции.

Операторы в фигурных скобках называются телом функции. В теле функции можно описывать переменные. Также в теле функции могут быть любые допустимые операторы языка С.

Обратить внимание! Нельзя ставить точку с запятой после `main()`. Нельзя опускать круглые скобки после `main`. Это касается всех функций!

Обратить внимание! В языке С запрещено определять одну функцию внутри другой (в теле другой функции).

18.2. Возвращение значений из функции

Поле [тип_возвращаемого_значения] задает тип возвращаемого функцией значения. Если оно отстутствует, считается, что функция возвращает значение типа `int`. Если тип возвращаемого значения задан как `void`, считается, что функция не возвращает никакого значения (это по сути своей процедура).

Функция может возвращать любой тип данных за исключением массивов.

Передача значения из вызванной функции в вызвавшую происходит с помощью оператора возврата `return`.

Задача. Написать функцию для нахождения минимума из двух целых чисел.

```
int min(int a, int b) {
  int m;
  if (a < b) m = a;     //  m = a;
  else m = b;           //  if (b < a) m = b;
  return m;
}
```

Задача. Написать функцию для определения, есть ли в массиве заданное число (1 – есть, 0 – нет).

```
int find(int m[10], int k) {
  int i, p = 0;
  for (i=0; i<10; i++)
    if (m[i] == k) {
```

```
      p = 1;
      break;
    }
  return p;
}
```

Операторов `return` в функции может быть несколько, и тогда они фиксируют соответствующие точки выхода:

```
int min(int a, int b) {
  if (a < b) return a;
  return b;
}

int find(int m[10], int k) {
  int i;
  for (i=0; i<10; i++)
    if (m[i] == k) return 1;
  return 0;
}
```

В операторе `return` можно записывать выражения. Тогда вернется результат вычисления выражения:

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

int min(int a, int b) {
  return a<b ? a : b;
}
```

После слова `return` можно ничего не записывать. В этом случае вызвавшей функции никакого значения не передается (тип возвращаемого значения `void`).

```
void f() {
  ...
  return;  // можно опустить
}
```

Если с помощью `return` ничего не возвращается, его писать не обязательно.

Управление передается вызвавшей функции и при выходе «по концу» (последняя закрывающая фигурная скобка).

Но бывают ситуации, когда оператор `return` без значения полезен.

Задача. Поменять знак у элементов массива до первого 0. Если 0 нет, то во всем массиве.

```
void negative(int m[5]) {
  int i;
  for (i=0; i<5 && m[i]!=0; i++)
    m[i] = -m[i];
  return;                    // можно опустить
```

```
}
void negative(int m[5]) {
  int i;
  for (i=0; i<5; i++) {
    if (m[i] == 0) return;     // if (!m[i]) return;
    m[i] = -m[i];
  }
  return;                      // можно опустить
}
```

В большинстве функций для завершения выполнения используется оператор `return` – или потому, что необходимо вернуть значение, или чтобы сделать код функции проще и эффективнее.

Обратить внимание! В функции, тип которой отличен от `void`, в операторе `return` необходимо *обязательно* указать возвращаемое значение. То есть, если для какой-либо функции указано, что она возвращает значение, то внутри этой функции у любого оператора `return` должно быть свое выражение. Однако если функция, тип которой отличен от `void`, выполняется до самого конца (то есть до закрывающей ее фигурной скобки), то возвращается произвольное (непредсказуемое с точки зрения разработчика программы!) значение. Хотя здесь нет синтаксической ошибки, это является серьезным упущением и таких ситуаций необходимо избегать.

По стандарту языка С функция `main()` возвращает значение типа `int`, поэтому правильнее будет писать так:

```
int main(){
  ...
  return 0;
}
```

Функция `main()` – главная в программе. Она возвращает значение тому, кто главнее ее, т. е. запустившей ее ОС.

В то же время использовать `void main(){}` также можно.

18.3. Формальные и фактические параметры функции

После имени функции в круглых скобках перечислены формальные параметры с указанием их типов. Формальные параметры разделены запятыми. В теле функции ими пользуются так же, как обычными переменными.

Функция может быть и без параметров, тогда их список будет пустым. Такой пустой список можно указать в явном виде, поместив для этого внутри скобок ключевое слово `void` или просто ничего не указывая в круглых скобках.

```
void prnErr() {       // void prnErr(void)
  printf("Ошибка!");
}
```

Аргументы, передаваемые функции при ее вызове, называются фактическими параметрами. Фактические параметры – это то, что стоит на самом деле при вызове функции. А при вызове функции в качестве фактических параметров могут стоять:

имена переменных (такие же или совершенно другие), выражения или просто константы.

Значения фактических параметров заносятся в соответствующие формальные параметры, т.е. фактические параметры как бы замещают формальные параметры при вызове функции.

Обратить внимание! Тип должен указываться для каждого формального параметра в отдельности:

```
int sum(int a, b) { // ошибка
  ...
}
```

18.4. Вызов функции

Когда имя функции встречается в выполнимой инструкции программы, говорится, что в этой точке функция вызывается. Основная идея заключается в том, что вызов функции выполняет ее тело. Каждое выполнение функции начинается с начала тела функции и, в конечном счете, возвращает управление в точку, находящуюся непосредственно за вызовом функции.

Определив функцию, мы можем ее неоднократно вызывать, задавая в качестве фактических параметров нужные нам переменные или значения.

```
s = sum(a,b);   // вызов функции с параметрами
printErr();     // вызов функции без параметров – скобки () обязательны !!!
```

Если функция не объявлена как имеющая тип `void`, она может использоваться как операнд в выражении. Объявляя функцию как возвращающую значение типа `void`, мы запрещам ее применение в выражениях, предотвращая таким образом случайное использование этой функции не по назначению.

Вызов функции не может находиться в левой части оператора присваивания. Выражение `sum(x,y)=100;` является неправильным.

```
void main() {
  int x = 10, y = 7, res;
  res = sum(x, y);
  // вызов функции в выражениях
  res = sum(x, y) + sum(5, 9) + sum(x+y, 100);
  if (sum(x,y) > 0)
    printf("Сумма положительная");
  if ((res = sum(x,y)) < 0)
    printf("Сумма отрицательная = %d", res);
  printf("%d", sum(x, y));  // печатаем возврат
  sum(x,y);                 // игнорируем возврат
}
```

При вызове функции мы можем использовать то значение, которое она возвращает, а можем его игнорировать (если нам просто надо, чтобы выполнились операторы в теле функции).

На первый взгляд возможность игнорировать возвращаемое значение может показаться странной и даже бессмысленной. Но в программах на С очень часто встречаются такие функции, возвращаемое значение которых содержит некоторую «не всегда нужную» программисту информацию. И очень хороший пример такой функции – `printf()`. Мы с вами этой функцией пользовались, и вы, даже не подозревали, что эта функция к тому же возвращает какой-то результат. А она возвращает значение типа `int`. Если число, которое она вернула, отрицательное – функции «что-то не понравилось». Если 0 или больше, то это длина напечатанной ей строки в символах. Такое отбрасывание возвращаемого значения встречается очень часто.

18.5. Объявление и определение функции: прототип функции

Определения (definition) функции следует отличать от ее объявления (declaration).

Определение функции – это ее полное описание, включающее тип возвращаемого значения, количество и типы самих параметров, и ее тело – тот код, который стоит внутри фигурных скобок после имени функции. Когда мы писали (создавали, а не вызывали) функции, мы выполняли именно определения функций.

Что же такое объявление функции, и зачем оно понадобилось? Допустим, что есть такой код:

```
void main() {
  int i = 10;
  f(i);
  ...
}
```

Что по ним можно сказать о функции `f()`? На первый взгляд, это функция, которой нужен один аргумент типа `int`. С возвращаемым значением ясности нет – либо `void`, либо любой другой тип, но значение в коде игнорируется.

А теперь посмотрим на определение этой функции:

```
void f(float x) {
  ...
}
```

Тип параметра, оказывается, не `int`, а `float`. И размеры у них разные – `int` занимает 2 байта, а `float` – 4.

Обрабатывая код с вызовом функции, транслятор решит, что ей надо передать 2 байта со значением `int`. И тот же транслятор, обрабатывая код с определением функции, создаст код, который использует для параметра 4 байта – размер переменной `float`. В результате вызванная функция получит 2 байта от целого, и еще 2 байта «мусора», и попытается с этими 4-мя байтами работать, как с нормальным числом типа `float`.

Именно для того чтобы избегать подобных неприятностей, в языке С помимо определений сушествуют еще и объявления (и это относится не только к функциям).

Если определение – это полное описание чего-либо (в нашем случае функции, но это может быть и тип данных, и переменная), то объявление содержит лишь ту

информацию, которая необходима транслятору, чтобы избегать ошибок вроде показанной выше.

Чтобы избежать недоразумений при вызове функции `f()`, надо добавить всего лишь одну строку с ее объявлением перед функцией `main()`:

```
void f(float x);     // void f(float); - можно и так
void main() {
  ...
}
```

Такое объявление не содержит тела функции – сразу за списком параметров стоит точка с запятой. Но транслятору этого вполне достаточно – он теперь будет знать, что где-то (возможно, даже в другом файле или в библиотеке) есть такая функция, и он теперь сможет или преобразовать фактический параметр к правильному типу перед вызовом, или хотя бы (в сложных случаях) сообщить о том, что переменная, которая используется при вызове функции, не подходит для вызова.

Компилятор также обнаружит различия в количестве аргументов, использованных при вызове функции, и в количестве параметров функции.

Вот такое объявление функции и называется прототипом функции.

В общем виде ***прототип функции*** выглядит так:

```
тип имя_функции(тип [имя_пар1], ..., тип [имя_парN]);
```

Использование имен параметров не обязательно.

В качестве прототипа функции может также служить ее определение, если оно находится в том же файле, что и вызов функции, причем до первого вызова этой функции.

```
void f(float x) {
  ...
}
void main() {
  int i = 1;
  f(i);
  ...
}
```

Транслятор, уже зная о функции `f()`, поймет, что надо сначала привести значение `i` к типу `float`, а уж потом передавать его в функцию. Т.е. в этом примере специальный прототип не требуется, так как функция `f()` определена еще до того, как она начинает использоваться в `main()`.

Единственная функция, для которой не требуется прототип, – это `main()`, так как это первая функция, вызываемая в начале работы программы.

Поскольку прототипы содержат «неполную» информацию – у функций нет тела, под переменные не резервируется память, и так далее – их можно включать хоть в каждый файл программы, на ее размере и скорости работы это не скажется (лишь бы объявления в разных файлах соответствовали друг другу). И чтобы не пи-

сать, рискуя ошибиться, одно и то же многократно, такие объявления обычно собирают в заголовочный файл, а потом этот файл при необходимости включают в нужные cpp-файлы с помощью директивы #include. Этим мы уже неоднократно пользовались. Например, файл <stdio.h>, который мы включали в наши программы, среди прочего содержит прототипы функций printf() и scanf(), в то время как сами эти функции находятся совсем в другом месте – в библиотеке.

19. ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИИ

19.1. Способы передачи параметров в функции

Когда одна функция вызывает другую, обычный метод сообщения между ними состоит в использовании глобальных переменных, возвращаемых значения и параметров вызываемой функции.

В языках программирования имеется два основных способа передачи параметров подпрограмме. Первый из них – *передача по значению*. При его применении в формальный параметр подпрограммы копируется значение фактического параметра (аргумента). В таком случае изменения формального параметра на фактический аргумент не влияют.

Вторым способом передачи параметров подпрограмме является *передача по ссылке*. При его применении в формальный параметр копируется адрес фактического аргумента. Это значит, что, в отличие от передачи по значению, изменения значения формального параметра приводят к точно таким же изменениям значения фактического аргумента.

В языке С есть только один способ сопоставления фактических и формальных параметров – передача по значению (передачи параметров по ссылке есть в С++). В Паскале есть передача по значениюю и по ссылке. Бывают и другие методы (в Fortran – копирование-восстановление, в Algol – передача по имени).

Передача по значению представляет собой простейший способ передачи параметров. При этом происходит вычисление фактических параметров, и полученные значения передаются вызываемой процедуре.

Метод передачи по значению реализуется следующим способом:

8. формальный параметр рассматривается как локальная переменная, так что память для нее выделяется в записи активации вызываемой функции, т.е. в стеке;

9. вызывающая функция вычисляет фактические параметры и помещает их значения в память, выделенную для формальных параметров.

19.2. Передача параметров в функции в языке С

В языке С всегда аргументы при вызове функции передаются по значению, т.е. в стеке выделяется место для формальных параметров функции и в это выделенное место при ее вызове заносятся значения фактических аргументов. Затем функция их использует и может изменять эти значения в стеке. Но при выходе из функции измененные значения теряются. Вызванная функция не может изменить значения пе-

ременных, указанных как фактические аргументы при обращении к данной функ-
функции.

```
void f(int k) {
  k = -k;
}
void main() {
  int i = 1;
  f(i);
  printf("i = %d\n", i);  // результат: i = 1
}
```

Обратить внимание! Надо запомнить, что в функцию передается копия аргумента. То, что происходит внутри функции, не влияет на значение переменной, которая была использована при вызове в качестве аргумента. Кстати, именно благодаря этому, при вызове функции в качестве фактических аргументов можно указывать константы и выражения, а не только переменные.

19.3. Передача указателей в функции

А что делать, если функция должна изменить значение фактического параметра? Самый очевидный, но не самый лучший, способ – заменить такой параметр глобальной переменной. Минус – повышение шансов ошибиться из-за неучтенных побочных эффектов при вызове функций.

В случае необходимости функцию можно использовать для изменения передаваемых ей аргументов. В этом случае в качестве аргумента необходимо в вызываемую функцию передавать не значение аргумента, а значение его адреса, т.е. указатель. Так как функции передается адрес аргумента, то ее внутренний код в состоянии изменить значение этого аргумента,.

Указатель передается функции так, как и любой другой аргумент – по значению. Понятно, что при передаче адреса параметр следует объявлять как один из типов указателей.

Поскольку функция получает копию аргумента, она не сможет повлиять на сам указатель. Но она может записать все, что угодно туда, куда он направлен, используя для обращения к значению аргумента-оригинала операцию разыменования *.

Задача. Написать функцию для замены местами значений двух переменных и вызвать ее из функции main().

```
void swap(int *pa, int *pb) { // параметры-указатели
  int temp;
  temp = *pa;   // сохранить значение a
  *pa = *pb;    // занести b в a
  *pb = temp;   // занести a в b
}

void main(void) {
  int i = 10, j = 20;
  printf("i и j перед обменом значениями: %d %d\n", i, j);
```

```
  swap(&i, &j);       // передаем адреса переменных i и j
  printf("i и j после обмена значениями: %d %d\n", i, j);
}
```

Функция `swap()` может выполнять обмен значениями двух переменных, на которые указывают `pa` и `pb`, потому что в функцию передаются адреса переменных, а не их значения. Внутри функции, используя стандартные операции с указателями, можно получить доступ к содержимому переменных и провести обмен их значений.

Обратить внимание! В любую функцию, в которой используются параметры в виде указателей, необходимо при вызове передавать адреса аргументов, используя операцию взятия адреса &.

При вызове функции с аргументами-указателями не обязательно указывать в качестве параметра адрес переменной. Можно вместо этого передать значение указателя, в котором такой адрес содержится.

```
void main(void) {
  int i = 10, j = 20;
  int *pi = &i, *pj = &j;
  printf("i и j перед обменом значениями: %d %d\n", i, j);
  swap(pi, pj);       // передаем адреса переменных i и j
  printf("i и j после обмена значениями: %d %d\n", i, j);
}
```

Здесь мы работаем с указателями как с обычными переменными – засылаем в них значения с помощью оператора присваивания, а потом передаем функции.

Вывод: Если вызываемая функция используется для изменения переменных в вызывающей функции, то в качестве параметров ей надо передавать не сами нужные переменные, а либо их адреса, либо указатели на них.

Задача. Написать две функции для вычисления суммы двух отрицательных чисел и их вызов из функции main(). Исходные данные должны вводиться в функции main(). Первая функция должна возвращать заданную величину. Во второй функции обеспечить контроль правильности исходных данных. Функция, кроме вычисления заданной величины, должна возвращать признак правильности исходных данных.

```
int sum1(int a, int b) {
  return a+b;
}

int sum2(int a, int b, int *sum) {
  if (a >= 0 || b >= 0)
    return 0;     // признак неверных данных
  *sum = a + b;
  return 1;       // признак правильных данных
}

void main(void) {
  int x, y, s;
  scanf("%d %d", &x, &y);
  printf("Сумма 1 = %d\n", sum1(x,y));
```

```
  if (sum2(x,y,&s) == 1)
    printf("Сумма 2 = %d\n", s);
  else
    printf("Неверные данные!\n");
}
```

20. КЛАССЫ ХРАНЕНИЯ И ВИДИМОСТЬ ПЕРЕМЕННЫХ

20.1. Общие положения

Компилятор языка С для установления корректной связи идентификаторов с объектами в памяти требует, чтобы для каждого идентификатора обязательно были заданы два атрибута: тип и класс хранения. *Тип* определяет размер памяти, выделяемой для объекта, и способ интерпретации выделенной памяти (например, целое число, вещественное число, адрес памяти). *Класс хранения* определяет место в памяти, где объект располагается. Таких мест всего три: сегмент данных (постоянная память), стек и регистры процессора. Кроме места класс хранения определяет и время жизни объекта (например, все время выполнения программы или время выполнения отдельной функции). Класс хранения можно либо задать явно, либо компилятор сам определяет класс хранения по местоположению описания объекта в тексте программы.

С классом хранения связано понятие *блока* программы. В языке С блоком считается последовательность операторов, заключенная в фигурные скобки. Существуют два вида блоков – составной оператор и определение функции, состоящее из составного оператора, являющегося телом функции, и предшествующего телу заголовка функции. Блоки могут включать в себя составные операторы, но не определения функций (нельзя определять функции внутри других функций). Внутренние блоки называются вложенным.

Переменные внутри функций можно определять в блочно-структурной манере. Объявления переменных (вместе с инициализацией) разрешено помещать не только в начале функции, но и после любой фигурной скобки, открывающей составную инструкцию:

```
if (n > 0) {
  int i;   // описание новой переменной i
  ...
}
```

Обратить внимание! Лучше не пользоваться одними и теми же именами для разных переменных, поскольку слишком велика возможность путаницы и появления ошибок.

Время жизни – это интервал времени выполнения программы, в течение которого программный объект (переменная или функция) существует. Жизнь любого объекта начинается с момента определения этого объекта. С точки зрения времени жизни различают три типа объектов: глобальные, локальные и динамические:

1) Объект с глобальным временем жизни имеет распределенную для него память и определенное значение на протяжении всего времени выполнения программы.

2) Объект с локальным временем жизни имеет распределенную для него память и определенное значение только во время выполнения блока, в котором этот объект определен.

3) Объект с динамическим временем жизни имеет распределенную для него память и определенное значение с момента динамического выделения памяти под него в программе и до момента завершения программы или до момента уничтожения объекта в программе (динамическое выделение памяти будем рассматривать позднее).

Область определения (видимости) – это та часть программы, в которой может быть использован данный объект. Есть несколько типов области определения: 1) в пределах блока, 2) в пределах функции (это по сути тот же блок), 3) в пределах исходного файла, 4) во всех исходных файлах, образующих программу. Это зависит от того, на каком уровне объявлен объект: на внутреннем, т.е. внутри некоторого блока, или на внешнем, т.е. вне всех блоков.

20.2. Спецификаторы класса памяти

В языке С есть четыре спецификатора класса памяти: `auto`, `extern`, `static` и `register`. Эти спецификаторы сообщают компилятору, в каком месте он должен разместить соответствующие объекты в памяти. Как уже говорилось, таких мест всего три: сегмент данных (постоянная память), стек и регистры процессора.

Объекты классов `auto` и `register` имеют локальное время жизни. Переменные класса памяти `auto` располагаются в стеке. Переменные класса памяти `register` располагаются, если это возможно, в регистрах процессора, или же в стеке.

Спецификаторы `static` и `extern` объявляют объекты с глобальным временем жизни. Память под переменные с глобальным временем жизни выделяется в сегменте данных.

Далее нам опять понадобятся чрезвычайно важные понятия объявления и описания. *Объявление* (*декларация*) объявляет имя и тип объекта. *Описание (определение)* выделяет для объекта участок памяти, где он будет находиться. Один и тот же объект может быть объявлен неоднократно в разных местах, но описан он может быть только один раз. В большинстве случаев объявление переменной является в то же время и ее описанием.

Общая форма ***объявления*** переменных при этом такова:

```
спецификатор_класса_памяти тип имя_переменой;
```

Спецификатор класса памяти всегда должен стоять первым. Если класс памяти не указан, то он определяется по умолчанию в зависимости от контекста.

Для переменных на внутреннем уровне может быть использован любой из четырех спецификаторов класса памяти, а если он не указан, то подразумевается класс

памяти auto.

При объявлении переменных на глобальном уровне может быть использован спецификатор класса памяти static или extern. Классы памяти auto и register для глобального объявления недопустимы.

20.3. Область видимости функций

Все функции в языке С имеют глобальное время жизни и существуют в течение всего времени выполнения программы, т.е. функции всегда определяются глобально. Они могут быть объявлены с классом памяти static или extern.

Правила определения области видимости для функций отличаются от правил видимости для переменных:

1. Функция, объявленная как static, видима в пределах того файла, в котором она определена. Каждая функция может вызвать другую функцию с классом памяти static из своего исходного файла, но не может вызвать функцию, определенную с классом static в другом исходном файле. Разные функции с классом памяти static, имеющие одинаковые имена, могут быть определены в разных исходных файлах, и это не ведет к конфликту.

2. Функция, объявленная с классом памяти extern, видима в пределах всех исходных файлов программы. Чтобы в каком-либо файле сделать видимой функцию, определенную в другом файле, надо в начало этого файла поместить прототип нужной функции. Любая функция может вызывать функции с классом памяти extern.

3. Если в объявлении функции отсутствует спецификатор класса памяти, то по умолчанию принимается класс extern.

20.4. Глобальные переменные

Глобальные переменные – это переменные, которые определены за пределами любой функции. Если при описании глобальной переменной нет ее инициализации, то начальное значение такой переменной будет равно 0.

Объявление переменных на глобальном уровне – это или определение переменных, или ссылки на определения, сделанные в другом месте программы.

Переменная, объявленная глобально, видима в пределах остатка исходного файла, в котором она определена. Выше своего описания и в других исходных файлах эта переменная невидима (но ее можно сделать видимой, как будет показано ниже).

```
int x = 10;    // глобальная переменная  x = 10
int y;         // глобальная переменная  y = 0

void main () {
   ... y++; ...
}
void f1 () {
  x = 5;
  y = 2;
```

```
}
```

Здесь переменные x и y видны и в функции main(), и в функции f1(). Что значит: «функция видит переменную»? Это значит, что внутри функции можно обращаться к этой переменной.

```
void main () {
  ...
}

void f1 () {
  x = 15;        // компилятор выдаст ошибку, переменная ещё не объявлена
}

int x = 5;       // определение глобальной переменной

int getX () {
  return x;      // getX видит переменную x
}
```

Переменная x объявлена после функций main() и f1(), а значит, в этих функциях она не видна.

Глобальная переменная может быть определена только один раз в пределах своей области видимости, но объявлена – много раз.

Спецификатор класса памяти extern для глобальных переменных используется в качестве ссылки на переменную, определенную в другом месте программы, т.е. для расширения области видимости переменной. При таком объявлении область видимости переменной расширяется до конца исходного файла, в котором сделано объявление.

```
extern int x; // объявляем глобальную переменную, теперь
              // ее можно использовать ниже в этом файле

void main () {
  ...
}
void f1 () {
  x = 15;        // компилятор уже  не выдаст ошибку
}
int x = 5;       // определение глобальной переменной
int getX () {
  return x;      // getX видит переменную x
}
```

Объявление extern сообщает компилятору, что переменная x определена в другом месте, и память под нее выделять не требуется. Поэтому программа компилируется без ошибки, несмотря даже на то, что x используется до своего описания. Здесь спецификатор extern сообщает компилятору, что эта переменная будут определена в файле позже.

Спецификатор `extern` играет большую роль в программах, состоящих из многих файлов. В языке С программа может быть записана в нескольких файлах, которые компилируются раздельно, а затем компонуются в одно целое. В этом случае необходимо как-то сообщить всем файлам о глобальных переменных программы. Самый лучший способ сделать это – определить (описать) все глобальные переменные в одном файле и объявить их со спецификатором `extern` в остальных файлах. Таким образом компилятор узнает имена и типы переменных, размещенных в другом файле, и может отдельно компилировать все файлы.

На практике программисты обычно включают объявления `extern` в заголовочные файлы, которые просто подключаются к каждому файлу исходного текста программы. Это более легкий путь, который к тому же приводит к меньшему количеству ошибок, чем повторение этих объявлений вручную в каждом файле.

В объявлениях с классом памяти `extern` не допускается инициализация, так как эти объявления ссылаются на уже существующие и определенные ранее переменные.

Основное применение глобальных переменных – они нужны для того, чтобы разные функции могли обмениваться между собой информацией.

Разумеется, в языке С это не единственный способ обмена информацией между функциями. Но он часто используется, причем не только новичками, которым такой путь кажется самым простым. Его выгода – сокращение накладных расходов при вызове функций и, следовательно, ускорение работы программы. Расплата за него – больший риск сделать ошибку из-за неизвестного или просто забытого побочного эффекта от вызова функции (случайное изменение глобальной переменой в функции, случайное определение другой переменной с таким же именем внутри функции, и тогда глобальная переменная не будет видна в этой функции).

20.5. Глобальные статические переменные

Спецификатор `static` в определении глобальной переменной заставляет компилятор создать глобальную переменную, видимую только в том файле, в котором она объявлена. Статическая глобальная переменная, таким образом, подвергается внутреннему связыванию. Это значит, что хоть эта переменная и глобальная, тем не менее процедуры в других файлах не увидят ее и не смогут случайно изменить ее значение. Этим снижается риск нежелательных побочных эффектов.

Глобальная переменная может быть определена только один раз в пределах своей области видимости. В другом исходном файле может быть объявлена другая глобальная переменная с таким же именем и с классом памяти `static`, конфликта при этом не возникает, так как каждая из этих переменных будет видимой только в своем исходном файле.

20.6. Локальные переменные

Переменные, определенные внутри блока (в том числе и функции), являются локальными и имеют по умолчанию класс памяти `auto`. Переменная с классом памяти `auto` имеет локальное время жизни и видна только в блоке, в котором объявлена (точнее, от места определения до конца блока). Память для такой переменной

выделяется в стеке при входе в блок и освобождается при выходе из блока. При повторном входе в блок этой переменной может быть выделен другой участок памяти. Таким образом, локальная переменная, например, не может сохранять свое значение в промежутках между вызовами функции.

```
void f1 () {
  int a = 0;
  a = a + 1;
}
```

В данном примере, сколько бы раз не вызывалась функция, переменная a никогда не станет больше единицы.

Переменная с классом памяти auto автоматически не инициализируется. Она может быть проинициализирована явно при определении путем присвоения ей начального значения. Значение неинициализированной переменной с классом памяти auto считается неопределенным (точнее, случайным). При входе в блок под нее просто отводится память, прямо с тем содержимым, которое в ней было. В результате в неинициализированных локальных переменных оказывается «мусор» – непредсказуемые значения, которые к тому же могут меняться от вызова к вызову функции, поскольку зависят от предистории работы программы.

Формальные параметры функции находятся в ее области действия. Это значит, что параметры доступны внутри всей функции. Параметры создаются в начале выполнения функции и уничтожаются при выходе из нее. По своей сути формальные параметры являются локальными переменными функции и тоже располагаются в стеке.

```
void f1 () {
  int a;        // объявление локальной переменной
  f2(100);
}
void f2 (int b) {
  b = 10;
  a = 3;        // ошибка, отсутствует объявление переменной
}
```

Описание локальных переменных возможно не только в начале, но и в произвольном месте в функции. Тогда эту переменную можно использовать от момента определения до конца функции.

Также можно объявить локальную переменную в блоке: в начале блока или в середине. Переменная, объявленная в блоке, «прячет» любую другую переменную с таким же именем, описанную вовне.

```
void main() {
  int x = 10;
  int a = 5, b = 7;
  if (a<b) {                  // начало вложенного блока
    int y;                    // локальная для блока переменная
    y = x+7;
```

```
    printf("x = %d\n", x);  // x = 10
    printf("y = %d\n", y);  // y = 17
    int x;                  // описание новой переменной внутри блока
    x = 20;
    y = x+7;
    printf("x = %d\n", x);  // x = 20
    printf("y = %d\n", y);  // y = 27
  }
  printf("x = %d\n", x);    // x = 10
  // printf("y = %d\n", y); // ошибка – неопределенная переменная y
}
```

Переменная, объявленная локально с классом памяти `extern`, является ссылкой на переменную с тем же самым именем, определенную глобально в одном из исходных файлов программы. Цель такого объявления состоит в том, чтобы сделать определение переменной глобального уровня из другого файла видимым внутри блока. Также объявление с классом памяти `extern` требуется при необходимости использовать глобальную переменную, описанную в текущем исходном файле, но ниже по тексту программы, т.е. до выполнения ее глобального определения.

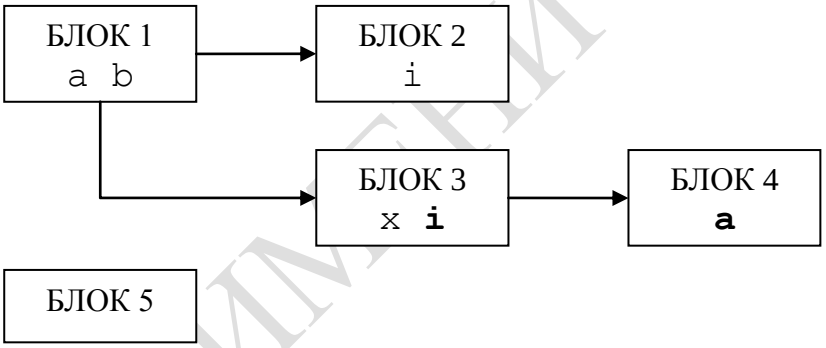
```
void main () {
  ...
}
void f1 () {
  extern int x;// объявляем глобальную переменную, теперь ее можно
               // использовать в этой функции, иначе компилятор выдал бы
  x = 15;      // ошибку, что переменная ещё не определена
}
int x = 5;     // определение глобальной переменной
int getX () {
  return x;    // getX видит переменную x
}
```

Обратить внимание! *В область видимости блока (в том числе и функции) входят* все глобальные переменные, объявленные до начала блока, а также локальные переменные объявленные в этом блоке и локальные переменные, объявленные во всех охватывающих блоках (другими словами, локальная переменная видна в блоке, в котором описана, в также во всех вложенных блоках при условии, что во вложенном блоке нет своей локальной переменной с таким же именем (если есть, то такая переменная делает недоступной переменную с таким же именем из внешнего блока, в том числе и глобальную)).

```
int getZ(void); // описание прототипа функции
int x = 5;      // определение глобальной переменной

void main () {                  // БЛОК 1
  extern int y;          // объявляем глобальную переменную y
  int a, b;              // описываем локальные a и b
```

```
  scanf("%d%d", &a, &b);
  if (a < b) {                        // БЛОК 2
    extern int z;          // объявляем глобальную переменную z
    int i = x + y + z;     // описываем локальную i
    printf("i = %d\n", i);            // при вводе 1 и 2 получим 8
  }
  else {                              // БЛОК 3
    int x = 0;             // перекрывает глобальную x локальной x
    int i;                 // это совсем другая локальная i, а не из блока 2
    for (i=0; i<5; i++) {             // БЛОК 4
      int a = getZ()*i; // перекрывает переменную a из main()
      x += a;
    }
    printf("x = %d\n", x);            // при вводе 2 и 1 получим 20
  }
}

int y = 1;    // определение глобальной переменной
int z = 2;    // определение глобальной переменной

int getZ () {                         // БЛОК 5
  return z;
}
```

Вложенность блоков и локальные переменные блоков:



БЛОК	*Доступные локальные переменные*	*Доступные глобальные переменные*
1	a b	x y
2	a b i	x y z
3	a b x **i**	y
4	b x **i a**	y
5		x y z

20.7. Статические локальные переменные

Локальные переменные, объявленные со спецификатором класса памяти `static`, обеспечивают возможность сохранить значение переменной при выходе из блока и использовать его при повторном входе в блок. Такая переменная имеет глобальное время жизни и область видимости внутри блока, в котором она объявлена.

В отличие от переменных с классом `auto`, память для которых выделяется в стеке, для переменных с классом `static` память выделяется в сегменте данных, и поэтому их значение сохраняется при выходе из блока. Для локальной переменной, описанной со спецификатором `static`, компилятор выделяет память точно так же, как и для глобальных переменных – в начале работы программы. Коренное отличие статических локальных от глобальных переменных заключается в том, что статические локальные переменные видны только внутри блока, в котором они объявлены. Говоря коротко, статические локальные переменные – это локальные переменные, сохраняющие свое значение между вызовами функции.

Статическую локальную переменную можно инициализировать. Это значение присваивается ей только один раз – в начале работы всей программы, но не при каждом входе в блок программы, как обычной локальной переменной. Если при описании статической локальной переменной нет ее инициализации, то начальное значение такой переменной будет равно 0.

```
int f() {
  static int a = 0;  // по умолчанию тоже равно 0
  a = a + 10;
  return a;
}

void main () {
  printf("Вызов функции 1: a = %d", f());  // a = 10
  printf("Вызов функции 2: a = %d", f());  // a = 20
  printf("Вызов функции 3: a = %d", f());  // a = 30
}
```

Подобные переменные крайне полезны и необходимы. Но это не значит, что надо добавлять модификатор `static` ко всем локальным переменным. Вы этим только добавите себе проблем. Программа не просто будет работать медленнее, возможно, что она будет работать вовсе не так, как вы ожидаете.

Для чего все-таки такие переменные придуманы? Эти переменные создаются один раз за время работы программы, и один раз инициализируются – либо 0, либо тем значением, которое вы задали. А раз они живут независимо от функции, значит в одном вызове функции в такую переменную можно что-то положить, а в следующем – это что-то использовать.

```
int f() {
  static int ncalls = 1;
  // который раз функция вызывается?
  printf("number of calls %d\n", ncalls++);
  ...
}
```

Такая функция помимо другой работы будет сообщать нам, который раз она вызвана. В первый раз напечатает 1, потом 2, и так далее. Подобная информация бывает порой очень полезной при отладке программы.

Другое полезное использование статических локальных переменных – возможность выполнять какие-то подготовительные операции только один раз.

```
void f() {
  static int first = 0;
  if (first == 0) {
    first = 1;
    ...
  }
  ...
}

void main() {
  int i;
  for (i=0; i<1000; i++)
    f();
}
```

Очень полезны бывают статические локальные переменные при работе со строками, но про это мы поговорим чуть позже.

20.8. Регистровые переменные

Спецификатор класса памяти register предписывает компилятору выделить, если это возможно, память для переменной в регистре процессора (есть такой специальный вид памяти), а не в оперативной памяти. Это приводит к тому, что операции с переменной `register` осуществляются намного быстрее, чем с обычными переменными, т.к. такая переменная уже находится в процессоре и не нужно тратить время на выборку ее значения из оперативной памяти (и на запись в память).

Переменная, объявленная с классом памяти `register`, имеет ту же область видимости, что и переменная `auto`. Число регистров, которые можно использовать для значений переменных, ограничено возможностями компьютера, и в том случае, если компилятор не имеет в распоряжении свободных регистров, то переменной выделяется память как для класса `auto`, но компилятор получает указание позаботиться о быстродействии операций с ними. В языке С с помощью оператора `&` нельзя получить адрес регистровой переменной, потому что она может храниться в регистре процессора, который не имеет адреса.

Переменные `register` идеально подходят для оптимизации скорости работы цикла. Как правило, переменные `register` используются там, где от них больше всего пользы, а именно, когда программа многократно обращается к одной и той же переменной.

20.9. Выводы

Место, где выделяется память для переменных, определяется по следующим правилам:

1. Глобальные и статические переменные (класс памяти `extern` и `static`) размещаются в постоянной памяти (сегменте данных).

2. Обычные локальные переменные (класс памяти `auto`), а также формальные параметры функций размещаются в стеке.

3. Регистровые переменные (класс памяти `register`) размещаются, если это возможно в регистрах процессора, иначе – в стеке.

Время жизни переменной определяется по следующим правилам:

1. Переменная, объявленная глобально (т.е. вне всех блоков), существует на протяжении всего времени выполнения программы.

2. Локальные переменные (т.е. объявленные внутри блока) с классом памяти `register` или `auto`, имеют время жизни только на период выполнения того блока, в котором они объявлены. Если локальная переменная объявлена с классом памяти `static` или `extern`, то она имеет время жизни на период выполнения всей программы.

Видимость переменных и функций в программе определяется следующими правилами:

1. Переменная, объявленная или определенная глобально, видима от точки объявления или определения до конца исходного файла. Можно сделать переменную видимой и в других исходных файлах, для чего в этих файлах следует ее объявить с классом памяти `extern`.

2. Переменная, объявленная или определенная локально, видима от точки объявления или определения до конца текущего блока.

3. Переменные из объемлющих блоков, включая переменные объявленные на глобальном уровне, видимы во внутренних блоках. Эту видимость называют вложенной. Если переменная, объявленная внутри блока, имеет то же имя, что и переменная, объявленная в объемлющем блоке или глобальная переменная, то это разные переменные, и переменная из объемлющего блока или глобальная переменная во внутреннем блоке будет невидимой.

21. ОРГАНИЗАЦИЯ ПАМЯТИ ПРОГРАММЫ

Блок памяти для запуска программы выделяется операционной системой. Скомпилированная программа С имеет четыре области памяти:

1) код программы (сегмент кода) – содержит машинные коды функций программы. Функции, присоединенные к exe-файлу на стадии линковки, размещаются вне области кода;

2) постоянные объекты данных программы (сегмент данных);

3) стек, который используется при вызове функций;

4) куча – это такая свободная область памяти, для получения участков памяти из которой программа вызывает функции динамического распределения памяти.

Код
Статические (постоянные) данные
Куча
↓

↑
Стек

Размер кода фиксируется во время компиляции, так что компилятор может разметить его в статически определенной области. Аналогично во время компиляции становится известным размер некоторых объектов данных (глобальные и статические переменные, константы), и они также могут быть размещены в статически определяемой области.

Язык С использует стек для работы с записями активаций функций или, другими словами стек используется при вызове функций.

Отдельная область памяти времени исполнения, называемая кучей (`heap`), предназначена для обеспечения запросов программы на динамическую память (будем рассматривать позднее). В кучу данные помещаются только по указанию программиста и не имеют имени. К ним можно обратиться только по адресу, расположенному в указателе.

Размеры стека и кучи могут изменяться в процессе работы программы (стек и куча растут навстречу друг другу).

Информация, необходимая для однократного выполнения функции, поддерживается с помощью непрерывного блока памяти, называемого записью активации. Этот блок состоит из набора полей. В языке С запись активации функции размещается в стеке при вызове функции и удаляется из стека при передаче управления вызывающей функции.

Запись активации функции состоит из следующих основных частей:

Возвращаемое значение
Формальные параметры
Сохраненное состояние компьютера
Локальные данные

Назначения полей:

1) Поле для локальных данных хранит данные, являющиеся локальными для функции. Это все локальные переменные, определенные внутри функции.

2) Поле сохраненного состояния компьютера хранит информацию о состоянии компьютера непосредственно перед вызовом функции. Эта информация включает значения счетчика программы (указатель выполняющейся инструкции) и регистров компьютера, которые должны быть восстановлены при возврате управления из функции. Данная информация, после восстановления из стека, позволяет продолжить выполнение программы после вызова функции.

3) Поле для формальных параметров используется вызывающей функцией для передачи параметров вызываемой функции.

4) Поле возвращаемого значения используется вызываемой функцией для возврата значения вызывающей функции. Возвращаемое значение функции может храниться и в регистре. Если его размер слишком велик для размещения в регистре, то оно размещается в стеке, а значение в регистре будет указывать на него.

В языке С используется так называемое стековое распределение памяти. При каждом вызове функции память для локальных переменных содержится в записи активации для этого вызова. Таким образом, локальные переменные при каждом вызове связываются с новой областью памяти, поскольку при вызове в стек вносится новая запись активации. Более того, значения локальных переменных уничтожаются по окончании активации, т.е. значения локальных переменных становятся недоступны, так как выделенная им память освобождается при удалении записи активации из стека.

Все локальные переменные (класс `auto`) размещаются в памяти в стеке. Все глобальные и статические переменные размещаются в постоянной памяти в области статических данных (сегмент данных). Регистровые переменные по мере возможности размещаются в регистрах процессора.

Рассмотрим организацию памяти на примере:

```
int k;
int sumk(int a, int b) {
  static int kol = 0;
  int s;
  kol++;
  printf("Вызов функции номер %d", kol);
  s = a+b;
  return k*s;
}
int min(int a, int b) {
  if (b<a) a = b;
  return a;
}
void main() {
  int x, y, z, t = 50;
  scanf("%d %d", &x, &y);
  scanf("%d", &k);
  z = sum(x,y);
  printf("sum * %d = %d", k, z);
  printf("min = %d", min(x,y));
  k = 2;
  printf("sum * 2 = %d", sum(10,t));
}
```

код	код	код
глобальная int k = 0 = 5 статическая int kol = 0 = 1 все константы (символьные и числовые)	глобальная int k = 0 = 5 статическая int kol = 0 = 1 все константы (символьные и числовые)	глобальная int k = 0 = 5 = 2 статическая int kol = 0 = 1 = 2 все константы (символьные и числовые)
вход в main()	**вход в main()**	**вход в main()**
не используется (return void)	не используется (return void)	не используется (return void)
не используется (нет параметров)	не используется (нет параметров)	не используется (нет параметров)

int x = мусор = 7	int x = мусор = 7	int x = мусор = 7
int y = мусор = 2	int y = мусор = 2	int y = мусор = 2
int z = мусор = 70	int z = мусор = 70	int z = мусор = 70
int t = 50	int t = 50	int t = 50
вход в sum()	**вход в min()**	**вход в sum()**
возврат = 70	возврат = 2	возврат = 120
параметр int a = 7	параметр int a = 7 = 2	параметр int a = 10
параметр int b = 2	параметр int b = 2	параметр int b = 50
int s = мусор = 14		int s = мусор = 60

22. МНОГОФАЙЛОВАЯ КОМПИЛЯЦИЯ (ПРОЕКТЫ)

Любую программу, текст которой занимает даже сотни строк, можно хранить в одном файле. Но это очень не удобно. Даже если допустить, что размер файла с программой может быть любого размера, как модифицировать и отлаживать программу, размер которой 1000 строк, а на экране видно только 20 строк? Кроме этого, при увеличении длины программы значительно увеличивается и время ее компиляции. Когда размер программы достигает некоторого предела, приходится разбивать ее на части, каждая из которых хранится отдельно – в отдельном файле.

При работе с большими программами намного удобнее размещать части программы не в одном, а в нескольких файлах. Это же можно сказать и про случай, когда одну программу разрабатывают несколько человек. Каждый файл должен включать целиком одну или несколько функций.

Преимущества такого подхода:

- разработку и отладку разных частей программы можно поручить разным людям;
- программу проще и удобнее отлаживать и модифицировать;
- ускорение в работе, ведь при запуске программы можно компилировать только те файлы, которые изменялись, для остальных можно использовать уже существующие `obj`-файлы (т.к. файлов может быть много, а процесс компиляции в С достаточно длинный, то это дает значительный выигрыш по времени в процессе работы). Эта последовательность действий (создание сначала *объектных* файлов, а затем самой программы) типична для многих компиляторов и сильно экономит время при разработке больших программ, потому что обрабатываются только *измененные* исходные тексты. Затем объектные файлы поступают редактору связей, и на выходе получается готовая программа.

Есть два подхода к многофайловой разработке программ на языке С.

Первый подход: нужные `cpp`-файлы подключать по `#include "*.cpp"` к главному `cpp`-файлу. Но это плохой подход, т.к. он не дает третьего преимущества – просто препроцессор создает один большой файл, который весь компилируется.

Второй подход: для объединений нескольких `cpp`-файлов в одну программу использовать проекты. Такой подход используется во многих интегрированных сре-

дах разработки (IDE *Integrated development environment* – система программных средств, используемая программистами для разработки программ. Обычно среда разработки включает в себя: текстовый редактор, компилятор и/или интерпретатор, средства автоматизации сборки, отладчик). Это подход, который позволяет использовать все преимущества многофайловой разработки программ.

Проект – это специальный файл, в который записываются имена файлов (в нашем случае `cpp`-файлов), которые IDE (в нашем случае BC++) следует объединять в один исполняемый `exe`-файл. Имя исполняемого файла совпадает в таком случае с именем проекта, а не с именем `cpp`-файла, как получается при однофайловой программе.

Все необходимые для работы с файлами проектов команды включены в меню `Project`.

Для организации файла проекта необходимо открыть файл проекта. Для этого выполняются команды `Project→Open Project…`. BC++ активизирует специальное окно «Project» в нижней части экрана и открывает окно диалога, позволяющее загрузить нужный файл проекта или создать новый с заданным именем.

Если создается новый файл проекта, окно «Project» первоначально будет пустым. Включение файлов в проект и их удаление выполняются либо через команды `Project→Add item…` и `Project→Delete item`, либо нажатием клавиш `Ins` и `Del`, в случае если курсор размещен в окне «Project». При добавлении файлов в проект открывается окно диалога, позволяющее выбрать нужный файл.

В проект можно включать не только `cpp`-файлы, но и `obj`-файлы и библиотеки (`.lib`). Не включаются в проект заголовочные `h`-файлы.

Сохранение файла проекта: `Options→Save…`. Надо убрать два верхних крестика, оставить только сохранение проекта.

Открыть существующий файл проекта можно и так: `bc имя[.prj]`.

Окно «Project» упрощает переход от одного файла, включенного в проект, к другому при их редактировании. Для этого надо выделить строку с нужным именем файла в окне «Project» и нажать клавишу ENTER.

При работе с проектом возникает необходимость взаимодействия отдельных файлов. Другими словами, функция из одного файла, должна иметь доступ к переменной или функции из другого файла.

```
// === файл func.cpp ================
extern int k;   // в другом файле определена эта переменнная
int sumk(int a, int b) {
  return k*(a+b);
}
int min(int a, int b) {
  return a<b ? a : b;
}
// === файл task.cpp ================
```

```
int k;                // глобальная переменная
int min(int, int);    // прототипы функций, определенных
int sum(int, int);    // в другом файле
void main() {
  int x, y, z;
  scanf("%d %d", &x, &y);
  scanf("%d", &k);
  printf("min = %d", min(x,y));
  c = sum(x,y);
  printf("sum * %d = %d", k, c);
}
```

В файле `func.cpp` надо использовать глобальную переменную, которая определена в файле `task.cpp`. Т.к. каждый файл проекта компилируется по отдельности, то при компиляции `func.cpp` будет ошибка компиляции «Переменная k не определена». При компиляции же файла `task.cpp` выдаст ошибку, что функции `min()` и `sum()` не имеют прототипов. Причина ошибок понятна: в одном файле ничего не известно про глобальную переменную `k`, в другом файле – про функции `min()` и `sum()`.

В файле `func.cpp` нужно как-то указать, что переменная `k` существует и определена в другом файле программы. Делается это с помощью указания `extern`. Строка `extern int k;` говорит компилятору, работающему с файлом `func.cpp`: «Переменная k определена в другом файле. В каком – не важно. Достаточно знать, что она – типа int». Увидев слово `extern`, компилятор создает объектный файл, в котором сказано, что переменная `k` – внешняя, определена в каком-то другом файле, и ее поисками займется уже редактор связей при внешнем связывании.

Нужно четко понимать, что жизнь дается переменной *один* раз. В нашем случае это делается в файле `task.cpp`, где объявление `int k;` велит компилятору выделить участок памяти для целочисленной переменной и заслать туда ее начальное значение 0. Слово `extern` ничего не создает и памяти никакой не выделяет.

Для того, чтобы указать компилятору, что какая-либо функция определена в программе, достаточно в нужном файле просто указать прототип нужной функции: `int min(int,int); int sum(int,int);`. Т.е. при многофайловой компиляции действуют обычные правила объявления функции, если функция не определена до момента своего использования.

Словом `extern` необходимо помечать только *переменные*. Для функций это можно не делать, потому что компилятор и так считает все функции внешними. Но чтобы указать компилятору, что функция определена в другом файле, ее прототип также можно предварить словом `extern`: `extern int min(int,int);`

23. ПЕРЕДАЧА В ФУНКЦИИ МАССИВОВ

23.1. Передача одномерных массивов в функции

При передаче имени массива в качестве параметра функции как аргумент передается не копия самого массива (это заняло бы слишком много места в стеке), а копия адреса нулевого элемента этого массива (или указатель на начало массива). Иными словами, массивы отличаются от других типов тем, что они не передаются и не могут передаваться по значению.

В программе ниже в `func()` передается указатель на массив `mas`:

```
void main(void) {
  int mas[10];
  ...
  func(mas);
  ...
}
```

Если в функцию передается указатель на одномерный массив, то в самой функции его можно объявить одним из трех вариантов: как указатель, как массив определенного размера и как массив неопределенного размера.

```
void func(int *x) {     // указатель
  ...
}
void func(int x[10]) { // массив определенного размера
  ...
}
void func(int x[]) {    // массив неопределенного размера
  ...
}
```

Эти три объявления тождественны, потому что каждое из них сообщает компилятору одно и то же: в функцию будет передан указатель на переменную целого типа.

В последнем примере форма объявления массива сообщает компилятору, что в функцию будет передан массив неопределенной длины. Длина массива не имеет для функции никакого значения, потому что в языке С проверка границ массива не выполняется. Эту функцию можно объявить даже так:

```
void func(int x[50]) {
  ...
}
```

Программа все равно будет работать правильно, потому что компилятор не создает массив из 50 элементов, а только подготавливает функцию к приему указателя.

Обратить внимание! Поскольку массив передается как указатель на его начало, то размер массива в объявлении аргумента можно не указывать. Это позволяет одной функцией обрабатывать массивы разной длины.

Раз функции передается адрес массива, значит функция работает с реальным содержимым этого массива и вполне может изменить это содержимое.

```
void f1(int m) {
  m++;
}
void f2(int m[]) {
  m[0]++;
}
void main() {
  int mas[2] = {1, 1};
  f1(mas[0]);
  printf("%d\n", mas[0]);   // mas[0] осталось равно 1
  f2(mas);
  printf("%d\n", mas[0]);   // mas[0] стало равно 2
}
```

В `f1()` в качестве аргумента передается копия элемента `mas[0]`. Изменение этой копии не приводит к изменению самого массива, т.к. аргумент `m` является локальной переменной в функции `f1()`. В функции `f2()` локальным является адрес массива `mas`, но не сам массив. Поэтому `m[0]++` изменяет сам массив `mas` (в то же время, например, `m++` внутри `f2()` изменило бы лишь локальный указатель `m`, но не адрес массива `mas`).

Для обработки массива, кроме адреса начала массива функция должна знать и количество элементов в массиве. Поэтому, если массив, для обработки которого используется функция, может иметь разный размер, в функцию дополнительно следует передавать и реальную размерность массива (точное число элементов в массиве). Тогда она сможет менять указатель в безопасных пределах и получит полный доступ к любому элементу массива.

Задача. *Разработать функцию для нахождения максимального элемента в одномерном массиве целых чисел.*

```
int f_max(int *m, int n) {
  int max, i;
  max = m[0];
  for (i=1; i<n; i++)
    if (m[i] > max) max = m[i];
  return max;
}
void main() {
  int mas[100], n, i, n2, k, *p, max2;
  printf("Введите размерность массива\n");
  scanf("%d", &n);
  printf("Введите массив\n");
  for (i=0; i<n; i++)
    scanf("%d", &mas[i]);
```

```
  printf("Максимальный элемент = %d\n", f_max(mas,n));
  // находим максимум во второй половине массива
  k = n2 = n / 2;  // k – смещение второй половины
  if (n&1) n2++;   // n2 – количество элементов во второй половине
                   // при нечетном n средний элемент относим ко второй половине
  printf("Максимальный элемент второй половины = %d\n",
         f_max(mas+k,n2));
  // можно и так вызвать функцию: p = mas+k; max2 = f_max(p,n2);
}
```

Задача. Разработать функцию для нахождения номера минимального элемента и суммы элементов одномерного массива целых чисел.

```
int func(int *m, int n, int *sum) {
  int i, min, n_min;
  min = *m; n_min = 0;
  *sum = m[0];
  for (i=1; i<n; i++) {
    if (m[i] < min) {
      min = *(m+i); n_min = i;    // находим номер минимального
    }
    *sum += m[i];  // находим сумму элементов
  }
  return n_min;    // возвращаем номер минимального элемента
}

void main() {
  int mas[100], n, i, sum;
  printf("Введите размерность массива\n");
  scanf("%d", &n);
  printf("Введите массив\n");
  for (i=0; i<n; i++)
    scanf("%d", &mas[i]);
  printf("Номер минимального = %d\n", func(mas, n, &sum));
  printf("Сумма = %d\n", sum);
}
```

Задача. Разработать функцию для замены всех нулевых элементов массива заданным числом.

```
void f(int *m, int n, int k) {
  int i;
  for (i=0; i<n; i++)
    if (m[i] == 0)
      *(m+i) = 100;        // m[i] = 100;
}

void main() {
  int mas[100], n, i;
```

```
  printf("Введите размерность массива\n");
  scanf("%d", &n);
  printf("Введите массив\n");
  for (i=0; i<n; i++)
    scanf("%d", &mas[i]);
  f(mas, n, 100);          // заменить все 0 на 100
  printf("Массив после замены\n");
  for (i=0; i<n; i++)
    printf("%d ", mas[i]);
  printf("\n");
}
```

23.2. Передача двумерных массивов в функции

Вместо одномерного массива функции передается указатель на его первый элемент. Точно так же не передается функции и двумерный массив. Массив массивов (т.е. двумерный массив в языке С) при передаче в функцию превращается в указатель на массив, а не в указатель на указатель. В функцию передается *указатель на массив,* число элементов которого равно длине строки двумерного массива.

Следующее описание является неверным:

```
func(int **mas) {      // ошибка!!!
  ...
}
```

Если в функцию передается двумерный массив:

```
int mas[N][M];
func(mas);
```

описание функции должно соответствовать

```
func(int mas[][M]) {
  ...
}
```

или

```
func(int (*pmas)[M]) {    // pmas - указатель на массив из M элементов
  ...
}
```

Скобки `(*pmas)` в объявлении обязательны, т.к. `int *pmas[M]` – это *массив указателей на `int`, а не указатель на массив*.

Обратить внимание! Так как функция не выделяет место для массива, нет необходимости знать его точный размер. Поэтому количество строк N может быть опущено. В то же время количество элементов в строке (количество столбцов M) опускать нельзя, т.к. «форма» массива (количество столбцов) используется для вычисления смещения элемента относительно начала массива. Поэтому количество столбцов (вторую размерность M) надо указывать обязательно.

<u>Задача.</u> Разработать функции для печати массива из 10 столбцов и замены знака у всех элементов массива с 5-ю столбцами.

```
void print10(int mas[][10], int n) {
  int i, j;
  for (i=0; i<n; i++) {
    for (j=0; j<10; j++)
      printf("%5d ", mas[i][j]);
    printf("\n");
  }
}
void negative5(int (*mas)[5], int n, int m) {
  int i, j;
  for (i=0; i<n; i++)
    for (j=0; j<m; j++)
      mas[i][j] = - mas[i][j];
}
void main(void) {
  int a[10][10],
      b[10][5] = {{1, 2, 3, 4},
                  {5, 6, 7, 8},
                  {9, 9, 9, 9}};
  int i, j, n=5;
  for (i=0; i<n; i++)
    for (j=0; j<10; j++)
      scanf("%d", &a[i][j]);
  print10(a,n);
  negative5(b,3,4);
  for (i=0; i<3; i++) {
    for (j=0; j<4; j++)
      printf("%5d ", b[i][j]);
    printf("\n");
  }
}
```

<u>Обратить внимание!</u> У двумерных (многомерных) массивов есть один неприятный недостаток при передаче их в функцию – для обработки массива с пятью столбцами нужно писать одну функцию, а для массива с десятью столбцами – другую!

При передаче функции двумерного массива как фактического параметра необходимо обязательно указывать количество столбцов. Однако, если работа с двумерным массивом не требует доступа к элементам посредством конкретного указания строк и столбцов, то функция может воспринимать многомерный массив как одномерный, хотя в качестве фактического параметра ей передается многомерный массив.

```
long sum(int m[], int n) {
```

```
  long s = 0;
  int i;
  for (i = 0; i < n; i++)
    s += m[i];              // s += *(m+i);
  return s;
}

void main(void) {
  int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int b[2][10] = {{ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10},
                  {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}};
  int c[5][10] = {{ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10},
                  {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
                  {21, 22, 23, 24, 25, 26, 27, 28, 29, 30}};
  printf("Сумма массива a = %d\n", sum(a, 10));
  printf("Сумма массива b = %d\n", sum((int *)b, 20));
  printf("Сумма массива c = %d\n", sum((int *)c, 30));
}
```

Функция `sum()` работает с двумерными массивами так же, как и с одномерными массивами. Так работать с двумерным массивом позволяет способ хранения в памяти многомерных массивов в языке С. Правда, есть одно ограничение – реальное количество элементов в строке такого массива должно строго совпадать со второй размерностью массива, а вот количество строк в таком массиве может быть любым.

23.3. Передача в функции символьных строк

В языке С строка – это одномерный массив символов. Передача символьной строки в функцию подобна передаче любого массива в качестве параметра. Поэтому, как и для любого одномерного массива, реально в функцию передается адрес строки. Но, в отличие от обычного одномерного массива, для строк нет необходимости передавать в функцию их реальный размер, так как все строки оканчиваются нулевым символом, и их размер в функции легко вычислить.

Задача. *Разработать функцию для вычисления длины строки.*

```
int lenStr(char *s) {   // int lenStr(char s[]) {
  int n;
  for (n=0; s[n]; n++);
  return n;
}

void main(void) {
  char str[80];
  gets(str);
  printf("Длина строки = %d\n", lenStr(str));
}
```

Задача. *Разработать функцию, которая возвращает указатель на первое вхождение символа в строку.*

```
char *func(char c, char *s) {
  for (; *s; s++)
    if (c == *s) return s;
  return NULL;
}
void main(void) {
  char ss[80], *p, ch;
  gets(ss);
  ch = getchar();
  p = func(ch, ss);
  if(p)
    printf("Символ найден: %s\n", p);
  else
    printf("Символ не найден\n");
}
```

Если заданного символа в строке нет, то функция возвращается указатель NULL, что позволяет проверить, было ли вхождение нужного символа или нет.

23.4. Возвращение указателей из функций

Функции, которые возвращают указатели, – это обычные функции. И работают с ними тоже обычным способом. Но есть несколько тонкостей при разработке таких функций.

Во-первых, указатели являются адресами в памяти. Поэтому в объявлении функции, которая возвращает указатель, тип возвращаемого указателя должен декларироваться явно. Например, нельзя объявлять возвращаемый тип как `int *`, если возвращается указатель типа `char *`. Иногда (правда, крайне редко) требуется, чтобы функция возвращала «универсальный» указатель, т.е. указатель, который может указывать на данные любого типа. Тогда тип результата функции следует определять как `void *`.

Во-вторых, такие функции возвращают адрес в памяти. Поэтому надо очень внимательно следить за тем, чтобы объект, адрес которого возвращает функция, существовал не только во время работы функции, но и после выхода из нее. Чаще всего это касается функций, которые возвращают `char *`.

Есть три подхода к решению проблемы существования объекта, адрес которого возвращает функция:

1) делать такие объекты глобальными. Это не очень хорошее решение, т.к. требуется помнить об описании нужных глобальных переменных;

2) описывать такой объект в вызывающей функции и передавать его в вызываемую функцию. Обычное решение для функций, которые возвращают `char *.`;

3) описывать такие объекты в функции как `static`. Нормальное решение, ведь статические переменные существуют в течение всего времени выполнения программы и, значит, их адрес можно без проблем возвращать из функций.

Ниже приведены два примера правильных функций, которые возвращают ука-

затели на строки.

Задача. *Разработать функцию для удаления в строке заданного символа.*

```
char *delChar(char *s, char c) {
  char ss[80];
  int i = 0, j = 0;
  while (s[i]) {
    if (s[i] != c)
      ss[j++] = s[i];
    i++;
  }
  ss[j]=0;
  for (i=0; i<=j; i++)
    s[i] = ss[i];
  return s;          // return ss; - ошибка!!!
}
void main(void) {
  char str[80];
  gets(str);
  puts(delChar(str,'a'));
}
```

Задача. *Разработать функцию для сцепления двух строк.*

```
char *strcat(char *s, char *ss) {
  static char str[500];      // обязательно надо static !!!
  int i = 0,j = 0;
  while(s[i])
    str[j++] = s[i++];
  i = 0;
  while(ss[i])
    str[j++] = ss[i++];
  str[j] = 0;
  return str;
}
void main(void) {
  char s1[80], s2[80], *p;
  gets(s1);
  gets(s2);
  p = strcat(s1,s2);    // адрес можно сохранить
  puts(p);
}
```

24. ФУНКЦИИ С ПЕРЕМЕННЫМ КОЛИЧЕСТВОМ АРГУМЕНТОВ

24.1. Соглашения о вызовах: модификаторы функций

Во всех языках программирования компилятор формирует ассемблерный код вызова функции одинаково: по ассемблерной команде `call` процессор запоминает в стеке текущий адрес (адрес текущей команды) и управление передаётся в начало функции. В конце функция исполняет инструкцию `ret`, которая извлекает сохранённый адрес из стека обратно и дальше процессор продолжает исполнять программу именно с этого адреса. Но вызвать функцию в большинстве случаев мало. Надо ей ещё передать аргументы. Собственно правила их передачи и называют соглашениями о вызовах.

Куда можно «запихнуть» параметры для передачи? Таких мест немного: это регистры процессора и стек. Наиболее общепринятым соглашением считается передача всех параметров через стек. Параметры последовательно заносятся в стек с помощью команды `push`. Занесли параметры в стек, выполнили функцию, и теперь параметры нужно убрать из стека, чтобы вернуть его в начальное состояние. Кто это должен делать: вызывающая или вызываемая функция?

Соглашения о вызовах (calling convention) – это порядок, в котором параметры функции помещаются в стек, и способ очистки стека при возврате из функции.

Параметры могут помещаться в стек в прямом и в обратном порядке:

5) прямой порядок – параметры размещаются в стеке в том же порядке, в котором они перечислены в описании функции (в вершине стека всегда находится последний параметр, затем предпоследний и т.д.);
6) обратный порядок – параметры передаются в порядке от конца к началу (при любом количестве параметров в вершине стека находится сначала первый параметр, за ним второй и т. д.). Это упрощает реализацию функций с неопределённым числом параметров произвольных типов.

Очищать стек может сама вызванная функция, а может – вызывающая функция.

Соглашения о вызовах используется в основном при одновременном использовании функций, написанных на разных языках программирования (в разных языках программирования используются разные соглашения о вызове) и при вызове функций стандартных системных библиотек (например, `WinAPI`).

Основные соглашения:

`__cdecl` (C declaration) – основной способ вызова в языке C. Параметры передаются через стек. Параметры помещаются в стек в обратном порядке: помещаются справа налево от последнего к первому, а изымаются из стека от первого к последнему. Таким образом становиться возможно передавать в функцию неопределённое количествово параметров (например, как в `printf()`). После вызова функции стек очищает тот, кто вызвал функцию, т.е. вызывающая функция.

`__pascal` – основной способ вызова в языке Паскаль. Параметры передаются через стек. Параметры помещаются в стек в прямом порядке: помещаются слева

направо от первого к последнему, а изымаются из стека от последнего к первому. Число параметров фиксировано. После вызова функции стек очищает сама функция.

__stdcall – применяется при вызове функций `WinAPI`. Параметры передаются через стек. Параметры помещаются в стек в обратном порядке: помещаются справа налево от последнего к первому, а изымаются из стека от первого к последнему. После вызова функции стек очищает сама функция.

__fastcall – так называемый «быстрый вызов», применяется, например, в Delphi и Builder. Аналогичен, `stdcall`, но параметры (или первые из них, если их много) передаются через регистры процессора. Параметры помещаются в регистры и стек в обратном порядке: помещаются справа налево от последнего к первому, а изымаются из стека от первого к последнему. После вызова функции стек очищает сама функция.

Например, пусть есть вызов функции `F(A,B,C);`

```
__cdecl: ; обратный порядок
  push C
  push B
  push A
  call F
  ; ret
  inc esp, 12  ; очистка стека здесь
__pascal: ; прямой порядок
  push A
  push B
  push C
  call F
  ; ret 12
  ; очистка стека - в функции
__stdcall: ; обратный порядок
  push C
  push B
  push A
  call F
  ; ret 12
  ; очистка стека - в функции
__fastcall: ; обратный порядок, но, например, первый
            ; параметр передается через регистры
  push C
  push B
  mov edx, A
  call F
  ; ret 8
  ; очистка стека - в функции
```

Компилятор языка С умеет работать с разными соглашениями о вызовах. Во-первых, специальными ключами командной строки компилятора (`-p/-pc`) или опциями среды разработки можно изменить тип соглашения о вызовах, используемого по умолчанию (в BC: `Options→Compiler→Entry/Exit Code…→Calling Convention (C/Pascal/Register)`), что крайне не рекомендуется делать, поскольку это поменяет тип всех функций (за исключением тех, тип вызова которых указан явно), в том числе и библиотечных, и ваша программа просто перестанет работать. Во-вторых, в объявлении функции можно в явном виде указать с помощью специального ключевого слова, какому соглашению о вызовах она должна следовать. Например:

```
void __fastcall func(int a, int b);
void __pascal func(int a, int b);
```

С соглашениями о вызовах теснейшим образом связаны **соглашения о внутренних именах функций**, которые им присваивает компилятор. На самом деле, в `obj-`, `lib-` и `dll-`файлах функции имеют несколько иные имена, нежели в исходном тексте программы.

Функции внутри программ на языке С распознаются по внутренним именам. Внутреннее имя представляет собой строку, создаваемую компилятором при компиляции прототипа или определения функции. Форма декорирования для функции С зависит от правила вызова, используемого в ее объявлении:

`__cdecl` – внутреннее имя формируется компилятором с различием верхнего и нижнего регистра добавлением в начало символа подчеркивания (`_func`);

`__pascal` – внутреннее имя формируется компилятором приведением имени функции к верхнему регистру (`FUNC`);

`__stdcall` – внутреннее имя формируется компилятором с различием верхнего и нижнего регистра добавлением в начало символа подчеркивания, а в конец символа @ и числа байт, занимаемых аргументами (`_func@4`);

`__fastcall` – внутреннее имя формируется компилятором с различием верхнего и нижнего регистра добавлением в начало символа @, а в конец символа @ и числа байт, занимаемых аргументами (`@func@4`).

24.2. Объявление списка параметров переменной длины

Как мы уже выяснили, в языке С можно использовать функции с переменным числом параметров (количество аргументов при компиляции функции не фиксировано). Количество и тип аргументов становится известным только в момент вызова функции, когда явно задан список фактических аргументов. Самым известным примером таких функций являются функции `printf()` и `scanf()`.

Чтобы сообщить компилятору, что функции будет передано заранее неизвестное количество аргументов, объявление списка ее параметров необходимо закончить многоточием. Это указание компилятору, что дальнейший контроль соответствия количества и типов параметров при обработке вызова функции проводить не нужно.

Например, следующий прототип указывает, что у функции `func()` будет как минимум два целых параметра и после них еще некоторое количество (в том числе и ноль) параметров:

```
int func(int a, int b, ...);
```

В любой функции, использующей переменное количество параметров, должен быть как минимум один реально существующий параметр. Например, следующее объявление неправильное:

```
int func(...);    // ошибка
```

Также понятно, что многоточие может быть только в конце списка параметров. Например, следующее объявление тоже неправильное:

```
int func(int a, ..., int b);    // ошибка
```

Функция с переменным числов параметров должна иметь способ вычисления точного количества параметров при каждом вызове функции.

Можно поступить так – передавать точное число параметров при вызове функции через один из аргументов. Это не совсем удобно, т.к. надо самому считать количество параметров (можно, например, просто ошибиться при подсчете).

Второй способ – использовать какое-либо значение параметра как признак конца списка параметров. Например, если число 0 не может быть параметром, то можно просто последним параметром при вызове функции указывать 0 и прекращать обработку параметров в функции как только встретиться 0.

Задача. Написать функцию для нахождения суммы произвольного числа неотрицательных длинных целых чисел (long).

```
long sum1(int n, long p, ...) { // n – количество параметров
  long s = 0, *pa;
  int i;
  pa = &p;              // получаем адрес первого параметра
  for (i = 0; i < n; i++) {
    s += *pa;
    pa++;               // переходим к следующему параметру
  }
  return s;
}
long sum2(long p, ...) {          // признак конца: число < 0
  long s = 0, *pa;
  pa = &p;              // получаем адрес первого параметра
  while (*pa >= 0) {
    s += *pa;
    pa++;               // переходим к следующему параметру
  }
  return s;
}
```

```
void main() {
  long s, p1 = 1, p2 = 2, p3 = 3, p4 = 4, p5 = 5;
  s = sum1(5, 1L, 2L, 3L, 4L, 5L);
  s = sum1(5, p1, p2, p3, p4, p5);
  s = sum2(1L, 2L, 3L, 4L, 5L, -1L);
  // ошибка, т. к. параметры должны занимать в стек по 4 байта (long),
  // а не по 2 (int)
  // s = sum1(5, 1, 2, 3, 4, 5);        // ошибка!!!
}
```

Для доступа к фактическим параметрам в стеке используется увеличение указателя, что соответствует соглашению о вызовах `cdecl` – последний параметра помещается в стек первым, затем – второй и т.д. Значит первый параметр будет иметь меньший адрес, чем второй и т.д.

Для вызова функции `sum2()` из примера в стеке будем иметь:

← вершина стека

1	параметр 1
2	параметр 2
3	параметр 3
4	параметр 4
5	параметр 5

25. ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИЮ MAIN()

Любая программа на языке С начинается с вызова функции `main()`. Эта функция должна быть в каждой программе.

Как и любая другая функция, функция `main()` может иметь параметры. Иногда при запуске программы бывает полезно передать ей какую-либо информацию. Такая информация передается функции `main()` с помощью аргументов командной строки. *Аргументы командной строки* – это информация, которая вводится в командной строке вслед за именем программы при запуске программы на выполнение не из среды разработки программы. Например, чтобы запустить архивацию файла `task.cpp`, необходимо в командной строке набрать следующее:

`winrar` *`a archTask task.cpp`* `//` `winrar.exe` *`a archTask task.cpp`*

где `winrar` – имя программы-архиватора, а строки «*a*», «*archTask*» и «*task.cpp»* представляет собой аргументы командной строки, которые говорят программе, что надо создать архив («*a*») с именем *archTask* из одного файла *task.cpp*.

При передаче параметров в функцию `main()` ее надо определять так:

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

Параметр `argc` содержит количество аргументов в командной строке и является целым числом, причем он всегда не меньше 1, потому что первым аргументом всегда передается имя программы (имя программы с полным путем к программе).

Параметр `argv` является указателем на массив указателей на строки. В этом массиве каждый элемент указывает на очередной аргумент командной строки. Пустые квадратные скобки указывают на то, что у массива неопределенная длина. Получить доступ к отдельным аргументам можно с помощью индексации массива `argv`. Например, `argv[0]` указывает на первую символьную строку, которой всегда является имя программы; `argv[1]` указывает на первый аргумент и так далее. Список аргументов ограничен `NULL`, т.е. `argv[argc] == NULL`.

Чтобы получить доступ к отдельному символу одного из аргументов командной строки, надо использовать в `argv` второй индекс. Т.е., первый индекс `argv` обеспечивает доступ к строке, а второй индекс – доступ к ее отдельным символам.

Все аргументы командной строки являются строковыми, поэтому преобразование числовых параметров в нужный формат должно быть предусмотрено в программе при ее разработке.

Пример программы с разными способами перевода чисел в символьном формате в целые и вещественные числа:

```
#include <stdio.h>
#include <stdlib.h>
// при запуске задаем, например, такие аргументы: 100 2.7
void main(int a, char *b[]) {
  int k;
  double f;
  char *ptr;
  k = atoi(b[1]);
  f = atof(b[2]);
  k = strtol(b[1], &ptr, 10);  // ptr = адрес места ошибки в строке
  f = strtod(b[2], &ptr);
  sscanf(b[1], "%d", &k);
  sscanf(b[2], "%lf", &f);
}
```

Имена `argc` и `argv` являются традиционными, но не обязательными. Эти два параметра в функции `main()` можно называть как угодно.

Простой пример использования аргументов командной строки:

```
int main(int argc, char *argv[]) {
  int k;
  if (argc != 4) {
    printf("Неверные параметры запуска программы!\n");
    return 1;
  }
  k = atoi(argv[3]);  // преобразование параметра-числа
```

```
  printf("Привет, %s из группы %s %d-го курса",
         argv[1], argv[2], k);
  return 0;
}
```

Если имя программы – `task`, а ваше имя «Вася», группа «ПМ-11» с первого курса, то для запуска программы следует в командную строку ввести:

```
task Вася ПМ-11 1
```

В результате выполнения программы на экране появится сообщение: «Привет, Вася из группы ПМ-11 1-го курса».

Обратите внимание: если не все аргументы командной строки будут заданы, то будет выведено сообщение об ошибке. В программах с аргументами командной строки часто делается следующее: в случае, когда пользователь запускает эти программы без ввода нужной информации, выводятся инструкции о том, как правильно указывать аргументы.

Аргументы командной строки необходимо отделять друг от друга пробелом. Если в самом аргумента есть пробелы, то, чтобы из него не получилось несколько аргументов, этот аргумент надо заключать в двойные кавычки. В результате вся строка в кавычках будет считаться одним аргументом. Например, программу можно запустить так: `task "Вася и Петя" ПМ-21 2`. В результате выполнения программы на экране появится сообщение: «Привет, Вася и Петя из группы ПМ-21 2-го курса».

Что такое `char *argv[]`? Это *массив, элементами которого служат указатели*, то есть *массив указателей*. Значит при передаче параметров в `main()` ее можно определять и так:

```
void main(int argc, char **argv) {
  ...
}
```

Задача. Вывести на экран все аргументы командной строки (имя программы выводить не надо).

```
#include <stdio.h>
void main(int argc, char *argv[]){
  int i;
  for (i = 1; i < argc; i++)
    printf("%s\n", argv[i]);
}
```

```
===== второй вариант ===============
```

```
#include <stdio.h>
void main(int argc, char **argv){
  char *p;
  argv++;
  while((p=*argv) != NULL) {
    printf("%s\n", p);
```

```
    argv++;
  }
}
```

Обычно аргументы командной строки используют для того, чтобы передать программе начальные данные, которые понадобятся ей при запуске (например, через аргументы командной строки часто передаются такие данные, как имя файла или параметры запуска программы).

Когда для программы не требуются параметры командной строки, в списке параметров функции `main()` используют ключевое слово `void` (или просто ничего не указывают).

Как отлаживать в BC программы, требующие аргументы командной строки. В меню `Run→Arguments...` необходимо ввести аргументы командной строки. Имя программы указывать не надо. Дальше можно просто запускать и отлаживать программу в среде разработки как обычно.

26. УКАЗАТЕЛИ НА ФУНКЦИЮ

С функцией в языке С можно проделать только две вещи: вызвать ее и получить ее адрес. Указатели на функции – очень мощное средство языка С.

Функции для процессора – те же двоичные коды, и с виду они мало отличаются от строк или массивов. Так же, как и массивы, они занимают последовательные ячейки памяти, где хранятся двоичные числа. Разница в том, что эту последовательность ячеек процессор воспринимает не как идущие подряд числа или буквы, а как последовательность команд.

Функция располагается в памяти по определенному адресу, который можно присвоить указателю в качестве его значения. Адресом функции является ее точка входа. Именно этот адрес используется при вызове функции. Так как указатель хранит адрес функции, то она может быть вызвана с помощью этого указателя. Он позволяет также передавать ее другим функциям в качестве аргумента.

Указатель на функцию имеет тип «указатель на функцию, которая возвращает значение заданного типа и имеет аргументы заданного типа»:

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

Например, объявление:

```
int (*pfunc)(int, int);
```

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

Указателю на функцию, как и любому указателю, нужно перед использованием придать разумное значение. В программе на С адресом функции служит ее имя без скобок и аргументов (это похоже на адрес массива, который равен имени массива без индексов).

```
int sum(int a,int b){
```

```
  return a+b;
}
void main(){
  int (*f)(int, int);    // описание указателя на функцию
  int (*ff)(int, int);   // описание указателя на функцию
  int k;
  k = sum(5,9);   // обычный вызов функции sum()
  f = &sum;       // можно так занести в указатель адрес функции sum(),
  ff = sum;       // а можно и так занести в указатель адрес функции sum()
  k = (*f)(4,1); // можно так вызвать функцию через указатель,
  k = ff(11,22); // а можно и так вызвать функцию через указатель
}
```

Обратить внимание! При присваивании тип функции и тип указателя на функцию должны в точности совпадать.

Вывод: Имя функции и указатель на нее практически равноправны. Разница между ними в том, что имя функции нельзя отделить от нее самой, оно всегда связано с постоянным адресом. Указатель же более свободен, и его можно направить куда угодно.

Какая польза от вызова функции с помощью указателя на функцию? Ведь вроде бы никаких преимуществ не достигнуто, этим только усложняется программа. Тем не менее, во многих случаях оказывается более выгодным передать имя функции как параметр или даже создать массив функций.

Например, в программе обработки меню часто вызываются различные функции для разных пунктов меню. В таких случаях чаще всего используют оператор `switch` с длинным списком меток `case`. Другой подход – создать массив функций и вызывать их с помощью индексов – делает программу менее громоздкой и, соответственно, менее подверженной ошибкам.

Задача. Организовать обработку пунктов меню с помощью оператора switch.

```
#include <stdio.h>
#include <stdlib.h>   // для функции exit()
#include <bios.h>
void f0() {           // обработка пункта меню 0 – выход из программы
  printf("menu 0\n");
  exit(0);            // завершить работу программы
}
void f1() {           // обработка пункта меню 1
  printf("menu 1\n");
}
void f2() {           // обработка пункта меню 2
  printf("menu 2\n");
}
```

```
void f3() {            // обработка пункта меню 3
  printf("menu 3\n");
}

void main() {
  int i;
  while(1) {              // цикл по обработке пунктов меню
    printf("Введите номер пункта меню : ");
    scanf("%d",&i);
    switch(i) {
      case 0: f0(); break;
      case 1: f1(); break;
      case 2: f2(); break;
      case 3: f3();
    }
    bioskey(0);
  }
}
```

Задача. Организовать обработку пунктов меню с помощью массива указателей на функции.

```
void f0() {
  printf("menu 0\n");
  exit(0);
}
void f1() {
  printf("menu 1\n");
}
void f2() {
  printf("menu 2\n");
}
void f3() {
  printf("menu 3\n");
}

void main() {
  int i;
// описание массива указателей на функции и его инициализация
  void (*ff[4])(void) = {f0, f1, f2, f3};
// описание массива указателей на функции и занесение в него адресов функций
// void (*ff[4])(void);
// ff[0] = f0;
// ff[1] = f1;
// ff[2] = f2;
// ff[3] = f3;
  while(1) {
    printf("Введите номер пункта меню : ");
```

```
    scanf("%d",&i);
    ff[i](); // вызов функции, адрес которой занесен в i-ый элемент массива
    bioskey(0);
  }
}
```

Задача. *Пример передачи адреса функции как параметра в функцию.*

```
void f1() {
  printf("Функция 1\n");
}
void f2() {
  printf("Функция 2\n");
}
void fp(void (*ptrF)(void)) {
  ptrF();          //  вызов функции, адрес которой передан как параметр
}
void main() {
  void (*f)(void);
  fp(f1);
  f = f2;
  fp(f);
}
```

27. СТАНДАРТНЫЕ ФУНКЦИИ ЯЗЫКА С

27.1. Функции для работы со строками

В языке С нет встроенных операций для работы со строками, но имеются библиотечные функции для обработки строк. Прототипы функций для обработки строк описаны в файле <string.h>. Для их использования в своей программе необходимо включить в программу файл командой #include <string.h>.

Все функции для строк в качестве параметров используют адреса строк, а не сами строки. Другими словами, мы передаем в функцию адрес, который она считает адресом строки, и обрабатывает строку, начиная с этого адреса и до ноль-символа. За наличие ноль-символа в исходной строке отвечаем мы (он должен быть!), а вот адрес не обязательно строго должен быть адресом начала строки.

Рассмотрим основные функции (подробно все можно найти в Касаткине).

Определение длины строки:

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

Символ '\0' в длину строки не входит. Не путать с длиной массива, в котором размещается строка.

Слияние двух строк:

```
char *strcat(char *s1, char *s2);
```

К строке, на которую указывает `s1`, приписываются все символы строки `s2`, включая и символ ‘\0’. Размер памяти для `s1` должен быть такого размера, чтобы вместить результирующую строку. Функция возвращает адрес строки `s1`.

Слияние строки с частью другой строки:

```
char *strncat(char *s1, char *s2, int n);
```

К строке, на которую указывает `s1`, приписываются `n` символов строки `s2` (или меньше, если среди этих `n` символов встретиться символ ‘\0’). К результату автоматически приписывается символ ‘\0’. Размер памяти для `s1` должен быть такого размера, чтобы вместить результирующую строку. Функция возвращает адрес строки `s1`.

Копирование строки в строку:

```
char *strcpy(char *s1, char *s2);
```

Строка, на которую указывает `s2`, включая символ ‘\0’, копируется в строку, на которую указывает `s1`. Размер памяти для `s1` должен быть такого размера, чтобы вместить строку `s2`. Функция возвращает адрес строки `s1`.

Копирование части строки в строку:

```
char *strncpy(char *s1, char *s2, int n);
```

Часть строки, на которую указывает `s2`, размером `n` символов копируется в строку, на которую указывает `s1`. Если среди `n` копируемых символов встречается символ ‘\0’, то он копируется в `s1` и копирование прекращается. Иначе, копируется ровно `n` символов, а **символ ‘\0’ к результату не добавляется**. Размер памяти для `s1` должен быть такого размера, чтобы вместить результат. Функция возвращает адрес строки `s1`.

Сравнение двух строк в алфавитном порядке:

```
int strcmp(char *s1, char *s2);
```

Функция возвращает значение больше нуля, если строка `s1` больше `s2` в алфавитном порядке, меньше нуля, если строка `s1` меньше `s2`, и равное нулю, если строки равны.

Сравнение части строк:

```
int strncmp(char *s1, char *s2, int n);
```

Работает также как `strcmp()`, но сравнивает только `n` символов строк (или меньше, если раньше встречается символ ‘\0’).

Занесение символа во всю строку:

```
char *strset(char *s, char ch);
```

Помещает символ ch во все позиции строки s. Возвращает указатель на строку s.

Занесение символа n раз в строку:

```
char *strnset(char *s, char ch, int n);
```

Помещает символ ch в n первые позиции строки s. Символ '\0' в строке s не изменится, даже если n будет больше длины строки s. Возвращает указатель на строку s.

Поиск подстроки в строке:

```
char *strstr(char *s1, char *s2);
```

Отыскивает место первого вхождения строки s2 в строку s1. Возвращает указатель на начало вхождения. Если строка не найдена, возвращает NULL.

Поиск символа в строке:

```
char *strchr(char *s, char ch);
```

Функция осуществляет поиск символа ch с начала строки, на которую указывает s, и возвращает адрес найденного символа. Если символ не найден, возвращает NULL.

Поиск символа с конца строки:

```
char *strrchr(char *s, char ch);
```

Функция осуществляет поиск символа ch с конца строки, на которую указывает s, и возвращает адрес найденного символа. Если символ не найден, возвращает NULL.

Поиск любого символа не из набора символов:

```
int strspn(char *s1, char *s2);
```

Функция ищет позицию в строке s1 первого символа, которые не принадлежит строке s2. Возвращает длину сегмента строки s1, состоящего только из символов, входящих в строку s2.

Поиск любого символа из набора символов:

```
int strcspn(char *s1, char *s2);
```

Функция ищет позицию в строке s1 первого вхождения любого символа из строки s2. Возвращает длину сегмента строки s1, состоящего только из символов, не входящих в строку s2.

Поиск любого символа из набора символов:

```
char *strpbrk(char *s1, char *s2);
```

Функция отыскивает место первого вхождения любого символа из строки s2 в строке s1. Возвращает указатель на первый найденный символ. Если символ не найден, возвращает NULL.

Реверсирование строки:

```
char *strrev(char *s);
```

Функция реверсирует строку, на начало которой указывает s, и возвращает указатель на полученную строку.

Создание копии строки:

```
char *strdup(char *s);
```

Функция распределяет память и делает копию строки. Возвращает указатель на начало строки-копии.

Форматный вывод в строку:

```
int sprintf(char *s, char *format, ...);
```

Функция работает подобно printf(), но вывод вместо экрана осуществляет в буфер, на который указывает s. Его размер должен быть достаточным для того, чтобы вместить всю выводимую информацию. Функция возвращает число выведенных байт.

Выделение в строке фрагментов:

```
char *strtok(char *s1, char *s2);
```

Функция выделяет лексему в строке s1. Выделяемая лексема – это фрагмент строки s1, ограниченный с обеих сторон любым из символов из строки s2. Возвращает адрес выделенной лексемы. При неудаче функция возращает NULL.

Рассмотрим функцию strtok() более подробно. У функции strtok() два параметра: первый – указатель на анализируемую строку, второй – вспомогательная строка, где указываются символы, которые strtok() будет считать разделителями.

Допустим, что надо найти и вывести на отдельной строке все слова в предложении. Будем считать, что слова в предложении отделяются пробелом или запятой.

```
ptr = strtok(s,", "); // ищем первое слово, исходная строка разрушается
while(ptr != NULL){
  puts(ptr);
  ptr = strtok(NULL,", ")  // ищем следующее слово в той же строке
}
```

Примеры вызова функций с пояснениями:

```
char s1[120], s2[120], s3[120], *p;
int len;
gets(s1);             // s1: |a|b|c|\0|
len = strlen(s1);     // len = 3
strcpy(s2,s1);        // s2: |a|b|c|\0|
gets(s3);             // s3: |u|v|\0|
strcat(s2,s3);        // s2: |a|b|c|u|v|\0|
strncpy(s3,s2,4);     // s3: |a|b|c|u|?|?|
s3[4]=0;              // s3: |a|b|c|u|\0|
```

```
strncat(s3,s1,2);     // s3: |a|b|c|u|a|b|\0|
if((p=strchr(s3,'b'))!=NULL) {...}     // p ≡ s3+1
if((p=strstr(s3,"def"))==NULL) {...}  // p ≡ NULL
if(!strcmp(s1,"abc")) {...}           // true
if(strcmp(s1+1,s2)<=0) {...}          // false
```

Обратить внимание! Нельзы вызывать `strcat()`, указывая при вызове адреса из одной и той же строки:

```
strcat(s1,s1); strcat(s1, s1+2);   // ошибка !!!
```

Обратить внимание! Надо четко понимать, что параметрами функций для работы со строками являются указатели на строки, а не сами строки. Поэтому совершенно не обязательно эти указатели должны указывать на начало строк. В функции строкой считается все, начиная от переданного адреса и до символа ‘\0’.

Задача. Если это возможно, то удалить в строке два первых и два последних символа.

```
#include <stdio.h>
#include <string.h>
void main(void) {
  char s[80], len;
  gets(s);
  len = strlen(s);
  if (len >= 4) {
    s[len-2] = 0;      // удаление двух последних символов
    strcpy(s,s+2);     // удаление двух первых символов
  }
  puts(s);
}
```

Задача. Ввести две строки и удалить в первой строке последнее вхождение второй строки.

```
void main(void) {
  char s1[80], s2[80], *p, *ps1, *ps2 = NULL;
  gets(s1); gets(s2);
  ps1 = s1;              // устанавливаем ps1 на начало строки
  p = strstr(ps1,s2);    // поиск подстроки
  while (p != NULL) {    // нашли s2
    ps2 = p;             // сохраняем адрес найденной s2
    ps1 = p + 1;         // смещаем ps1 для следующего поиска
    p = strstr(ps1,s2);// поиск подстроки
  }
  if (ps2)               // нашли s2
    strcpy(ps2,ps2+strlen(s2));  // удаляем последнюю s2
  puts(s1);
}
```

Задача. *Создать массив слов из строк, вводимых с клавиатуры. Окончание ввода – пустая строка.*

```
#define MAXSTR 20
void main(void) {
  char w[MAXSTR][80];
  int k = 0, i;
  printf("Введите строки (<%d)\n", MAXSTR);
  printf("Окончание ввода – пустая строка\n");
  while(*gets(w[k])) {
    k++;
    if(k == MAXSTR) break;
  }
  puts("Введенный массив слов");
  for(i=0; i<k; puts(w[i++]));
}
```

Задача. *Создать предложение из слов, вводимых с клавиатуры. Окончание ввода – пустая строка.*

```
void main(void) {
  char w[10], s[120]="";
  while(*gets(w)) {
    strcat(s, w);          // одним оператором:
    strcat(s, " ");        // strcat(strcat(s,w)," ");
  }
  if(*s)                   // предложение не пустое
    *(s+strlen(s)-1) = 0; // удаляем последний лишний пробел
  puts(s);
}
```

Задача. *Дано предложение. Занести все его слова в массив строк.*

```
void main(void){
  char s[120], w[20][40], *p;
  int k = 0, i = 0;
  gets(s);
  p = strtok(s, " ");
  while(p != NULL) {
    strcpy(w[k++], p);
    p = strtok(NULL, " ");
  }
  while(i < k)
   puts(w[i++]);
}
```

Задача. *Дано предложение. Создать новое предложение, удалив из исходного знаки пунктуации и лишние пробелы.*

```
void main(void){
  char s1[120], s2[120] = "", *p, d[]=" .,:;!?-";
  gets(s1);
  p = strtok(s1, d);
  while(p != NULL) {
    strcat(strcat(s2,p)," ");
    p = strtok(NULL, d);
  }
  if(s2[0])                // предложение не пустое
    s2[strlen(s2)-1] = 0; // удаляем последний лишний пробел
  puts(s2);
}
```

Задача. Дано предложение. Создать массив указателей на встречающиеся в нем слова.

```
void main(void) {
  char s[80], *d[60], *p, *razd[] = " ";
  int i, k = 0;
  gets(s);
  p = strtok(s), razd);
  while( p!= NULL) {
    for(i=0; i<k; i++)          // ищем слово в массиве
      if(!strcmp(p,d[i])) break;
    if(i == k)                 // слова еще не было
      d[k++] = p;
    p = strtok(NULL, razd);
  }
  for(i=0; i<k; puts(w[i++]));
}
```

Задача. Дано предложение. Реверсировать каждое его слово.

```
void main(void) {
  char s[120], ss[120], *p, lenp;
  gets(s);
  strcpy(ss,s);
  p = strtok(s," ");
  while(p!=NULL) {
    lenp = strlen(strrev(p));
    p[lenp] = ss[p-s+lenp];
    p = strtok(NULL," ");
  }
  puts(s);
}
```

27.2. Функции для проверки символов и преобразования данных

Макроопределения, описанные в заголовочном файле <ctype.h>:

`int isalnum(int ch)`	истина, если `ch` буква или цифра
`int isalpha(int ch)`	истина, если `ch` буква
`int isdigit(int ch)`	истина, если `ch` цифра
`int islower(int ch)`	истина, если `ch` буква нижнего регистра
`int isupper(int ch)`	истина, если `ch` буква верхнего регистра
`int ispunct(int ch)`	истина, если `ch` знак пунктуации
`int isspace(int ch)`	истина, если `ch` пробел, знак табуляции, возврат каретки, символ перевода строки, вертикальной табуляции, перевода страницы
`int isxdigit(int ch)`	истина, если `ch` шестнадцатеричная цифра

Функции преобразования символов:

`int tolower(int ch)`	преобразует `ch` к нижнему регистру (работает только для букв латинского алфавита)
`int toupper(int ch)`	преобразует `ch` к верхнему регистру (работает только для букв латинского алфавита)

Функции преобразования данных:

`int atoi(char *s)`	преобразует строку `s` в десятичное целое
`long atol(char *s)`	преобразует строку `s` в длинное десятичное целое
`double atof(char *s)`	преобразует строку `s` в вещественное число
`char *itoa(int v, char *s, int baz)`	преобразует целое `v` в строку `s` для системы счисления `baz`
`char *ltoa(long v, char *s, int baz)`	преобразует длинное целое `v` в строку `s` для системы счисления `baz`
`char *ultoa(unsigned long v, char *s, int baz)`	преобразует длинное беззнаковое целое `v` в строку `s` для системы счисления `baz`

27.3. Функция быстрой сортировки – gsort()

Вы уже должны знать некоторые способы сортировки – и быстрые, и не очень. Если вам надо отсортировать большой массив данных, без метода быстрой сортировки не обойтись. Для быстрой сортировки есть функция в стандартной библиотеке языка С, причем она позволяет сортировать массивы данных произвольного типа (как базового так и пользовательского). Программисты часто изобретают собственные функции, потому что им лень разбираться с чужими. Но в результате затрачивается больше времени, а программа получается хуже. Стандартными функциями стоит еще заниматься потому, что они открывают много нового в самом языке С. Особенно это справедливо в отношении функции `qsort()`. Описание функции находится в файле `<stdlib.h>`.

Прототип функции быстрой сортировки:

```
void qsort(const void *base,
```

```
unsigned int n, unsigned int size,
int (*cmp)(const void *, const void *));
```

Описание параметров: `base` – указатель на начало сортируемого массива, `n` – число сортируемых элементов в массиве, `size` – размер в байтах одного элемента массива, `int (*cmp)(const void*, const void*)` – указатель на функцию сравнения, которая показывает, какой из двух сравниваемых элементов больше. Использование в качестве параметра указателя на функцию сравнения и позволяет сортировать нам массивы произвольного типа и так, как нам захочется – по возрастанию или убыванию, по одному или нескольким полям, строки по алфавиту и по длине и т.д.

Т.к. тип массива может быть произвольный, формальный параметр с адресом массива имеет тип `void *base`. В применении к указателям `void *` говорит об универсальности. Указатель на `void` – это *обобщенный* указатель, о котором нельзя сказать, на что он указывает, до тех пор, пока его не приведут к какому-то конкретному типу. Объявление первого параметра функции как указатель на `void` позволяет принять *любой* указатель. Если бы в списке параметров стояло `int *base`, функция `qsort()` могла бы обрабатывать только целочисленные данные.

Функция, на которую указывает cmp, сравнивает два сортируемых элемента. Ее нужно писать самостоятельно, потому что, сравнивать можно по-разному. Функция `qsort()` вызывает, когда это нужно, функцию `cmp`, передавая ей *указатели* на сравниваемые элементы. Функцию `qsort()` не интересуют внутренности функции сравнения `cmp`. Нужно только, чтобы `cmp` возвращала положительное значение, если первый аргумент больше второго, ноль – если они равны и отрицательное значение, если первый аргумент меньше второго. Указатели на `void` сообщают нам, что функция `cmp` может сравнивать элементы любого размера, а слово `const` говорит о том, что с помощью передаваемого указателя нельзя изменить *сам* элемент. И это разумно, потому что функция `cmp` не должна *менять* передаваемые ей элементы массива. Она должна только решать, какой из них больше.

Функция сравнения сравнивает между собой два адресуемых элемента массива и возвращает в зависимости от результата сравнения целое число:

если элементы:	*возвращает:*
`*elem1 < *elem2`	`целое число < 0`
`*elem1 == *elem2`	`0`
`*elem1 > *elem2`	`целое число > 0`

При сравнении «меньше, чем» означает, что левый элемент в конце сортировки должен оказаться перед правым аргументом. Аналогично, «больше, чем» означает, что в конце сортировки левый элемент должен оказаться после правого.

Задача. *Примеры применения функции `qsort()` для сортировки массива целых чисел по возрастанию и убыванию абсолютных величин, и массива строк по убыванию длины, причем строки одной длины сортируются по алфавиту.*

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

```
#include <stdlib.h>   // для функции qsort()
#include <string.h>
#include <conio.h>

int cmpInt(const void *,const void *);
int cmpAbs(const void *,const void *);
int cmpString(const void *, const void *);
int cmpPointer(const void *, const void *);

void main(){
  int i;
  int x[10] = {-5, 3, 2, -4, 6, 7, 11, -17, 0, 13};
  char w[10][20] = {"b", "bbb",  "ab",  "a",   "aa",
                    "c", "dddd", "aaa", "ccc", "ba"};
  char *p[10] = {"b", "bbb", "ab", "a", "aa",
                 "c", "dddd", "aaa", "ccc", "ba"};

  puts("============================");
  qsort(x, 10, sizeof(int), cmpInt);
  for(i=0; i<10; i++)
    printf("%d ", x[i]);
  printf("\n============================\n");
  getch();
  qsort(x, 10, sizeof(x[0]), cmpAbs);
  for(i=0; i<10; i++)
    printf("%d ", x[i]);
  printf("\n============================\n");
  getch();
  qsort(w, 10, sizeof(w[0]), cmpString);
  for(i=0; i<10; i++)
    puts(w[i]);
  puts("============================");
  getch();
  qsort(p, 10, sizeof(char *), cmpPointer);
  for(i=0; i<10; i++)
    puts(p[i]);
  puts("============================");
  getch();
}
// для сортировки целых цисел по возрастанию
int cmpInt(const void *a, const void *b){
  int n1, n2;
  n1 = *(int *)a;
  n2 = *(int *)b;
  return n1-n2;
}

// для сортировки целых цисел по убыванию абсолютных величин
```

```
int cmpAbs(const void *a, const void *b){
  int n1, n2;
  n1 = abs(*(int *)a);
  n2 = abs(*(int *)b);
  if(n1 > n2) return -1;
  else if(n1 == n2) return 0;
  else return 1;
  // можно и так: return n2-n1;
}

// для сортировки строк по убыванию длины,
// строки равной длины сортируются по алфавиту
int cmpString(const void *a, const void *b){
  char *s1, *s2;
  int n1, n2;
  s1 = (char *)a;
  s2 = (char *)b;
  n1 = strlen(s1);
  n2 = strlen(s2);
  if(n1 > n2) return -1;
  else if(n1 == n2) return strcmp(s1,s2);
  else return 1;
}

// для сортировки строк как массива указателей по алфавиту
int cmpPointer(const void *a, const void *b){
    char *s1,*s2;
    int n1, n2;
    s1=*(char **)a;
    s2=*(char **)b;
    return strcmp(s1,s2);
}
```

<u>Задача.</u> Пример частичной сортировки массива целых чисел.

```
#include <stdio.h>
#include <stdlib.h>   // для функции qsort()
#include <conio.h>

// для сортировки целых цисел по возрастанию
int cmpInt(const void *a, const void *b){
  return *(int *)a - *(int *)b;
}

void main(){
  int i;
  int x[10] = {-5, 3, 2, -4, 6, 7, 11, -17, 0, 13};
  qsort(x+2, 6, sizeof(int), cmpInt);
  for(i=0; i<10; i++)
    printf("%d ", x[i]);
```

```
//   увидим массив с отсортированными средними шестью элементами:
//   -5 3 -17 -4 2 6 7 11 0 13
  printf("\n");
  getch();
}
```

27.4. Функция двоичного поиска – bsearch()

Двоичный поиск (дихотомия) – это быстрый поиск элемента в отсортированном множестве данных. Для двоичного поиска в стандартной библиотеке языка С есть функция `bsearch()`, причем она позволяет выполнять поиск элементов в массиве данных произвольного типа (как базового так и пользовательского). Описание функции находится в файле `<stdlib.h>`.

Прототип функции двоичного поиска:

```
void *bsearch(const void *key, const void *base,
              unsigned int n, unsigned int size,
              int (*cmp)(const void*, const void*));
```

Функция осуществляет поиск в массиве и возвращает адрес первого элемента, который соответствует шаблону поиска. Если соответствие не найдено, то функция возвращает значение NULL. Так как `bsearch()` выполняет двоичный поиск, то первый соответствующий элемент не обязательно будет первым таким элементом в массиве.

Описание параметров: `key` – адрес переменной, значение которой надо найти в массиве, `base` – указатель на начало массива для поиска, `n` – число элементов в массиве, `size` – размер в байтах одного элемента массива.

Параметр `int (*cmp)(const void*, const void*)` указатель на функцию сравнения, которая показывает, какой из двух сравниваемых элементов больше. Подробное описание такой функции приведено при описании функции быстрой сортировки `qsort()`.

Задача. *Пример поиска в одномерном и двумерном массивах целых чисел.*

```
#include <stdlib.h>  // для функций qsort() и bsearch()
#include <stdio.h>

int cmp (const void *p1, const void *p2)  {
 return *(int *)p1-*(int*)p2;
}

void main(){
  int *ptrFind;  // адрес найденного элемента в массиве
  int find;      // для поиска
  int i;
  int a[3][3] = { {5, 0, 12},
                  {1, 20, 9},
                  {17, 3, 7} };
```

```
  int m[10] = {11, 3, 9, 4, 0, 15, 19, 18, 12, 7};
  // сортируем двумерный массив, считая его одномерным из 9 элементов
  // результат сортировки: {0, 1, 3},
  //                       {5, 7, 9},
  //                       {12, 17, 20}
  qsort(a, 9, sizeof(a[0][0]),cmp);
  // сортируем одномерный массив
  qsort(m, 10, sizeof(m[0]),cmp);
  for (i=0; i<5; i++) {
    printf("find = ");
    scanf("%d", &find);
    ptrFind = (int *)bsearch(&find,a,9,sizeof(int),cmp);
    if (ptrFind != NULL)
      printf("%d есть в двумерном массиве a\n", find);
    else
      printf("%d нет в двумерном массиве a\n", find);
    ptrFind = (int *)bsearch(&find,m,10,sizeof(int),cmp);
    if (ptrFind != NULL)
      printf("%d есть в одномерном массиве m", find);
    else
      printf("%d нет в одномерном массиве m", find);
  }
}
```

28. РАБОТА С ФАЙЛАМИ

28.1. Основные понятия

До сих пор во всех наших программах ввод данных производился только с клавиатуры, а вывод – только на экран. Следующий шаг – научиться писать программы, которые умели бы работать с файлами.

Если в качестве устройств ввода/вывода ограничиться только клавиатурой и экраном, то в таком случае будет сложно обработать большие объемы входных данных. Выходные данные, отображенные на экране, тоже безвозвратно теряются. Для устранения подобных проблем удобно сохранять данные на запоминающих устройствах, предназначенных для долговременного хранения данных (диски, флэшки и т.п.).

На дисках данные хранятся в виде структур данных, обслуживаемых операционной системой, – в виде файлов. Файл – это именованная область внешней памяти, в которой хранится логически завершенный объем данных. В языке С файл – это неструктурированная линейная последовательность байтов. Файлы широко применяются при решении различных задач. Каждый файл имеет имя и расширение, которого может и не быть (в MS DOS максимальная длина соответственно: 8 и 3 символа).

В языке С отсутствуют операторы для работы с файлами. Все необходимые действия с файлами выполняются с помощью функций стандартной библиотеки.

Функции ввода/вывода для работы с файлами стандартной библиотеки языка С, которые позволяют читать данные из файлов и записывать данные в файлы, делятся на два класса:

1. *Ввод/вывод верхнего уровня* (с использованием понятия «поток»). Эти функции обеспечивают *буферизацию* работы с файлами. Это означает, что при чтении или записи информации обмен данными осуществляется не между программой и указанным файлом, а между программой и промежуточным буфером, расположенным в оперативной памяти.

При записи в файл информация из буфера записывается или при его заполнении, или при закрытии файла. При чтении данных программой информация берется из буфера, а в буфер она считывается при открытии файла и впоследствии каждый раз при опустошении буфера. Буферизация ввода/вывода выполняется автоматически, она позволяет ускорить выполнение программы за счет уменьшения количества обращений к сравнительно медленно работающим внешним устройствам.

Для пользователя файл, открытый на верхнем уровне, представляется как последовательность считываемых или записываемых байтов. Чтобы отразить эту особенность организации ввода/вывода, предложено понятие «*поток*» (англ. stream). Когда файл открывается, с ним связывается поток. Далее выводимая информация записывается «в поток», а считываемая – берется «из потока».

Когда файл открывается для ввода/вывода, он связывается со структурой типа `FILE`, определенной с помощью `typedef` в файле `<stdio.h>`. Эта структура содержит всю необходимую информацию о файле (его имя, открыт ли файл для чтения или записи, указатель текущей позиции в файле, адрес буфера, были ли ошибки при работе с файлом и не встретился ли конец файла). При открытии файла с помощью стандартной функции `fopen()` возвращается указатель на структуру типа `FILE`. Этот указатель, называемый *указателем потока (файла)*, используется для последующих операций с файлом. Его значение передается всем библиотечным функциям, используемым для ввода/вывода через этот поток.

2. *Ввод/вывод низкого уровня* (с использованием понятия «дескриптор» или «префикс»). Функции низкого уровня не выполняют буферизацию и форматирование данных, они позволяют непосредственно пользоваться средствами ввода/вывода операционной системы.

При низкоуровневом открытии файла с помощью функции `open()` с ним связывается файловый дескриптор (англ. handle). Дескриптор является целым числом, характеризующим размещение информации об открытом файле во внутренних таблицах операционной системы (`FCB – File Control Block`). Дескриптор используется при последующих операциях с файлом.

Файлы делятся на текстовые и двоичные.

Текстовый файл – это файл, в котором каждый символ хранится в виде одного байта (кода, соответствующего символу). Текстовые файлы разбиваются на несколько строк с помощью специального символа «конец строки» (пара символов CR LF или 0Dh 0Ah).

Двоичный файл – файл, данные которого представлены в бинарном виде. При записи в двоичный файл символы и числа записываются в виде последовательности байтов (в своем внутреннем двоичном представлении в памяти компьютера).

Для обоих классов функций файлового ввода-вывода возможны 2 режима доступа к файлу: текстовый и двоичный. Режим доступа к файлу задается при его открытии. В *текстовом* режиме производится преобразование пары символов CR LF (0Dh 0Ah) – при чтении из файла в один символ новой строки `'\n'(0xA)`, а при записи в файл обратно в пару символов. Кроме того, при вводе символа Ctrl-Z (1Ah) считается, что достигнут конец файла и ввести информацию после этого символа в текстовом режиме нельзя. В *двоичном* режиме никакого преобразования байтов не происходит и не делается никаких предположений об их смысле.

Доступ к файлу может быть последовательным и произвольным (прямым). Последовательный доступ – файл читается/пишется последовательно байт за байтом. Прямой доступ – чтение/запись возможны из произвольного места файла с установкой указателя чтения/записи в нужное место файла.

Далее мы подробно рассмотрим только доступ к файлам через поток ввода/вывода. Ввод/вывод низкого уровня во многом похож на ввод/вывод высокого уровня и предлагается вам для самостоятельного изучения.

28.2. Основные функции для работы с файлами

Прототипы функций для работы с файлами находятся в файле `<stdio.h>`.

Функция	Краткое описание
`fopen()`	Открыть файл
`freopen()`	Переоткрыть файл с другими правами доступа
`fclose()`	Закрыть файл
`fcloseall()`	Закрыть все открытые файлы
`fputc()`	Записать символ в файл
`fgetc()`	Прочитать символ из файла
`fputs()`	Записать строку в файл
`fgets()`	Прочитать строку из файла
`fprintf()`	Форматированный вывод в файл
`fscanf()`	Форматированный ввод из файла
`fwrite`	Записать блок данных в файл
`fread`	Прочитать блок данных из файла
`fflush()`	Дозаписать определенный поток в файл
`flushall()`	Дозаписать все открытые потоки в файлы
`feof()`	Возвращает значение true (истина), если достигнут конец файла
`fseek()`	Установить указатель текущей позиции на определенный байт файла
`ftell()`	Получить значение указателя текущей позиции в файле
`rewind()`	Устанавить указатель текущей позиции в начало файла

Также в <stdio.h> определяется несколько макросов: EOF, FOPEN_MAX, SEEK_SET, SEEK_CUR и SEEK_END. Макрос EOF, часто определяемый как -1, является значением, возвращаемым тогда, когда функция ввода пытается выполнить ввод после конца файла. FOPEN_MAX определяет целое значение, равное допустимому максимальному числу одновременно открытых файлов. Другие макросы используются в функции fseek(), выполняющей операции прямого доступа к файлу.

28.3. Открытие и закрытие файлов

Прежде чем работать с файлом (проводить обмен информацией между ним и программой), его надо открыть.

Функция fopen() открывает файл в заданном режиме. Прототип функции:

```
FILE *fopen(const char *name, const char *mode);
```

где *name* – это указатель на строку символов, представляющую собой допустимое имя файла (надо задавать полный путь к файлу, если он находится не в каталоге запуска программы). Строка *mode* определяет режим открытия файла.

Функция fopen() возвращает указатель файла – указатель на структурную переменную по шаблону FILE. Никогда не следует изменять значение этого указателя в программе. Если при открытии файла происходит ошибка, то fopen() возвращает пустой указатель (NULL).

Режимы открытия файла:

Режим	Описание
r	Открыть существующий файл для чтения
w	Открыть существующий файл для записи с усечением (старое содержимое файла стирается). Если такого файла нет, он будет создан
a	Открыть файл для добавления в существующий файл (старое содержимое сохраняется). Если такого файла нет, он будет создан
r+	Открыть существующий файл для чтения/записи. Чтение/запись допустимы в любом месте файла. В этом режиме невозможна запись в конец файла, то есть недопустимо увеличение размеров файла
w+	Открыть файл для чтения/записи. Если файл с таким именем уже существует, он будет очищен. Чтение/запись допустимы в любом месте файла, в том числе запись разрешена и в конце файла, т.е. файл может увеличиваться.
a+	Открыть файл для добавления или создать для чтения/записи, если он не существует. Чтение/запись допустимы в любом месте файла. При этом в отличие от режима "w+" можно открыть существующий файл и не уничтожать его содержимое; в отличие от режима "r+" в режиме "a+" можно вести запись в конец файла, то есть увеличивать его размеры.

Для указания на текстовый режим работы с файлом после любого из этих значений может быть добавлен символ 't' (режим по умолчанию), для указания на

бинарный режим – символ 'b'. В бинарном режиме с файлами рекомендуется работать с помощью функций блокового ввода/вывода fread() и fwrite().

Пример открытия текстового файла по имени "test.txt" для записи:

```
FILE *fp;
if ((fp = fopen("test", "w")) == NULL) {
  printf("Ошибка открытия файла!\n");
  exit(1);              // завершение работы программы
}
```

Обратить внимание! Всегда нужно вначале убедиться, что файл открылся успешно, и лишь затем выполнять с файлом какие-либо операции. Это позволяет при открытии файла обнаружить любую ошибку (например, защиту от записи или отсутствие места на диске, причем обнаружить еще до того, как программа попытается в этот файл что-либо записать).

Допустимое максимальное число одновременно открытых файлов определяется константой FOPEN_MAX. Это значение не меньше 8, но чему оно точно равняется определяется компилятором.

Функция fclose() закрывает файл, который был открыт с помощью функции fopen(). Функция fclose()закрывает файл на уровне операционной системы (при выводе в файл записывает в него все данные, которые еще остались в буфере).

При нормальном завершении работы программы для каждого открытого файла fclose() вызывается автоматически. Но отказ от использования в программе fclose() может повлечь всевозможные неприятности, включая потерю данных, испорченные файлы и возможные периодические ошибки в программе.

Прототип функции закрытия файла:

```
int fclose(FILE *fp);
```

где *fp* – указатель файла, возвращенный в результате вызова fopen().

Возвращение функцией fclose() нуля означает успешную операцию закрытия. В случае же ошибки возвращается EOF.

28.4. Ввод/вывод символов

Функция fputc() выводит один символ в файл, который с помощью функции fopen() открыт в режиме записи. Прототип функции:

```
int fputc(int ch, FILE *fp);
```

где *fp* – это указатель файла, в который надо вывести символ, а *ch* – выводимый символ (в файл выводится только младший байт).

Если функция выполнилась успешно, то возвращается записанный символ. В противном случае возвращается EOF.

Функция `fgetc()` вводит один символ из файла, который с помощью `fopen()` открыт в режиме чтения. Прототип функции:

```
int fgetc(FILE *fp);
```

где *fp* – это указатель файла, из которого надо ввести символ.

Функция возвращает целое значение, в котором введенный символ находится в младшем байте (старший байт/байты будет нулевыми).

При достижении конца файла функция `fgetc()` возвращает `EOF`. Однако `fgetc()` возвращает `EOF` и в случае ошибки. Для определения того, что же на самом деле произошло, можно использовать функцию `ferror()`.

Задача. Ввести символы из одного файла и вывести в другой файл.

```
void main() {
  FILE *fin, *fout;
  int ch;
  if ((fin=fopen("in","r")) == NULL) {
    printf("Ошибка открытия входного файла\n"); return;
  }
  if ((fout=fopen("out","w")) == NULL) {
    printf("Ошибка открытия выходного файла\n"); return;
  }
  ch = fgetc(fin);          //
  while (ch != EOF) {       //  while ((ch=fetc(in)) != EOF)
    fputc(ch, fout);        //    fputc(ch, fout);
    ch = fgetc(fin);        //
  }                         //
  fclose(fin);
  fclose(fout);
  printf("Работа программы завершена успешно\n");
}
```

28.5. Ввод/вывод строк

Функции `fgets()` и `fputs()` вводят строки символов из файла и выводят строки символов в файл. Прототипы функций:

```
int fputs(const char *str, FILE *fp);
char *fgets(char *str, int N, FILE *fp);
```

Функция `fputs()` выводит в файл строку, на которую указывает *str*. Вывод строки выполняется до нуль-символа. Функция не добавляет в файл при выводе автоматически символ новой строки `'\n'` и не выводит нулевой байт. В случае успеха возвращает последний записанный символ, в случае ошибки возвращает `EOF`.

Функция `fgets()` вводит из файла строку, и делает это до тех пор, пока не будет прочитан символ новой строки или количество прочитанных символов не станет равным *N*-1. Если был прочитан символ `'\n'`, он записывается в строку (этим

функция `fgets()` отличается от функции `gets()`). В конец введенной строки всегда автоматически добавляется нулевой байт конца строки. В случае успеха функция возвращает указатель на прочитанную строку, в случае ошибки или конца файла – `NULL`.

Задача. *Ввести строки символов из одного файла и вывести их в другой файл и на экран.*

```
void main() {
  char s[100];
  FILE *fin, *fout;
  int ch;
  if ((fin=fopen("in","r")) == NULL) {
    printf("Ошибка открытия входного файла\n"); return;
  }
  if ((fout=fopen("out","w")) == NULL) {
    printf("Ошибка открытия выходного файла\n"); return;
  }
  while (fgets(s,100,fin) != NULL) {
    // fputs(s,fout);        // файл out == in
    int len = strlen(s);
    if (s[len-1] == '\n')    // убираем из строки символ '\n'
      s[len-1] = 0;
    puts(s);
    fputs(s,fout);
  }
  fclose(fin);
  fclose(fout);
  printf("Работа программы завершена успешно\n");
}
```

Структура исходного файла "`in`":

```
1          // 31 0D 0A              =>  "1\n"
22         // 32 32 0D 0A           =>  "22\n"
333        // 33 33 33 0D 0A        =>  "333\n"
4444       // 34 34 34 34 0D 0A     =>  "4444\n"
55555      // 35 35 35 35 35 0D 0A  =>  "55555\n"
```

Структура полученного файла "`out`":

```
122333444455555 // 31 32 32 33 33 33 34 34 34 34 35 35 35 35 35
```

Если ввод выполнять так – `fgets(s,5,fin)`, будут введены строки:

```
"1\n"  "22\n"  "333\n"  "4444"  "\n"  "5555"  "5\n"
```

28.6. Форматированный ввод/вывод

Форматированный файловый ввод/вывод выполняется с помощью функций `fscanf()` и `fprintf()`. Они предназначены для работы с файлами, но ведут себя

точно так же, как printf() и scanf(). Прототипы функций:

```
int fprintf(FILE *fp, char *format, ...);
int fscanf(FILE *fp, char *format, ...);
```

В случае успеха функция fprintf() вернет число записанных байт, в случае ошибки – EOF.

В случае успеха функция fscanf() вернет число успешно прочитанных, преобразованных и сохраненных полей ввода; вернет 0, если не удалось сохранить ни одно поле; вернет EOF при попытке чтения за пределами конца файла.

```
fout = fopen("out","w");      // текстовый режим
fprintf(fout, "%s %d %d\n", "abcd", 10, 100);
=== F4 === out === (13 байтов)
abcd 10 100
=== 16-ричный вид === out ===
61 62 63 64 20 31 30 20 31 30 30 0D 0A

fout = fopen("out","wb");     // двоичный режим
fprintf(fout, "%s %d %d\n", "abcd", 10, 100);
=== F4 === out === (12 байтов)
abcd 10 100         // преобразование в строку и вывод в файл
=== 16-ричный вид === out ===
61 62 63 64 20 31 30 20 31 30 30 0A
```

Задача. *Вывести в файл, а затем ввести из файла 5 строк. Каждая из записываемых строк содержит строку с текстом "Line" (5 байт, включая нулевой байт), пробел, целое значение i, пробел и вещественное значение квадратного корня из i.*

```
void main() {
  FILE *f;
  int i, k;
  double d;
  char s[] = "Line";
  if ((f=fopen("file.txt","w")) == NULL) {
    printf("Ошибка открытия файла\n"); return;
  }
  for(i=1; i<6; i++) {
    fprintf(f, "%s %d %.2lf\n", s, i, sqrt(i));// запись в файл
    printf("%s %d %.2lf\n", s, i, sqrt(i));   // вывод на экран
  }
  fclose(f);
  printf("\n");
  if ((f=fopen("file.txt ","r")) == NULL) {
    printf("Ошибка открытия файла\n"); return;
  }
  for(i=1; i<6; i++) {
```

```
    fscanf(f, "%s %d %lf", s, &k, &d);          // чтение из
файла
    printf("%s %d %.2lf\n", s, k, d);           // вывод на экран
  }
  fclose(f);
  printf("Работа программы завершена успешно\n");
}
```

```
//=== Результат работы программы ===
```

```
Line 1 1.00
Line 2 1.41
Line 3 1.73
Line 4 2.00
Line 5 2.24
```

```
Line 1 1.00
Line 2 1.41
Line 3 1.73
Line 4 2.00
Line 5 2.24
```

Обратить внимание! Если при выводе в файл между `%s` и `%d` не сделать пробел, то в файле текст `"Line"` склеится с последующим целым числом. После этого при чтении из файла в переменную `s` будут попадать строки вида `"LineN"`, в переменную `k` будут считываться старшие цифры корня (до точки), а в переменной `d` окажутся дробные разряды соответствующего корня.

```
//=== Результат работы программы ===
```

```
Line 1 1.00
Line 2 1.41
Line 3 1.73
Line 4 2.00
Line 5 2.24
```

```
Line1 1 0.00
Line2 1 0.41
Line3 1 0.73
Line4 2 0.00
Line5 2 0.24
```

Хотя читать разносортные данные из файлов и писать их в файлы часто легче всего именно с помошью функций `fprintf()` и `fscanf()`, это не всегда самый эффективный способ выполнения операций чтения и записи. Так как данные в формате `ASCII` записываются так, как они должны появиться на экране (а не в двоичном виде), то каждый вызов этих функций сопряжен с определенными накладными расходами. Поэтому, если есть ограничения на размер файла или скорость работы программы, надо использовать функции `fread()` и `fwrite()`.

28.7. Ввод/вывод блоков данных

Блоковый ввод/вывод чаще всего используется при работе со структурными переменными или массивами. Функции блокового ввода/вывода позволяют переносить между файлом и программой блок данных. Под блоком будем понимать группу подряд расположенных байтов. При использовании функций блокового ввода/вывода файл следует открывать в двоичном режиме, а в качестве размера блока указывать размер переменной `sizeof(var)`. Прототипы функций:

```
int fwrite(void *ptr, int size, int n, FILE *fp);
int fread(void *ptr, int size, int n, FILE *fp);
```

Эти функции позволяют читать и записывать блоки данных любого типа.

Функция `fwrite()` выводит в файл `n` элементов данных по `size` байт каждый. Общее число записываемых байт равно `n*size`. Данные берутся из буфера, на который указывает `ptr`. В случае успеха функция возвращает число элементов (не байт!), записанных в файл, в случае ошибки – меньшее, чем `n` значение.

Функция `fread()` вводит из файла `n` элементов данных по `size` байт каждый в память, на которую указывает `ptr`. Общее число прочитанных байт будет равно `n*size`. В случае успеха функция возвращает число прочитанных элементов (не байт!), в случае ошибки – меньшее, чем `n` значение.

```
char *sa = "abcd";
int a = 10, b = 100;

fout = fopen("out","w");    // текстовый режим
fwrite(sa, sizeof(sa), 1, fout);
fwrite(&a, sizeof(a), 1, fout);
fwrite(&b, sizeof(b), 1, fout);
```

```
=== F4 === out === (9 байтов)
abcd ???        // вывод в файл во внутреннем представлении
=== 16-ричный вид === out ===
61 62 63 64 OD 0A 00 64 00  // при вводе 0D 0A => 0A
```

```
fout = fopen("out","wb");     // двоичный режим
fwrite(sa, sizeof(sa), 1, fout);
fwrite(&a, sizeof(a), 1, fout);
fwrite(&b, sizeof(b), 1, fout);
```

```
=== F4 === out === (8 байтов)
abcd ???
=== 16-ричный вид === out ===
61 62 63 64 0A 00 64 00     // для a=12 => 0C 00
```

в обоих случаях

Задача. *Вывести массив A в файл, а затем ввести данные из файла в массив B.*

```
void main(void) {
  FILE *fin, *fout;
```

```
  int A[10] = {1,2,3,4,5,6,7,8,9,10};
  int B[10];
  if((fout=fopen("test", "wb")) == NULL) {
    printf("Ошибка открытия файла\n"); return;
  }
  fwrite(A, sizeof(int), 10, fout);
  // fwrite(A, sizeof(a), 1, fout);
  fclose(fout);
  if((fin=fopen("test", "rb")) == NULL) {
    printf("Ошибка открытия файла\n"); return;
  }
  fread(B, sizeof(int), 10, fin);
  // fread(B, sizeof(b), 1, fin);
  fclose(fin);
  printf("Работа программы завершена успешно\n");
}
```

Задача. Вывести в файл разнотипные данные, а затем ввести их из файла.

```
void main(void) {
  FILE *fin, *fout;
  double d = 10.15;
  int i = 101;
  long l = 1234567L;
  if((fout=fopen("test", "wb")) == NULL) {
    printf("Ошибка открытия файла\n"); return;
  }
  fwrite(&d, sizeof(double), 1, fout);
  fwrite(&i, sizeof(int), 1, fout);
  fwrite(&l, sizeof(long), 1, fout;
  fclose(fout);
  if((fin=fopen("test", "rb")) == NULL) {
    printf("Ошибка открытия файла\n"); return;
  }
  fread(&d, sizeof(double), 1, fin);
  fread(&i, sizeof(int), 1, fin);
  fread(&l, sizeof(long), 1, fin);
  fclose(fin);
  printf("%f\n%d\n%ld\n", d, i, l);
  printf("Работа программы завершена успешно\n");
}
```

В качестве буфера можно использовать (и часто именно так и делают) просто память, в которой размещена переменная.

Одним из самых полезных применений функций `fread()` и `fwrite()` является чтение и запись данных пользовательских типов, особенно структур.

```
struct Student {
```

```
  char fio[50];
  int year;
  char adr[80];
} stud;
fwrite(&stud, sizeof(struct Student), 1, fout);
fread(&stud, sizeof(struct Student), 1, fin);
```

28.8. Другие средства для работы с файлами

Функция feof(). Данная функция (на самом деле это макрос) позволяет определить, достигнут ли конец файла. Прототип функции:

```
int feof(FILE *fp)
```

Функция `feof()` возвращает ненулевое значение, если при последней операции ввода из файла был обнаружен конец файла, и 0 в противном случае. Можно использовать как для двоичных, так и для текстовых файлов.

Задача. Ввести из файла неопределенной количество чисел типа long.

```
void main(void) {
  FILE *fp;
  long t;
  if((fp=fopen("in", "r")) == NULL) {
    printf("Ошибка открытия файла\n"); return;
  }
  fscanf(fp, "%ld", &t);
  while (!feof(fp)) {
    printf("%ld\n", t);
    fscanf(fp, "%ld", &t);
  }
  fclose(fp);
}
```

Обратить внимание! Функция `feof()` возвращает ненулевое значение, только если была попытка чтения чего-либо после конца файла. Поэтому код ниже является ошибочным (последнее введенное из файла число на экране будет отображено дважды).

```
while (!feof(fp)) {
  fscanf(fp, "%ld", &t);
  printf("%ld\n", t);
}
```

Функции fflush() и flushall(). Данные функции применяются для флэширования потоков (одного конкретного потока или всех открытых потоков). Прототипы функций:

```
int fflush(FILE *fp);
int flushall();
```

Если заданный поток *fp* открыт для вывода, то функция `fflush()` записывает все данные, находящиеся в буфере, в соответствующий файл Если поток открыт для ввода, то функция `fflush()` очищает содержимое буфера (следующая операция чтения считывает новые данные из входного файла в буфер). После вызова функции поток остается открытым. После своего успешного выполнения функция `fflush()` возвращает нуль, в противном случае – `EOF`.

Функция flushall() выполняет действия, аналогичные fflush(), только не для одного конкретного потока, а для всех открытых потоков. Функция возвращает количество открытых потоков (входных и выходных). В случае ошибки возвращаемого значения нет.

Буферы автоматически обновляются, когда они полны, когда потоки закрываются или происходит нормальное завершение работы программы без закрытия потоков.

Произвольный доступ к файлам. Ввод/вывод обычно бывает последовательным, т. е. каждая новая операция чтения/записи имеет дело с позицией файла, следующей за той, что была в предыдущей операции чтения/записи. Другими словами, при последовательном доступе файл читается или пишется последовательно байт за байтом. Операции ввода/вывода начинаются с текущей позиции в файле. Указатель чтения/записи в файле изменяется после каждой операции чтения или записи. Например, при чтении символа из файла указатель продвигается на 1 байт, так что следующая операция начнется с первого несчитанного символа. Для того, чтобы получить доступ к байту с номером `K`, необходимо просмотреть (прочитать или записать) от начала файла `K-1` предшествующих байтов. После обращения к байту с номером `K` можно обращаться к байту с номером `K+1`. Если файл открыт для добавления, указатель файла автоматически устанавливается на конец файла перед каждой операцией записи.

Последовательный доступ не всегда удобен. В ряде случаев бывает необходимо обеспечить произвольный (прямой) доступ к файлу. Прямой доступ означает возможность заранее определить в файле блок, к которому будет применена операция ввода/вывода. Т.е. для произвольного доступа необходимо наличие средств позиционирования внутри файла.

Файлы произвольного доступа очень распространены в реальной работе, т.к. в них можно легко добавлять, удалять записи, не перезаписывая остальные данные. Это очень удобно, так как иногда приходится изменять одну и ту же запись много раз.

При последовательном доступе данные в файле обычно имеют разную длину. При произвольном доступе очень желательно, чтобы данные имели определенную длину, что позволяет быстро и легко найти нужную нам запись. Произвольный доступ к файлу с данными различной длины значительно усложняет программу.

Обычно прямой доступ используют для файлов, открытых в двоичном режиме. Причина тут простая – так как в текстовом режиме могут выполняться преобразования символов, то может и не быть прямого соответствия между тем, что находится в файле и тем байтом, к которому нужен доступ. Даже если в файле находится

один только текст, все равно этот файл при необходимости можно открыть и в двоичном режиме. Никакие ограничения, связанные с тем, что файлы содержат текст, к операциям прямого доступа не относятся. Эти ограничения относятся только к файлам, открытым в текстовом режиме.

Место в файле, куда ведется запись или откуда осуществляется считывание, определяется значением указателя чтения/записи. Стандартная библиотека языка С содержит функции для установки этого указателя в нужное место, что позволяет получить доступ к любому байту в файле.

Функция `rewind()`. Данная функция устанавливает указатель текущей позиции в файле на начало файла. Прототип функции:

```
void rewind(FILE *fp);
```

Функция `fseek()`. Данная функция предоставляет способ передвигаться по файлу, не читая и не записывая данные, т.е. устанавливает указатель текущей позиции в файле. Функция `fseek()` позволяет нам обрабатывать файл подобно массиву. Прототип функции:

```
int fseek(FILE *fp, long offset, int from_where);
```

Параметр *`offset`* определяет, как далеко следует передвинуться от начальной точки, заданной параметром *`from_where`*. Он должен иметь значение типа `long`, которое может быть положительным (движение вперед) или отрицательным (движение назад). Параметр *`from_where`* определяет, откуда отсчитывать смещение *`offset`*: значение `SEEK_SET` – от начала файла; `SEEK_CUR` – от текущей позиции указателя чтения/записи в файле; `SEEK_END` – от конца файла. Функция вернет 0, если указатель успешно перемещен, в противном случае – вернет ненулевую величину (-1).

Функция `ftell()`. Данная функция возвращает текущее положений указателя чтения/записи в файле. Прототип функции:

```
long ftell(FILE *fp);
```

Позиция указателя чтения/записи отсчитывается от начала файла, начиная с 0. В случае ошибки функция вернет -1.

Значение, возвращаемое `fteel()`, физически не отражает байтового смещения `offset` для файла, открытого в текстовом режиме, т.к. текстовый режим преобразует комбинации символов. Для текстового режима надо использовать функцию `ftell()` с параметром `offset=SEEK_SET` совместно с `fseek()`, чтобы корректно запомнить и восстановить месторасположения указателя в файле.

28.9. Ввод/вывод низкого уровня (префиксный доступ к файлам)

Информация по теме: А.И. Касаткин «Управление ресурсами» – стр. 103-112.

Функции нижнего уровня не требуют включения файла `<stdio.h>`. Тем не менее, несколько общих констант, определенных в этом файле, могут оказаться по-

лезными (например, признак конца файла EOF). Прототипы функций нижнего уровня содержатся в файле <io.h>.

Функции низкоуровневого ввода-вывода не выполняю никакой буферизации и форматирования. Они непосредственно обращаются к средствам ввода/вывода операционной системы. При открытии файла на этом уровне возвращается дескриптор (file handle), представляющий собой целое число, используемое затем для обращения к этому файлу при дальнейших операциях. Для открытия используется функция open(), для закрытия – функция close(). Функция read() читает данные в указанный буфер, а write() – выводит данные из буфера в файл, lseek() – используется для перемещения по файлу.

Ниже кратко перечислены функции для низкоуровневого доступа к файлам.

Функция	Краткое описание
open()	Открыть файл
close()	Закрыть файл
creat()	Создать файл
eof()	Проверка на конец файла
tell()	Получить текущую позицию указателя файла
lseek()	Позиционирование указателя файла в заданное место
read()	Читать данные из файла
write()	Записать данные в файл

29. ТИПЫ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ: ПЕРЕЧИСЛЕНИЯ, СТРУКТУРЫ И ОБЪЕДИНЕНИЯ

Базовые типы данных (int, char, float, …) языка С, а также массивы и указатели, являются фундаментом, на котором строится обработка реальной информации. Но их бывает недостаточно для представления некоторых сложных совокупностей данных. Язык С позволяет программисту определять (или создавать) собственные типы данных.

29.1. Переименование типов – оператор typedef

Переименование типов используется для введения осмысленных или сокращенных имен типов, что повышает понятность программ, и для улучшения переносимости программ (имена одного типа данных могут различаться на разных машинах, и если с помощью переименования типов объявить типы данных, которые являются машинно-зависимыми, то при переносе программы на другую машину потребуется внести изменения только в определения переименований типов).

В языке С есть оператор typedef, который позволяет давать типам данных новые имена. Определение типа с typedef имеет следующий вид:

```
typedef <тип> <новое_имя_типа>;
```

Примеры:

```
typedef long Large; // определяет тип Large, эквивалентный типу long

typedef int Len;    // делает имя Len синонимом int

typedef char* String; // делает String синонимом char *,
                      // т.е. указателем на char
```

Объявляемый в `typedef` тип стоит на месте имени переменной в обычном объявлении, а не сразу за словом `typedef`. Имена новых типов обычно пишут с заглавных букв для того, чтобы они выделялись в тексте программы.

После такого переименования типов типы `Large`, `Len` и `String` можно применять в объявлениях, в операции приведения типа и т.д. точно так же, как типы `long`, `int` и `char *`:

```
Large x, *plong;
Len *plen, maxlen;
String p = "ggg", ptr[10];
```

Обратить внимание! Объявление `typedef` не создает объявления нового типа, оно лишь дает новое имя уже существующему типу. Никакого нового смысла эти новые имена не несут. Они объявляют переменные в точности с теми же свойствами, как если бы те были объявлены напрямую без переименования типа.

29.2. Перечисления (enum)

Перечисления (enumeration) – это пользовательский тип данных. Перечисления используют для описания какого-то небольшого множества целых чисел. С помощью перечислений можно задать дни недели, месяцы, цвета и т.п..

Определение перечислимого типа данных:

```
enum имя_перечислимого_типа {
  имя1 [ = константа1],
  имя2 [ = константа2],
  ...
};
```

Сначала идет ключевое слово «enum». Затем указывается имя перечисления, после которого в фигурных скобках задаются значения, которые смогут принимать переменные этого нового типа. После фигурных скобок ставится точка с запятой. Значения констант задаются в виде: `имя[=константа]`, где `имя` – символическое имя перечислимой константы, `константа` – необязательное значение, задающее значение перечислимой константы.

Значение перечислимым константам задаются либо явной инициализацией, либо в случае ее отсутствия – по умолчанию. В случае отстутствия явной инициализации первой константе ставится в соответствие значение 0, а каждая следующая перечислимая константа считается имеющей значение на 1 больше, чем предыдущая. Можно задавать значения только части констант. Если мы зададим для какой-либо константы значением 51, то следующая константа получит значение 52. И так

до тех пор, пока не встретится константа с заданным явно инициализирующим значением.

Примеры определения перечислений:

```
enum color {  // цвета: 0, 1, 2
  red, green, yellow
};
enum days {   // день недели: 1, 2, 3, 4, 5, 6, 7
  monday = 1, tuesday, wednesday, thursday, friday,
  saturday, sunday
};
enum arrow {  // клавиши стрелок: 72, 75, 77, 80
  north = 72, west = 75, east = 77, south = 80
};
```

После того, как определено перечисление, можно создавать переменные нового типа:

```
enum color pen;
enum days day_week;
enum arrow x;
enum {                // можно и так описывать переменные,
  red, green,         // без определения имени enum
  yellow, white
} col;
```

С помощью `typedef` можно определить новый тип для перечисления:

```
typedef enum {
  red, green, yellow
} COLOR;

COLOR pen1;
```

Имя переменной перечислимого типа можно использовать всюду в выражениях. Но присваивать такой переменной можно только значение, задаваемое перечисляемыми константами (явное значение или символическое имя константы).

```
void main() {
  enum color {  // цвета: 0, 1, 2
    red, green, yellow
  };
  enum color pen;
  pen = red;
  if (pen!= green) { ... }
  pen = 2;
  if (pen == yellow) { ... }
  if (pen == 2) { ... }
  int f = pen;
//  red = 10;   // ошибка!!! red - это константа
```

```
//  pen = 10;    // ошибка!!! нет такой константы в color
}
```

29.3. Основные сведения о структурах

В отличие от массивов или перечислений, структуры позволяют определять новые типы путем логического группирования переменных различных типов. Необходимость: часто реальный объект характеризуется величинами разного типа. Пример: товар на складе => название `char name[21];`
цена `float price;`
количество `int number;`

Все три переменных неразрывно связаны с каким-то товаром.

Структуры – это одна или несколько переменных (возможно, разных типов), описывающих, обычно, какую-нибудь сущность, и которые для удобства работы с ними сгруппированы под одним именем (в Паскале структуры называются записями). Структуры помогают в организации сложных данных (особенно в больших программах), поскольку позволяют группу связанных между собою переменных трактовать не как множество отдельных элементов, а как единое целое. Переменные в структурах хранятся в одном месте, можно создать несколько структурных переменных и у всех у них будут одинаковые характеристики.

Описание структурной переменной состоит из двух шагов: задание шаблона структуры и собственно описание структурной переменной.

Шаблон структуры определяет новый структурный тип данных и имеет следующий формат:

```
struct имя_шаблона_структуры {
  описание_элементов
};
```

Каждый шаблон имеет собственное имя. Имена шаблонов должны быть уникальными в пределах их области видимости (определяется по тем же правилам, что и для обычных переменных). Имя шаблона может отсутствовать (считается что такой шаблон имеет имя «нет имени»). Понятно, что без имени можно определять только единственный шаблон в функции.

Перечисленные в структуре переменные называются полями или элементами структуры. В структуре должен быть указан хотя бы один элемент. Элемент структуры не может быть структурой того же типа, в которой он содержится. Однако он может быть объявлен как *указатель на тип структуры*, в которую он входит. Это позволяет создавать связанные списки структур.

Имена полей в одном шаблоне должны быть уникальными, однако в разных шаблонах можно использовать совпадающие имена. Кроме этого, имена шаблонов не должны совпадать только между собой. Это значит, что шаблон может иметь такое же имя как и обычная переменна (хотя так делать не рекомендуется).

Примеры шаблонов структур:

1) студент (ФИО, год рождения, адрес):

```
struct Student {
```

```
  char fio[80];
  int year;
  char address[100];
};
```

2) точка на плоскости как пара целых координат:

```
struct Point {
  int x;
  int y;
};
```

Определяя шаблон структуры, мы по сути создаем новый тип данных – «struct имя_шаблона_структуры». Такие типы данных, определяемые программистом, называются пользовательскими.

Когда мы определяем шаблон структуры, под неё не выделяется память. Но теперь на основе этого шаблона можно создать много структурных переменных, под которые уже будет выделяться память, достаточная для хранения всех полей структуры (размер памяти для хранения одной структурной переменной можно определить с помощью sizeof(struct имя_шаблона_структуры)). Все эти переменные будут иметь тип «struct имя_шаблона_структуры» и у каждой будет свой набор переменных – полей структуры.

```
struct Student stud = {"Иванов", 1990, "Гомель"};
struct Student stud1;
struct Point pnt1, pnt2 = {5, 7};
```

Имя структуры обозначает значение всей области памяти, которую она занимает.

При описании структурных переменных допускается инициализация. Для этого используется список значений в фигурных скобках. Значения в скобках присваиваются переменным внутри структуры в том порядке, в котором они были объявлены при определении структуры. Если заданы значения не для всех полей, то оставшиеся без инициализации поля обнуляются.

Можно совмещать описание шаблона и структурной переменной:

```
struct Point {
  int x;
  int y;
} p1, p2 = {0,0};

struct {
  int x, y;
  char a[50];
} a, b = {1, 10}, c;     // b = {1, 10, ""}
```

Для упрощения описания структурных переменных рекомендуется в описании структуры использовать оператор typedef:

```
typedef struct Student {
  char fio[80];
```

```
  int year;
  char address[100];
} STUDENT_GGU;

typedef struct {   // имя шаблона можно и не задавать
  int x;
  int y;
} MYPOINT;
```

В этом случае можно не писать ключевое слово `struct` в описании структур вновь созданного типа или при передаче их функциям.

```
MYPOINT pnt1, pnt2;
STUDENT_GGU stud;
```

Обратить внимание! Каждая структурная переменная обладает своим набором переменных, которые были объявлены в структуре.

В языке С реализован ограниченный набор операций над структурами как единым целым: передача структуры в качестве аргумента функции, возврат структуры из функции, получение ее адреса и определение указателя на структуру. Структуры нельзя сравнивать.

Можно присваивать одну структуру другой, если они соответствуют одному шаблону (имеют одинаковое имя шаблона).

```
MYPOINT pnt1, pnt2 = {5, 5};
pnt1 = pnt2;
struct Student stud = {“Иванов”, 1990, “Гомель”};
struct Student stud1;
stud1 = stud;
```

Операция присваивания структурных переменных приводит к физической пересылке в памяти числа байтов, равного размеру шаблона структуры.

29.4. Структурные переменные в памяти компьютера

Поля структурной переменной располагаются в оперативной памяти последовательно в том порядке, в котором они объявляются: первому элементу соответствует меньший адрес памяти, а последнему – больший. Адрес переменной (поля структуры) – адрес младшего байта этой переменной. Адрес самой структурной переменной совпадает с адресом первого поля.

Обратить внимание! Поля структурной переменной хранятся в памяти друг за другом, между ними нет никаких других данных. Две структурные переменные необязательно расположены в памяти рядом.

При выделении памяти под структурную переменную учитывается такой параметр, как выравнивание структуры. Выравнивание задается опцией компилятора (`Options→Compiler→Code generation...→Word alignment`). Может быть задано выравнивание структуры или на границе байта, или на границе слова.

При выравнивании на границу байта все поля структуры в памяти располагаются вплотную одно за другим, без дырок. Длина структурной переменной будет равна сумме длин всех ее полей, а адрес может быть как четным, так и нечетным.

При выравнивании на границе слова компилятор при размещении структурной переменной в памяти вставляет между ее полями (а также между элементами массива структур) пустые байты так, чтобы соблюдались следующие правила:

1) отдельная структурная переменная (элемент массива) начинается на границе слова (с четного адреса);

2) любое поле не типа char начинается с четного адреса (имеет четное смещение от начала самой переменной);

3) при необходимости в конце структурной переменной добавляется еще один байт, чтобы общее число байтов для переменной было четным.

```
struct AAA {
  int a;
  char c[3];
  int b;
};
int x = sizeof(AAA);   // x = 8  при выравнивании на границу слова
                       // x = 7  при выравнивании на границу байта
```

29.5. Доступ к полям структуры

Для доступа к отдельным полям структурной переменной используется операция `'.'`, а доступ к отдельному полю структуры осуществляется посредством конструкции вида:

```
имя_структурной_переменной.имя_поля
```

Сначала мы указываем имя структурной переменной, затем ставим точку и в конце указываем имя поля структуры.

Так как структура – это новый тип, можно описывать указатели на этот тип. Для доступа к полям структуры через указатель (через адрес структуры) используется операция `'->'` вместо операции `'.'`.

Ссылка на поле структурной переменной с использованием `'.'` или `'->'` может располагаться в любом месте выражения, точно так же, как и простая переменная. Она также обладает всеми свойствами обычной переменной.

Пример:

```
struct Point {
  int x;
  int y;
};

struct Point pnt1, pnt2 = {5, 7};

pnt1.x = 1;
pnt1.y = 10;
```

```
printf("Точка 2: x=%d y=%d", pnt2.x, pnt2.y);
float dist;              // расстояние между точками 1 и 2
dist = sqrt((pnt1.x - pnt2.x) * (pnt1.x - pnt2.x)+
            (pnt1.y - pnt2.y) * (pnt1.y - pnt2.y));

struct Point *ptr = &pnt2;
ptr->x = 0;   // (*ptr).x = 0;  pnt2.x = 0;  (&pnt2)->x = 0;
ptr->y = 0;   // (*ptr).y = 0;  pnt2.y = 0;  (&pnt2)->y = 0;
dist = sqrt((pnt1.x - ptr->x) * (pnt1.x - pnt2.x)+
            (pnt1.y - ptr->y) * (pnt1.y - pnt2.y));
```

Поля структуры могут иметь любой тип, в том числе и быть другой структурой. Другими словами, структуры могут быть вложены друг в друга. Понятно, что шаблон вкладываемой структуры должен быть уже известен компилятору. Ссылка на поле вложенной структуры формируется из имени структурной переменной, имени структурного поля и имени поля вложенной структуры.

Пример (прямоугольник как пара точек на углах одной из его диагоналей):

```
struct Point {
  int x;
  int y;
};

struct Rect {
  struct Point pnt1;
  struct Point pnt2;
  int color;
};

struct Rect t1, t2 = {{0,0}, {5,5}, 2};
struct Rect *p_rect;
struct Point *p_point;

t1.pnt1 = t2.pnt2;
t1.pnt2.x = 11;
t1.pnt2.y = t2.pnt1.y + 10;
t1.color = 4;            // t1 = {{5,5}, {11,10}, 4};

p_rect = &t2;
p_point = &t2.pnt2;

p_rect->pnt1 = t1.pnt2;
p_rect->pnt2.x = 55;
p_point->y = 77;
t2.color = 1;            // t2 = {{11,10}, {55,77}, 1};
```

Структура типа `S` не может содержать элемент, являющийся структурой типа `S` (структура не может вкладываться сама в себя). Однако структура типа `S` может содержать элемент, указывающий на структуру типа `S`. Это позволяет использовать структуры для построения сложных динамических структур данных – стеков, списков, деревьев и т.п.

```
struct Node {
  int data;
  struct Node *next;  // указатель на объект типа Node
};
```

29.6. Массивы структур

Процесс описания массива структур совершенно аналогичен описанию массива любого другого типа:

```
struct Student {
  char fio[80];
  int year;
  char address[100];
};

struct Student grp[35];
```

Объявляем массив `grp`, состоящий из 35 элементов. Каждый элемент массива представляет собой структуру типа `Student`, т.е. `grp[0]` является в массиве первой `Student`-структурой, `grp[1]` – второй `Student`-структурой и т.д.

Массив структур можно инициализировать при описании:

```
struct Student grp1[]= { {“Иванов”, 1990, “Гомель”},
                         {“Петров”, 1989, “Минск”},
                         {“Волков”, 1993, “Мозырь”},
                         {“Кузмин”, 1988, “Добруш”},
                         {“Зайцев”, 1990, “Гомель”} };
```

Число элементов массива `grp1` будет вычислено по количеству инициализаторов, т.к. внутри квадратных скобок `[]` ничего не задано.

Доступ к элементам массива структур может выполняться с использованием индекса или через указатель-константу, которым является имя массива.

Пример доступа к студенту с номером `i`:

```
strcpy(grp[i].fio, “Иванов”);
(*(grp+i)).year = 1990;
strcpy((grp+i)->address, “Гомель”);
```

Для доступа к элементам массива структур может использовать и обычный казатель:

```
struct Student *ptr;
ptr = grp;
strcpy(ptr[i].fio, “Иванов”);
(*(ptr+i)).year = 1990;
strcpy((ptr+i)->address, “Гомель”);
ptr++;       // переход к студенту с номером i+1, т.е.
             // увеличивает указатель на размер типа struct Student
```

<u>Задача.</u> Описать группу студентов, заполнить информацию о группе и студентах группы с клавиатуры, определить самого юного студента в группе, отсортировать студентов группы по фамилии.

```
void main() {
  int i, j;
  struct Student {
    char fio[80];
    int year;
    char adr[100];
  };
  struct Gruppa {
    char name[10];          // название группы
    int num;                // количество студентов в группе
    struct Student stud[35];  // студенты группы
  } grp;
  struct Student *min, st;
  printf("Название группы: ");
  scanf("%s", grp.name);
  printf("Количество студентов: ");
  scanf("%d", &grp.num);
  printf("Введите информацию о студентах группы: \n");
  for (i=0; i<grp.num; i++) {
    printf("Студент %d: \n", i);
    flushall();
    printf("Фамилия: ");
    gets(grp.stud[i].fio);
    printf("Год рождения: ");
    scanf("%d", &grp.stud[i].year);
    flushall();
    printf("Адрес: ");
    gets(grp.stud[i].adr);
  }
  // поиск самого юного студента
  if (grp.num == 0) min = NULL;
  else min = &grp.stud[0];
  for (i=1; i<grp.num; i++)
    if (grp.stud[i].year <  min->year)
      min = &grp.stud[i];
  if (min) {
    printf("Самый юный студент: \n");
    printf("Фамилия = %s\n", min->fio);
    printf("Год рождения = %d\n", min->year);
    printf("Адрес = %s\n", min->adr);
  }
  else
```

```
    puts("В группе нет студентов!\n");
  // сортировка студентов по фамилии
  for (i=0; i<grp.num-1; i++)
    for (j=i+1; j<grp.num; j++)
      if (strcmp(grp.stud[i].fio,grp.stud[j].fio)>0) {
        st = grp.stud[i];
        grp.stud[i] = grp.stud[j];
        grp.stud[j] = st;
      }
  printf("Отсортированный список группы %s:\n", grp.name);
  for (i=0; i<grp.num; i++) {
    printf("Студент %d:\n", i);
    printf("Фамилия = %s\n", grp.stud[i].fio);
    printf("Год рождения = %d\n", grp.stud[i].year);
    printf("Адрес = %s\n", grp.stud[i].adr);
  }
}
```

29.7. Структуры и функции

Передавать структуры в функции можно по-разному. Допустим нам надо написать функции для точек и прямоугольников.

```
struct Point {
  int x;
  int y;
};
struct Rect {
  struct Point pnt1;
  struct Point pnt2;
  int color;
};
```

Возникает вопрос: а как передавать функциям информацию о точках и прямоугольниках? Существует по крайней мере три подхода: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый подход имеет свои плюсы и минусы.

```
// формирование точки по компонентам x и y
struct Point makepoint(int x, int y) {
  struct Point temp;
  temp.x = x;
  temp.y = y;
  return temp;
}
```

Никакого конфликта между именами аргументов и именами членов структуры не возникает.

Теперь с помощью функции makepoint() можно выполнять инициализацию любой структуры или формировать структурные аргументы для той или иной функции:

```
struct Rect screen;
struct Point middle;
screen.pnt1 = makepoint(0, 0);
screen.pnt2 = makepoint(25, 50);
middle = makepoint((screen.pnt1.x + screen.pnt2.x) / 2,
                   (screen.pnt1.y + screen.pnt2.y) / 2);

// сложение двух точек

struct Point addpoint(struct Point p1, struct Point p2) {
  p1.x += p2.x;
  p1.y += p2.y;
  return p1;
}
```

Здесь оба аргумента и возвращаемое значение – структуры. В функции мы увеличиваем поля прямо в p1 и не используем для этого временной переменной, т.к. структурные параметры функции передаются по значению точно так же, как и любые другие параметры.

Если функции передается большая структура, то эффективнее передавать указатель на структуру, а не создавать в стеке копию структуру целиком. Также, если функция должна изменять структурную переменную, в нее следует передавать указатель на модифицируемую структуру.

```
// определение местоположения и цвета прямоугольника

struct Rect makerect(struct Rect *p) {
  printf("Введите координаты прямоугольника:\n");
  scanf("%d%d%d%d", &p->pnt1.x, &p->pnt1.y,
                    &p->pnt2.x, &p->pnt2.y);
  printf("Введите цвет прямоугольника:\n");
  scanf("%d", &p->color);
  return *p;
}

struct Rect t1, t2;
t2 =  makerect(&t1);    // t2 == t1
```

Без каких либо ограничений можно возвращать из функции и указатель на структуру.

29.8. Объединения (union)

Объединения позволяют хранить разнотипные данные в одной и той же области памяти. По своей сути объединение описывает переменную, которая может иметь любой тип из некоторого множества типов. Другими словами, объединение – это способ по-разному обратиться к одной и той же области памяти.

Определение объединения аналогично определению структуры:

```
union имя_объединения {
  описания_элементов;
};
```

При объявлении объединения для него описывается набор типов значений, которые могут с ним ассоциироваться. В каждый момент времени объединение интерпретируется как значение только одного типа из набора. Контроль над тем, какого типа значение хранится в данный момент в объединении, возлагается на программиста. Память, которая выделяется для объединения, определяется размером наиболее длинного из его элементов. Все элементы объединения размещаются с одного и того же адреса памяти. Значение текущего элемента объединения теряется, когда другому элементу присваивается значение.

Доступ к полям объединения выполняется или через ‘`.`’, или через ‘`->`’, если для доступа используется указатель:

```
имя_объединения.элемент
```

или

```
указатель_на_объединение->элемент
```

Пример объединения, которое можно интерпретировать как знаковое или беззнаковое целое число:

```
union number {
  int svar;
  unsigned int uvar;
};

number x = {100};     // описание переменной типа
number *p;            // можно описывать указатели на union
number y[10];         // можно описывать массивы

union {               // можно и так описывать переменные,
  int svar;           // без определения имени union
  unsigned int uvar;
} numb;
```

Инициализировать объединение можно только значением, имеющим тип его первого элемента (например, переменную `x` можно инициализировать лишь значением типа `int`).

Значение одного из этих двух заданных типов может быть присвоено переменной `x` и далее использовано в выражениях, если это правомерно, т.е. если тип взятого из нее значения совпадает с типом последнего присвоенного ей значения. Выполнение этого требования в каждый текущий момент – задача программиста. В том случае, если нечто запомнено как значение одного типа, а извлекается как значение другого типа, результат зависит от реализации.

Объединения могут входить в структуры и массивы, и наоборот. Запись доступа к элементу объединения, находящегося в структуре (как и структуры, находящейся в объединении), такая же, как и для вложенных структур.

Например, в массиве структур

```
struct {
  char *name;
  int flags;
  int type;
  union {
    int ival;
    float fval;
    char *cval;
  } u;
} xxx[10];
```

к `ival` `i`-го элемента массива обращаются так: `xxx[i].u.ival`.

Фактически объединение – это структура, все элементы которой имеют нулевое смещение относительно ее базового адреса, и размер которой позволяет поместиться в ней самому большому ее элементу. Операции, применимые к структурам, годятся и для объединений, т.е. законны присваивание объединения и копирование его как единого целого, получение адреса объединения и доступ к отдельным его элементам.

Пример работы с объединением:

```
void main() {
  union number {
    int svar;
    unsigned int uvar;
    char cvar[2];
  };
  // аппаратные особенности IBM PC – размещение числовых кодов в памяти,
  // начиная с младшего адреса (пары младших разрядов шестнадцатиричного
  // числового кода размещаются в байтах памяти с меньшими адресами, т.е.
  // при хранении в памяти целых чисел для int байты переставляются,
  // для long переставляются сначала по 2 байта, а затем уже байты
  // int:  1б 2б       => в памяти 2б 1б
  // long: 1б 2б 3б 4б => в памяти 4б 3б 2б 1б
  // 50₁₀ = 32₁₆ = 0x32
  number x = {50};           // {50, 50, "0x32 0x0" ("2")}
  char ss[50];
  strcpy(ss, x.cvar);        // ss = "2"
  x.svar = -100;             // {-100, 65436, "0x9C 0xFF"}
  unsigned int xx = x.uvar;  // xx = 65436
  x.svar >>= 1;              // {-50,   65486, "0xCE 0xFF"}
  // 1111 1111 1001 1100 => 1111 1111 1100 1110 (размножение знака)
  x.svar = -100;             // {-100,  65436, "0x9C 0xFF"}
  x.uvar >>= 1;              // {32718, 32718, "0xCE 0x7F"}
  // 1111 1111 1001 1100 => 0111 1111 1100 1110
}
```

Применение объединений очень сильно ограничено (обработка ввода с клавиатуры, обработка кодов нажатых клавиш и т.п.).

30. ДИНАМИЧЕСКАЯ ПАМЯТЬ

30.1. Понятие динамического объекта

Данные, которые создаются и уничтожаются по требованию программиста, называются динамическими.

Чтобы выделить память под обычную переменную, ее надо описать: `int a;`. Обратиться к такой переменной можно или по имени (`a=10;`), или через указатель, в который занесен адрес переменной (`int *pa=&a; *pa=20;`). Количество таких переменных фиксировано – ровно столько, сколько их определено в тексте программы.

Динамические объекты создаются динамически в процессе выполнения программы. Число динамических объектов не фиксировано тем, что записано в тексте программы, – динамические объекты могут создаваться и уничтожаться в процессе ее выполнения. Динамические объекты не имеют имен, и ссылка на них выполняется только с помощью указателей, в которых хранятся адреса динамических объектов. Т.о указатели используются при создании и обработке динамических объектов.

Время жизни именованного объекта (обычной переменной) определяется его областью видимости. С другой стороны, часто возникает необходимость в создании объектов, которые существуют вне зависимости от области видимости, в которой они были созданы. Типичным примером является создание объектов, которые используются после возвращения из функции, в которой они были созданы. Еще одна причина использования динамических объектов – оптимизация использования памяти (например, выделение памяти под массив ровно такого размера, который нужен пользователю).

Память под динамические объекты выделяется в куче (`heap`). Существуют такие объекты с момента создания и до момента уничтожения (явно или по концу работы программы). Доступ к динамическим объектам возможен из любого места программы (понятно, что в этом «любом месте» должен быть известен адрес динамического объекта). Количество динамических объектов, одновременно существующих в программе, ограничивается только наличием доступной свободной памяти (размером кучи). Говорят, что место под динамические объекты выделяется из «свободной памяти» (такие объекты называют еще «объектами из кучи» или «объектами, размещенными в динамической памяти»).

30.2 Создание и уничтожение динамических объектов

Стандартная библиотека языка С предоставляет функции для управления динамической памятью. Эти функции позволяют динамически запрашивать из программы дополнительные области памяти, а также освобождать ставшие ненужными запрошенные области.

Работа функций динамического распределения памяти различается для различных *моделей памяти*, поддерживаемых системой программирования. Происходит это из-за разных способов организации и размеров куч в малых моделях памяти (`tiny`, `small`, `medium`) и в больших моделях памяти (`compact`, `large`, `huge`).

Прототипы функций для работы с динамической памятью содержатся в файле `<alloc.h>`, а также в файле `<stdlib.h>`.

Функции `malloc()` и `calloc()` служат для динамического запроса блоков свободной памяти, т.е. для создания динамических объектов. Эти функции возвращают адрес созданного динамического объекта.

```
void *malloc(unsigned int size);
```

Выделить память объемом `size` байт. Функция возвращает указатель на выделенную память или `NULL`, если запрос удовлетворить нельзя. Значение `NULL` возвращается и в том случае, когда значение параметра `size` нулевое. Выделенная память никак не инициализируется (содержит «мусор» – случайные значения).

```
void *calloc(unsigned int n, unsigned int size);
```

Выделить память под `n` элементов по `size` байт каждый. Функция возвращает указатель на выделенную память или `NULL`, если запрос не удается удовлетворить. Выделенная память обнуляется.

Обратить внимание! В модели `Large` (установлена по умолчанию в классе 1-1) параметры функций имеют тип `unsigned long`.

Для определения необходимого объема памяти желательно использовать оператор `sizeof`.

Функции возвращают указатель типа `void *`, поэтому при вызове функций в программе необходимо применять операцию приведения к соответствующему типу:

```
char *ps;
int *pi;

ps = (char *)malloc(100);
if (ps == NULL) return;

pi = (int *)calloc(1, sizeof(int));
if (pi == NULL) return;
```

Обратить внимание! Всегда надо проверять в программе, успешно ли выделилась память.

Функция `realloc()` служит для изменения размера ранее выделенного блока памяти, адрес которого содержится в указателе `block`. Параметр `size` задает новый размер блока. Функция гарантирует сохранность старых данных в блоке (понятно, что сохранность не более `size` байтов).

```
void *realloc (void *block, unsigned int size);
```

Функция возвращает указатель на перезахваченный блок памяти. Блок может быть передвинут, если его размеры изменены, поэтому аргумент `block` не обязательно должен быть таким же, как и возвращаемое значение. Возвращается значение `NULL`, если памяти недостаточно, чтобы расширить блок к заданному размеру. Если это происходит, то первоначальный блок освобождается.

```
char *ps;
// захватывает блок памяти для 50 символов
ps = malloc(50 * sizeof(char));
if (ps == NULL) return;
// перезахватывает блок для 100 символов
ps = realloc(ps, 100 * sizeof(char));
if (ps == NULL) return;
```

Обратить внимание! Все рассмотренные функции могут выделять память размером не более одного сегмента, то есть не более 64K в 16-ти разрядных моделях и не более 4G в 32-х разрядных моделях памяти. При работе с динамической памятью следует иметь в виду, что в каждом выделенном блоке несколько байт отводится на служебную информацию. Поэтому последствия затирания динамической памяти могут быть существенными.

К динамическим объектам нельзя обратиться по имени, так как они просто не имеют имени. У нас есть только указатель с адресом динамического объекта. Поэтому для доступа к самому динамическому объекту используется операция разименования *:

```
int *pi;
pi = (int *)calloc(1, sizeof(int));
if (pi == NULL) return;
*pi = 55;
```

Одно и то же значение может быть присвоено более чем одной переменной-указателю. Таким образом, можно ссылаться на динамический объект с помощью более одного указателя.

Объект, созданный при помощи функций динамического выделения памяти, существует или до тех пор, пока он не удален явно при помощи функции освобождения памяти, или до конца работы программы. После овобождения память, занимаемая объектом, может быть снова использована при следующих запросах памяти.

В языке С нет так называемого «сборщика мусора» (есть, например, в Java), осуществляющего поиск объектов, на которые отсутствуют ссылки, и делающего эту память доступной для повторного использования. Вот поэтому очень желательно всю захваченную в программе память освобождать явно, несмотря даже на то, что она неявно автоматически освобождается по концу работы программы. Бывают программы, которые работают длительно (так называемый режим работы «24/7»). Конечно, доступной для выделения памяти в современных компьютерах много, но все-таки ее не бесконечно много. И когда-нибудь она может и закончиться. Еще одна причина для явного освобождения памяти в программе – так называемая «утеч-

ка» памяти, когда происходит потеря памяти, т.е. становится невозможным ее повторное использование (например, в результате потери адреса выделенной области).

Для явного освобождения памяти используется функция `free()`.

```
void free(void *block);
```

Функция освобождает область памяти, на которую указывает block. В указателе должен содержаться адрес области памяти, полученной с помощью `malloc()`, `calloc()` или `realloc()`. Нельзя освобождать области, которые не были получены с помощью этих функций. Нельзя также повторно освобождать те области памяти, которые уже освобождены. Вызов `free(p)` для `p=NULL` не вызывает никаких действий со стороны функции `free()`.

После освобождения считается, что память больше не принадлежит программе. Поэтому использование области памяти, которая уже освобождена с помощью `free()` (так называема проблема «висящих» ссылок), является ошибкой.

```
fpee(p);
p = p->next;  // ошибка!!!
```

Задача. Разработать функцию для нахождения минимума из двух целых чисел и вызвать ее из функции main(). Все переменные в программе должны быть переменными-указателями. Формальные параметры функции могут быть обычными переменными.

```
void  fmin(int a, int b, int *min) {
  *min = a;
  if (b < *min)
    *min = b;
}

void main() {
  int *a, *b, *m;       // все переменные – только указатели
  a = (int *)malloc(sizeof(int)); //выделение памяти под a
  b = (int *)malloc(sizeof(int)); //выделение памяти под b
  m = (int *)malloc(sizeof(int)); //выделение памяти под c
  printf("Введите два числа: ");
  scanf("%d %d", a, b);
  printf("a=%d  b=%d\n", *a, *b);
  fmin(*a, *b, m);
  printf("min=%d\n", *m);
  free(a); free(b); free(m);
}
```

30.3 Динамическое размещение одномерных массивов и строк

Функции для работы с динамической памятью позволяют использовать массивы с границами, задаваемыми переменными, а не константами.

При динамическом распределении памяти для одномерных массивов следует описать указатель соответствующего типа и присвоить ему значение при помощи фунций `malloc()` или `calloc()`.

Одномерный массив из `n` целых чисел можно создать двумя способами:

```
int *a, *b;
int n = 10;

a = (int *)malloc(n * sizeof(int));
b = (int *)calloc(n, sizeof(int));     // инициализация 0
```

Освобождение памяти, выделенной под массивы:

```
free(a);
free(b);
```

Обратить внимание! Адреса освобождаемых массивов должны строго совпадать с адресами, которые вернули функции динамического размещения массивов.

```
int *a, n = 10;
a = (int *)malloc(n * sizeof(int));
a += 2;       // перешли к элементу массива с индексом 2
*a = 100;     // a[2] = 100
free(a);      // ошибка, т.к. a != адрес начала массива

int *a, *p, n = 10;
a = (int *)malloc(n * sizeof(int));
p = a;        // сохранение адреса начала массива
a += 2;
*a = 100;
free(p);      // ошибки нет
```

Задача. Пример программы для работы с динамическим одномерным массивом целых чисел.

```
// выделение памяти и ввод массива
int vvod(int **m) {
  int nn;
  scanf ("%d", &nn);
  *m = (int *)calloc(nn, sizeof(int));
  // ввод массива
  for (int i=0; i<nn; i++)
    scanf("%d", *m+i);        // *m+i == &(*m)[i]
  return nn;
}

// поиск минимального элемента в массиве
int fmin(int *m, int nn) {
  int min = 0x7FFF;
  for (int i=0; i<nn; i++)
    if (m[i] < min)
      min = *(m+i);
```

```
  return min;
}
void main() {
  int n, i, min;
  int *b;            // указатель для массива b[n]
  n = vvod(&b);      // передается адрес указателя, а не его значение
  // печать массива
  for (i=0; i<n; i++)
    printf("%d ", b[i]);     // b[i] == *(b+i)
  min = fmin(b, n);
  printf("\nmin = %d\n", min);
  free(b);           // освобождение памяти
}
```

Задача. Пример программы для работы с динамическим одномерным массивом структур: создание массива, ввод, вывод на печать, поиск книги с минимальной стоимостью, освобождение памяти.

```
void main() {
  struct Book {
    char name[20];
    int cost;
  };
  struct Book *p, min;
  p = (struct Book *)malloc(5 * sizeof(struct Book));
  for (int i=0; i<5; i++)
    scanf("%s %d", (p+i)->name, &(p+i)->cost);
  for (i=0; i<5; i++)
    printf("%s %d\n", p[i].name, p[i].cost);
  min = p[0];
  for (i=1; i<5; i++)
    if (min.cost > (p+i)->cost)
      min = *(p+i);
  printf("min: %s %d\n", min.name, min.cost);
  free(p);
}
```

Выделение памяти под динамические строки ничем не отличается от выделения памяти под одномерные массивы других типов, только не следует при этом забывать про ноль-символ.

Задача. Пример программы для работы с динамической строкой.

```
char *strsave(char *s) {
  char *p = NULL;
  p = (char *)malloc(strlen(s) + 1);
  if (p != NULL) strcpy(p, s);
  return p;
```

```
}
void main() {
  char s1[80], *s2;
  gets(s1);
  s2 = strsave(s1);
  if (s2 != NULL) {
    puts(s2);
    free(s2);
  }
}
```

30.4 Динамическое размещение двумерных массивов

При работе с динамическими двумерными массивами возникают определенные трудности, связанные с тем, что в языке С нет встроенных средств для учета длины строки двумерного массива при индексации. Поэтому программист сам должен обеспечить возможность индексации двумерного массива.

Есть два стандартных подхода к динамическому созданию двумерного массива: работа с двумерным массивом как с одномерным и имитация устройства двумерного массива как массива массивов. Рассмотрим оба эти подхода.

В примере ниже двумерный массив представляется в виде одномерного, а местоположения каждого элемента двумерного массива в одномерном определяется суммой номера столбца и произведения номера строки на длину строки.

Задача. Пример программы для работы с динамическим двумерным массивом как с одномерным.

```
void vvodMatr (int *matr, int n, int m) {
  int i, j;
  for(i=0; i<n; i++)
    for(j=0; j<m; j++)
      scanf("%d", &matr[i*m+j]); // matr+i*m+j==&matr[i*m+j]
}
void main() {
  int n, m, i, j;
  int *a;           // указатель для массива a[n][m]
  scanf("%d %d", &n, &m);
  // выделение памяти под матрицу a[n][m]
  a = (int *) malloc(n * m * sizeof(int));
  if(!a) {          //  a == NULL
    printf("Недостаточно памяти!\n"); return;
  }
  vvodMatr(a, n, m);
  printf("\nМатрица --------------\n");
  for(i=0; i<n; i++) {
    for(j=0; j<m; j++)
      printf("%5d ", *(a+i*m+j));  // a[i*m+j] == *(a+i*m+j)
```

```
    printf("\n");
  }
  free(a);
}
```

Следующий вариант представления динамического двумерного массива позволяет использовать привычную индексацию двумерного массива. Массив представляется в виде динамического одномерного массива указателей на строки двумерного массива. Для создания двумерного массива вначале нужно выделить память под массив указателей на строки (на одномерные массивы), а затем каждый из этих указателей связать с динамически выделенными областями, куда запишутся элементы строк (одномерных массивов).

Задача. *Пример программы для работы с динамическим двумерным массивом как с обычным двумерным массивом.*

```
void vvodMatr (int *matr[], int n, int m) {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      scanf("%d", &matr[i][j]);// &matr[i][j] == *(matr+i)+j
}

void freeMatr (int *matr[], int n) {
  // освобождение строк матрицы
  for(int i=0; i<n; i++)
    free(matr[i]);                  // matr[i] == *(matr+i)
  // освобождение массива указателей
  free(matr);
}

void main() {
  int n, m, i, j;
  int **a;                        // матрица a[n][m]
  scanf("%d %d", &n, &m);
  // выделение памяти под матрицу
  // выделение памяти под n указателей на строки матрицы
  a = (int **) malloc( n * sizeof(int *) );
  if(!a) { printf("Недостаточно памяти!\n"); return; }
  // выделение памяти под строки матрицы
  for(i = 0; i < n; i++) {
    a[i] = (int *) malloc( m * sizeof(int) );
    if(!a[i]) {
      printf("Недостаточно памяти!\n");
      freeMatr(a, i);
      return;
    }
  }
  vvodMatr(a, n, m);
  printf("\nМатрица---------------\n");
```

```
  for(i = 0; i < n; i++) {
    for(j = 0; j < m; j++)
      printf("%4d ", a[i][j]);      // a[i][j] == *(*(a+i)+j)
    printf("\n");
  }
  freeMatr(a, n);
}
```

30.5. Функции для работы с блоками памяти

Данные функции упрощают работу с блоками информации в памяти. Блок – это подряд расположенные байты оперативной памяти. Функции используют наиболее эффективные машинные команды – так называемые цепочечные примитивы. Поэтому эти функции очень быстро работают с большими блоками памяти. Все функции используют указатель на начало блока памяти как параметр. Функции для блоков похожи на функции для строк. Отличие – функции для строк концом блока информации считают `'\0'`, а эти функции всегда ориентируются явно на длину блока. При работе с этими функциями надо аккуратно резервировать память под буфер-назначение.

Прототипы функций для работы с блоками памяти содержатся в заголовочном файле `<mem.h>`, а также в `<string.h>`.

Рассмотрим основные функции (подробно все можно найти в Касаткине).

Копирование блоков:

```
void *memcpy(void *d, void *s, unsigned int n);
```

Копируется `n` байт из блока, на начало которого указывает `s`, в другое место памяти, на начало которого указывает `d`. Блоки не должны пересекаться в памяти. Функция возвращает адрес блока-назначения.

Копирование блоков с условием:

```
void *memccpy(void *d, void *s, int c, unsigned int n);
```

Копируется блок, на начало которого указывает `s`, в другое место памяти, на начало которого указывает `d`. Копирование продолжается до тех пор, пока не произойдет одно из двух событий: функция встретит в блоке-источнике байт, содержащий символ `c`, который тоже перенесется в блок-назначение; или общее число скопированных байтов достигнет значения `n`. Если происходит первое событие, функция возвращает указатель на следующий после символа `c` байт в блоке-назначении, иначе возвращает `NULL`.

Пересылка блоков:

```
void *memmove(void *d, void *s, unsigned int n);
```

Копируется `n` байт из блока, на начало которого указывает `s`, в другое место памяти, на начало которого указывает `d`. Блоки могут пересекаться в памяти. Функция возвращает адрес блока-назначения.

Сравнение двух блоков:

```
int memcmp(void *s1, void *s2, unsigned int n);
```

Сравнивает n первых байтов двух блоков, на начало которых указывают s1 и s2. Функция возвращает значение больше нуля, если s1 > s2, меньше нуля, если s1 < s2, и равное нулю, если s1 == s2.

Занесение символа n раз в буфер:

```
void *memset(void *s, int c, unsigned int n);
```

Устанавливает n байтов буфера, на начало которого указывает s, в заданное значение c. Функция возвращает указатель на блок s.

Поиск символа в блоке:

```
void *memchr(void *s, int c, unsigned int n);
```

Просматривается n байтов блока, на начало которого указывает s, сопоставляя каждый байт с кодом символа c. Если совпадение найдено, функция возвращает указатель на этот байт с символом c в буфере, иначе возвращает NULL.

31. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

31.1. Понятие структуры данных

При решении любой задачи возникает необходимость работы с данными и выполнения операций над ними. Набор этих операций для каждой задачи, вообще говоря, свой. Однако, если некоторый набор операций часто используется при решении различных задач, то полезно придумать способ организации данных, позволяющий выполнять именно эти операции как можно эффективнее. После того, как такой способ придуман, при решении конкретной задачи можно считать, что у нас в наличии имеется «черный ящик» (его мы и будем называть структурой данных), про который известно, что в нем хранятся данные некоторого рода, и который умеет выполнять некоторые операции над этими данными. Это позволяет отвлечься от деталей и сосредоточиться на характерных особенностях задачи. Внутри этот «черный ящик» может быть реализован различным образом, при этом следует стремиться к как можно более эффективной (быстрой и экономично расходующей память) реализации.

Решение многих задач предполагает выделение структур сложного типа с их последующей реализацией средствами выбранного языка программирования. Под *структурой данных* понимают совокупность элементов фиксированных типов и набор базовых операций, определяющих организацию и способ обработки данных. Для каждой структуры характерен свой собственный набор операций.

Структуры данных могут быть статическими и динамическими. Ранее мы рассматривали структуры данных фиксированного размера: одномерные и двумерные массивы, структуры. Основное отличие динамических структур от статических заключается в возможности менять в ходе работы количество содержащихся в структуре данных элементов.

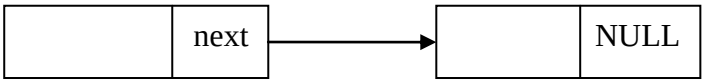
31.2. Структуры, ссылающиеся на себя

Структуры, ссылающиеся на себя, содержат в качестве элемента указатель, который ссылается на структуру того же типа.

Пример:

```
struct Student {
  char fio[80];
  int year;
  struct Student *next;
};
```

Структура типа `struct Student` содержит указатель `next`, который указывает на структуру того же типа `struct Student` (отсюда и термин «структура, ссылающаяся на себя»). Указатель `next` называют связкой, так как его используют для того, чтобы связать структуру типа `struct Student` с другой структурой того же типа. Структуры, ссылающиеся на себя, могут связываться вместе для образования полезных структур данных, таких как списки, очереди, стеки и деревья.



Первая структура содержит в указателе `next` адрес второй структуры (ссылается на нее). Во второй структуре `next=NULL` служит признаком того, что вторая структура ни на что не ссылается.

Создание и использование динамических структур данных требует динамического распределения памяти – возможности получать в процессе выполнения программы дополнительную память для хранения новых элементов структур данных и освобождать блоки памяти, ставшие ненужными, после удаления элементов.

31.3. Связанные списки

Понятие списка хорошо известно из жизненных примеров: список студентов группы, список призёров олимпиады, список документов для представления в приёмную комиссию, список литературы для самостоятельного чтения и т.п. Важной структурной особенностью списка является то, что его элементы линейно упорядочены в соответствии с их позицией в списке.

Связанный список (или просто список) – это структура данных, в которой элементы следуют в некотором порядке. Однако, в отличие от массива, этот порядок определяется указателями, связывающими элементы списка в линейную цепочку. Стеки и очереди – это специальные разновидности связанных списков.

Список – это динамическая структура данных. Длина списка при необходимости может увеличиваться или уменьшаться. Размер списка может увеличиваться до тех пор, пока имеется свободная память.

Списки представляют собой удобную структуру данных для решения многих практических задач. Они используются в программах информационного поиска, трансляторах, при моделировании различных процессов. В виде списков удобно представлять большие объемы информации, размер которых заранее неизвестен.

Достоинство организации данных в виде линейного связанного списка в том, что нет ограничения на длину списка и эффективно используется память (используется ровно столько памяти, сколько надо плюс накладные расходы – поля адресов). Недостатки – необходимость хранить дополнительную информацию (поля адресов) и отсутствие прямого доступа к `i`-му элементу списка, как в массивах по индексу (для получения доступа к определенному элементу списка надо всегда просматривать список с начала или с конца). По этой причине в задачах, где списки меняются сравнительно редко, а просмотр элементов осуществляется часто, выгоднее использовать массивы. В то же время, операции вставки и удаления для связанных списков требуют меньше действий, чем для массивов, так как не требуется перемещать элементы, следующие за вставляемым (или удаляемым). Поэтому использование связанных списков больше подходит для задач, в которых изменения в списках происходят часто.

В языке С нет встроенных типов данных и операций для работы со списками, подобные тем, что имеются для массивов. Поэтому для работы со списками на языке С потребуется их *программная реализация*. Для этого необходимо:

1) сконструировать средствами С структуру данных, которая будет представлять в программе список (в этой структуре будут храниться элементы списка);

2) описать в виде функций требуемые операции над списками.

Для хранения отдельного элемента списка создается динамический объект – структура, состоящая из двух частей: основной, содержащей нужную информацию, и дополнительной, содержащей ссылку на следующий элемент списка. Группировка смысловых полей в отдельную структуру полезна тем, что позволяет быстро сохранять список в файле, а также вводить из файла. Кроме этого упрощается перенос информации между элементами списка. Отметим, что соседние элементы списка располагаются в оперативной памяти произвольно относительно друг друга, в отличие от соседних компонент массива, всегда занимающих смежные участки памяти. Такое расположение элементов облегчает операции вставки и удаления, так как нет необходимости перемещать элементы, как это делается для массивов.

Рассмотрим, например, как можно организовать в виде списка хранение информации о книгах в магазине:

```
typedef struct {     // шаблон данных элемента списка
  char name[80];     // название книги
  int kol;           // количество книг
} InfoBook;
struct List {       // шаблон элемента списка
  InfoBook data;
  struct List *next;
};
```

Доступ к связанному списку обеспечивается через указатель на первый элемент списка. Называется такой указатель головой списка (`head`).

Поэтому программа должна иметь переменную – указатель на первый элемент списка, который равен `NULL`, если список пустой. Имея доступ к первому элементу

списка, без труда можно просмотреть весь список, просто переходя по связям от одного элемента к другому.

```
struct List *head;      // голова списка
```

Для выделения памяти под элементы списка необходимо пользоваться любой из функций:

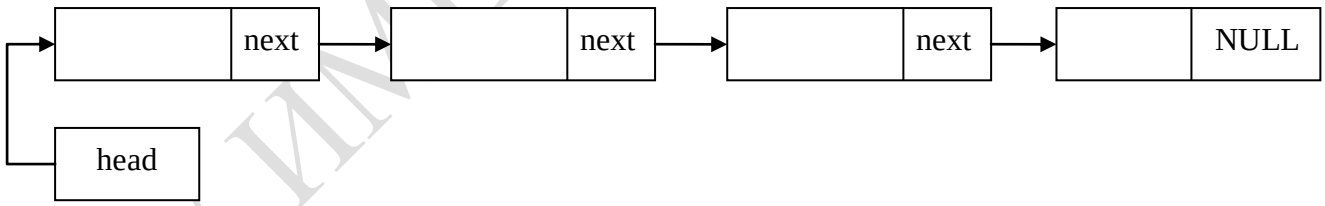
```
malloc(sizeof(struct List))
calloc(1, sizeof(struct List))
```

Для перехода к следующему элементу списка используется его адрес в памяти, который хранится в указателе `next`. В последнем элементе списка указатель на следующий элемент имеет значение `NULL` – это является признаком конца списка.

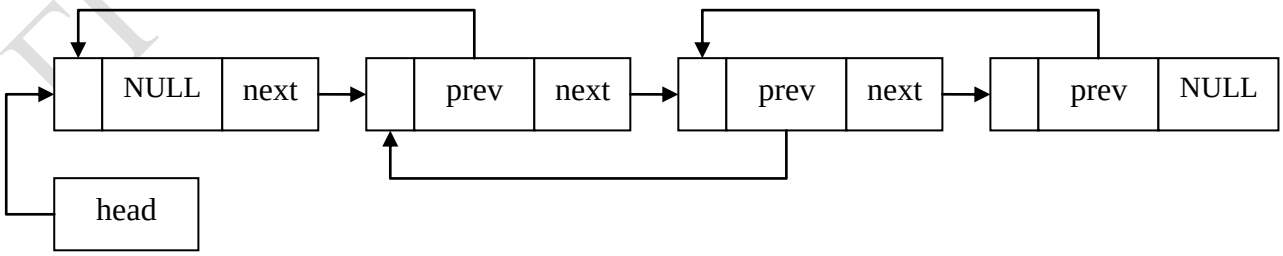
Возможна различная связь данных в списке. Если каждый элемент списка содержит указатель на элемент, следующий непосредственно за ним, то получаемый список называют *односвязным (однонаправленным)*. Если в дополнение к этому каждый элемент списка содержит указатель на элемент, следующий непосредственно перед ним, то такой список называют *двусвязным (двунаправленным)*. Обычно у последнего элемента списка указатель на следующий элемент равен `NULL`, отмечая конец списка. Но в некоторых списках удобно, чтобы этот указатель показывал на первый элемент списка. Таким образом, список из цепочки превращается в кольцо. Такие списки (однонаправленные и двунаправленные) называют *кольцевыми*.

Можно создавать отсортированные списки, если помещать каждый новый элемент списка в соответствующую позицию списка. При этом списки обеспечивают простой механизм вставки и удаления элементов путем модификации указателей, в то время как, вставка и удаление в упорядоченном массиве требует определенного времени на выполнение, так как все элементы, следующие за вставляемым или удаляемым элементом, необходимо соответствующим образом сдвинуть.

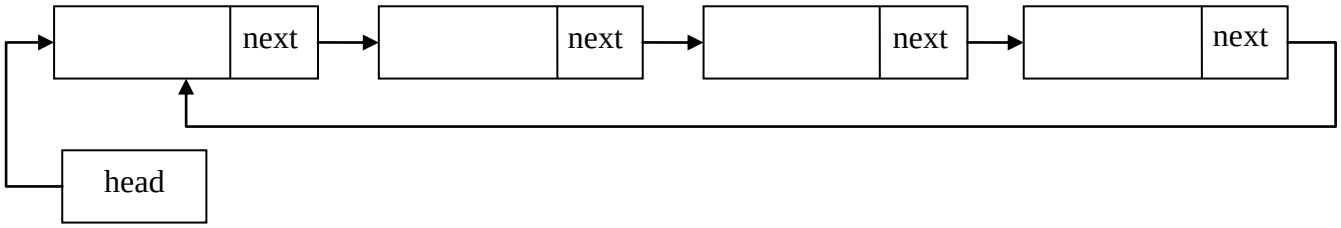
Однонаправленный список:



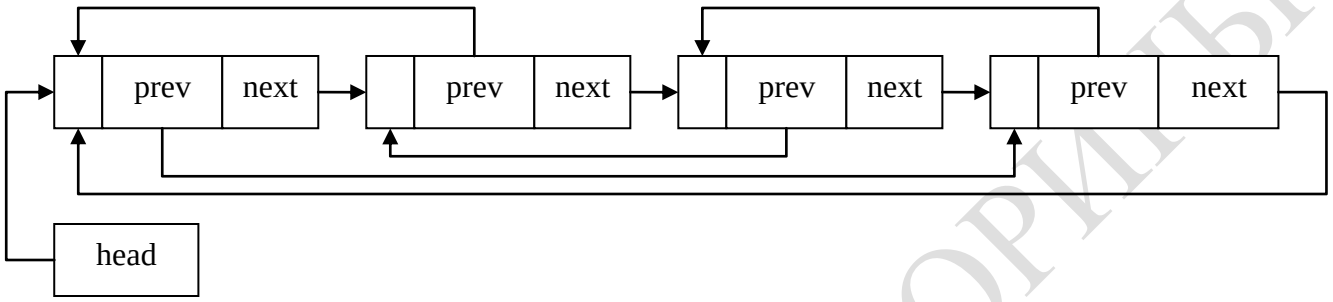
Двунаправленный список:



Кольцевой однонаправленный список:



Кольцевой двунаправленный список:



Двунаправленные списки удобны тем, что позволяют двигаться по списку в обоих направлениях, но, в то же время, работа с такими списками сложнее, так как требуется поддерживать две связи.

При работе со списками на практике чаще всего приходится выполнять следующие операции:

- просмотреть весь список;
- найти один элемент с заданными свойствами;
- вставить новый элемент в список;
- удалить заданный элемент из списка;
- упорядочить список в определенном порядке.

Возможны и более сложные операции над линейными списками – соединить два линейных списка в один список, разбить список на два списка, создать копию списка и т.п.

Рассмотрим основные операции над списками на примере однонаправленного списка.

Просмотр всего списка. Осуществляется линейный просмотр списка от первого до последнего элемента. Просмотр всего списка, например, необходим при печати списка, при удалении всего списка с освобождением всей занятой им памяти, при определении длины списка, при вычислении каких-либо характеристик по данным из всех элементов списка (определить общее количество книг), при поиске элемента списка с заданными характеристиками (не конкретное значение, а максимальное или минимальное – поиск книги, которой больше всего в магазине или подсчет вхождений в список заданного элемента) и т.п.

```
// печать всего списка
void printList() {
  struct List *p = head;  // стать на первый элемент списка
  printf("╔═══════════════════════════════╗\n");
```

```
  printf("╟      Наименование       ┃ Кол-во ╟\n");
  printf("╟─────────────────────────┼────────╫\n");
  while (p != NULL) {  // пока не дошли до конца списка
    printf( "╟ %20s ┃ %6d ╟\n", p->data.name, p->data.kol );
    p = p->next;     // перейти к следующему элементу списка
  }
  printf("╚═════════════════════════╧════════╝\n");
  bioskey(0);
}

// удаление списка с освобождением всей занятой им памяти
// в результате получаем head=NULL: список пустой
void freeList() {
  struct List *p;
  while(head != NULL) {
    p = head->next;   // p = head;
    free(head);       // head = head->next;
    head = p;         // free(p);
  }
}
```

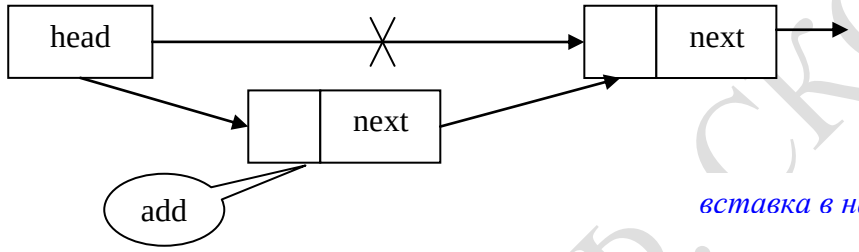
Поиск одного элемента с заданными свойствами. Осуществляется линейный просмотр списка от первого элемента до тех пор, пока не будет найден искомый элемент или же до конца списка, если элемента с заданными свойствами в списке нет. Данная операция используется, например, когда надо получить данные i-го элемента, изменить i-ый элемент или найти один элемент с конкретным значением полей.

```
// есть ли в магазине книга с заданным названием
struct List *findNameList() {
  char s[80];
  puts("Введите название книги для поиска:");
  gets(s);
  struct List *p = head;  // стали на первый элемент списка
  while (p != NULL) {  // пока не дошли до конца списка
    if (strcmp(p->data.name, s) == 0) // нашли совпадение
      break;            // закончили поиск
    p = p->next;     // перешли к следующему элементу списка
  }
  if (p != NULL)  // элемент найден
    printf("Такая книга есть в магазине\n");
  else
    printf("Такой книги нет в магазине\n");
  bioskey(0);
  return p;
}
```

Вставка нового элемента. Вставка новых элементов в список может осуществляться: всегда в начало списка, всегда в конец списка, перед заданным эле-

ментом, после заданного элемента, в нужное место в отсортированном списке. Решение данной задачи состоит из двух этапов. Во-первых, необходимо создать динамический объект для вставляемого элемента списка и занести в него информационные поля. Во-вторых, путём изменения указателей, включить новый элемент в список.

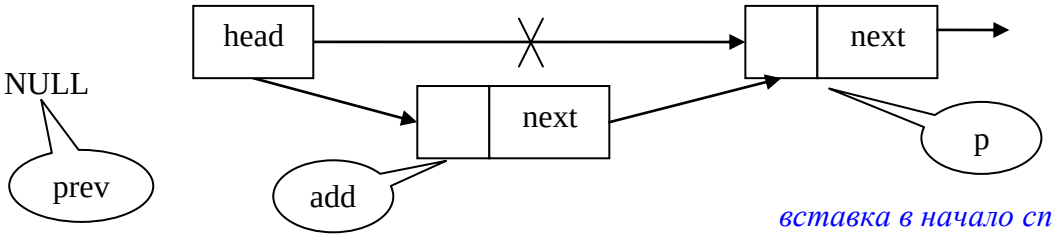
```
// вставка нового элемента в начало списка
struct List *addBegibList(InfoBook a) {
  struct List *add, *p;
  add = (struct List *)malloc(sizeof(struct List));
  if (add) {           // память выделилась
    add->data = a;     // информационная часть
    add->next = head; // новый элемент указывает на прежний первый
    head = add;        // вставка в начало списка
  }
}
```



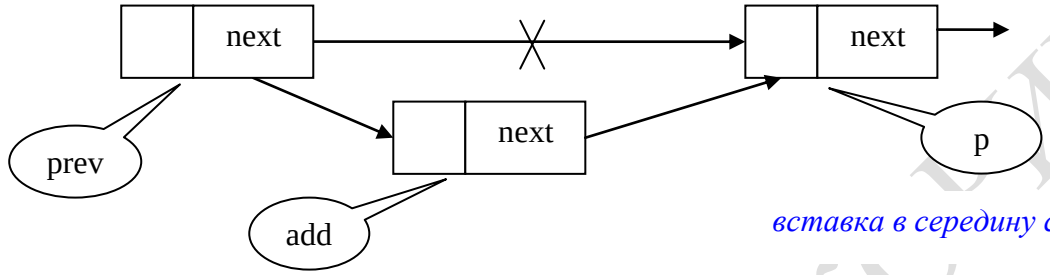
вставка в начало списка

```
// вставка нового элемента перед элементом с заданным
// названием, адрес которого находится в указателе p
struct List * addBeforeList(InfoBook a, char *s) {
  struct List *add, *prev, *p;
  add = (struct List *)malloc(sizeof(struct List));
  if (add) {     // память выделилась
    add->data = a;          // информационная часть
    prev = NULL;            // указатель на элемент перед нужным
    p = head;               // поиск нужного элемента
    while (p != NULL) {
      if (!strcmp(p->data.name, s))
        break;
      prev = p;
      p = p->next;
    }
    if (p == head) {     // вставка в пустой список или в начало списка
      add->next = head;
      head = add;
    }
    else {               // вставка в середину списка или в конец списка
      add->next = p;
      prev->next = add;
    }
```

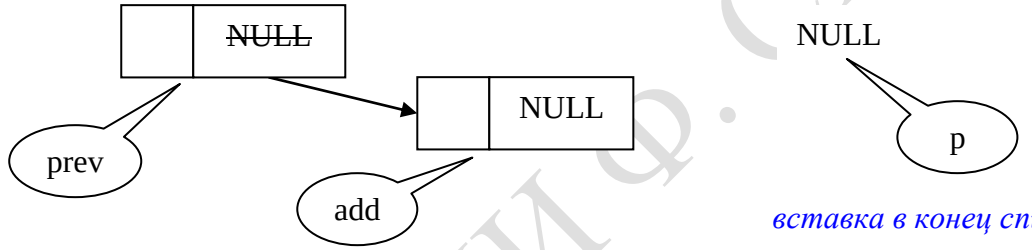
```
    }
}
```



вставка в начало списка



вставка в середину списка



вставка в конец списка

```
// вставка нового элемента в отсортированный по возрастанию
// по количеству кник список
struct List * addSortList(InfoBook a) {
  struct List *add, *p, *prev;
  add = (struct List *)malloc(sizeof(struct List));
  if (add) {
    add->data = a;
    if (head == NULL) {    // список пустой
      add->next = NULL;
      head = add;          // вставка в пустой список
    }
    else {
      p = head;            // поиск элемента, перед которым
      while(p != NULL) {   // надо вставлять новый
        if (add->data.kol < p->data.kol) break;
        prev = p;
        p = p->next;
      }
      if (p == head) {     // вставка в начало списка
        add->next = head;
```

```
        head = add;
      }
      else {
        add->next = p;    // вставка в середину или в конец
        prev->next = add;
      }
    }
  }
  return add;
}
```

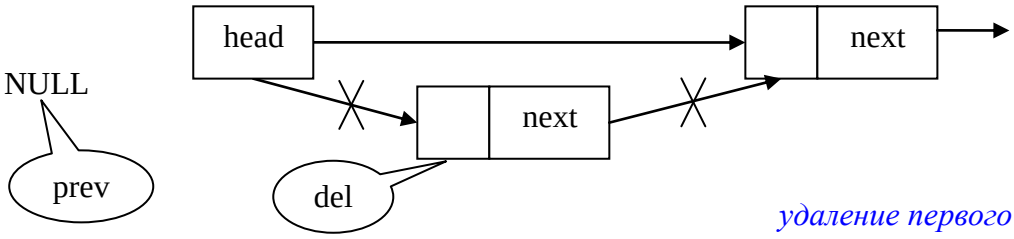
Задача. Пример программы создания и печати списка.

```
typedef struct {    // шаблон данных элемента списка
  char name[80];    // название книги
  int kol;          // количество книг
} InfoBook;

struct List {      // шаблон элемента списка
  InfoBook data;
  struct List *next;
};

struct List *head; // глобальный указатель – голова списка

// вставка нового элемента в конец списка
struct List *addEndList(InfoBook a) {
  struct List *add, *p;
  add = (struct List *)malloc(sizeof(struct List));
  if (add) {  // память выделилась
    add->data = a;          // информационная часть
    add->next = NULL;       // новый элемент будет последним
    if (head == NULL)       // в списке нет элементов
      head = add;           // вставка первого элемента
    else {
      p = head;                 // становимся на начало списка
      while(p->next != NULL) // пока не последний элемент
        p = p->next;    // переходим к следующему элементу
      p->next = add;    // вставка нового следом за последним
    }
  }
  return add;     // адрес нового элемента или NULL
}

void main() {
  InfoBook a;
  int i, n = 5;
  struct List *p;
  // создание списка из 5 элементов
  head = NULL;
```

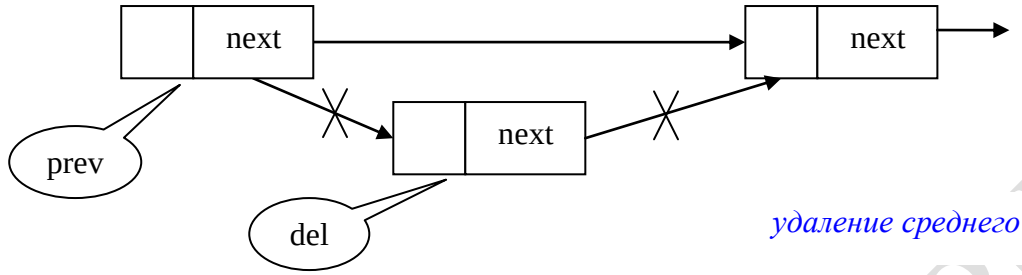
```
  for (i=0; i<n; i++) {
    scanf("%s %d", a.name, &a.kol);
    if (!addEndList(a)) {       //  addEndList(a) == NULL
      printf("Ошибка создания списка\n"); break;
    }
  }
  // печать всего списка
  p = head;          // стать на первый элемент списка
  printf("╔═══════════════════════════════════╤═══════╗\n");
  printf("║      Наименование                 │ Кол-во ║\n");
  printf("╟───────────────────────────────────┼───────╢\n");
  while (p != NULL) {  // пока не дошли до конца списка
    printf( "║ %20s │ %6d ║\n", p->data.name, p->data.kol );
    p = p->next;     // перейти к следующему элементу списка
  }
  printf("╚═══════════════════════════════════╧═══════╝\n");
  bioskey(0);
  // удаление списка с освобождением всей занятой им памяти
  while(head != NULL) {
    p = head->next; free(head); head = p;
  }
}
```

Удаление заданного элемента. Удаление элемента из список может осуществляться: первого элемента в списке, последнего элемента в списке, элемента внутри списка. Решение данной задачи состоит из двух этапов. Во-первых, путём изменения указателей, исключить элемент из списка. Во-вторых, необходимо освободить память, занятую удаляемым элементом.

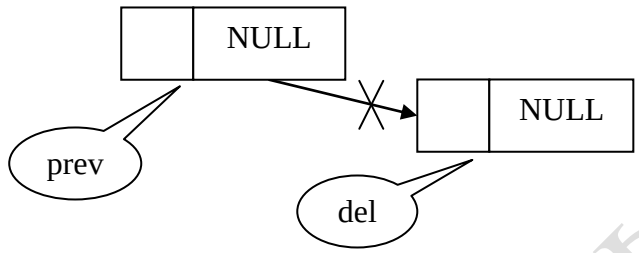
```
// удаление элемента с заданным наименованием
void delList(char *s) {
  struct List *p, *prev = NULL;
  // поиск заданного элемента
  for(p=head; p!=NULL; prev=p, p=p->next)
    if(!strcmp(p->data.name,s)) break;
  if( p!= NULL) {          // элемент найден
    if(p == head)          // удаление первого элемента
      head = p->next;
    else          // удаление среднего или последнего элемента
      prev->next = p->next;
    free(p);               // освобождение памяти
  }
}
```



удаление первого элемента



удаление среднего элемента



удаление последнего элемента

Сортировка списка. Обычно, если требуется иметь отсортированный список, его сразу формируют как отсортированный. В то же время список можно и отсортировать. Надо только четко понимать, что сортировка списка – это или перестройка его связей, или перестановка только информационных полей элементов списка. Данная операция является достаточно сложной в реализации и требует повышенной внимательности.

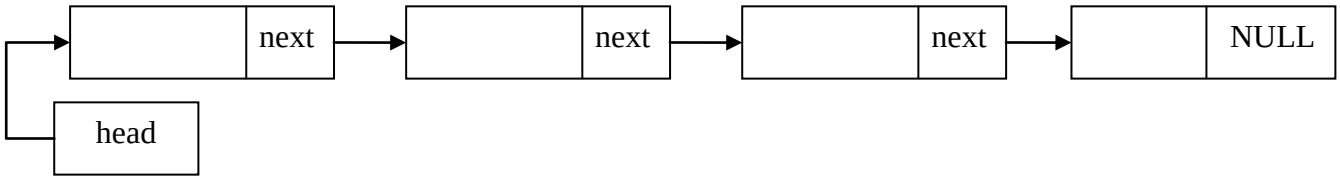
31.4. Стеки

Стеком называется структура данных, добавление или удаление элементов для которой осуществляется с помощью указателя стека в соответствии с правилом `LIFO` (last-in, first-out – последним пришел, первым ушел).

Указатель стека (`head`) содержит в любой момент времени адрес текущего элемента (всегда верхний элемент стека), который является единственным элементом стека, доступным в данный момент времени для работы со стеком.

Основные операции:

1. Добавление элемента в стек – создать новый элемент (выделить для него память и заполнить данные) и поместить его в вершину стека.
2. Удаление элемента из стека – удалить верхний элемент (на него указывает `head`) из стека и освободить память, которая была ему выделена.



Указатель в последнем элементе стека устанавливается равным NULL, чтобы отметить нижнюю границу стека.

Задача. *Пример программы работы со стеком.*

```
typedef struct {     // шаблон данных элемента стека
  char name[80];
  int kol;
} Info;

struct Node {        // шаблон элемента стека
  Info data;
  struct Node *next;
};

struct Node *head;  // вершина стека

// добавление элемента в стек
struct Node *addStack(Info add) {
  struct Node *p;
  p = (struct Node *)malloc(sizeof(struct Node));
  if (p) {            // память выделена успешно
    p->data = add;    // заполнение данных
    p->next = head;   // ссылка на предыдущий элемент стека
    head = p;         // вставка в вершину стека
  }
  return p; // возврат адреса нового элемента или NULL – ошибка добавления
}

// удаление элемента из стека
void delStack() {
  struct Node *p;
  if (head != NULL) {  // стек не пустой
    p = head->next;    // запомнить адрес следующего
    free(head);        // освобождение памяти
    head = p;          // вершина указывает на новый элемент
  }
}

// удаление всех элементов из стека (очистка стека)
// в конце работы функции: head = NULL
void freeStack() {
  struct Node *p;
  while (head != NULL) {  // пока в стеке есть элементы
    p = head->next;
```

```
    free(head);
    head = p;
  }
}
// печать содержимого стека
void printStack() {
  struct Node *p;
  printf("==================\n");
  p = head;              // стать в вершину стека
  while (p != NULL) {
    printf("%s %d\n", p->data.name, p->data.kol);
    p = p->next;         // переход к следующему элементу
  }
}
void main() {
  int i, n = 5;
  Info s;
  head = NULL;    // стек пустой
  for(i=0; i<n; i++) {
    scanf("%s %d", s.name,&s.kol);
    if (!addStack(s)) {
      printf("Ошибка добавления в стек!\n"); break;
    }
  }
  printStack();   //  в стеке 5 элементов
  delStack();
  printf("Вершина стека: %s %d\n",
         head->data.name, head->data.kol);
  delStack();
  printStack();   //  в стеке 3 элемента
  freeStack();
  printStack();   //  в стеке 0 элементов
}
```

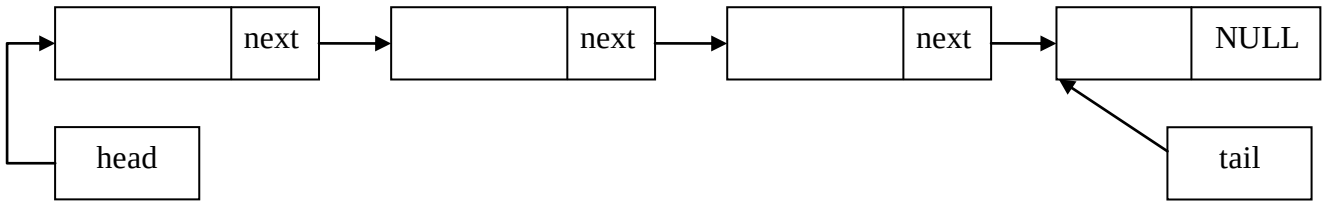
Стеки имеют множество разнообразных применений. Примеры системных задач – стек используется при вызове функций, в том числе и рекурсивном, компиляторы используют стек в процессе вычисления выражений и создания машинного кода (генерации объектной программы). Примеры прикладных задач – построение минимальной выпуклой оболочки.

31.5. Очереди

Очередь – структура данных, для которой удаление или добавление элементов осуществляется с помощью указателей начала (`head`) и конца (`tail`) очереди в соответствии с правилом `FIFO` (first-in, first-out – первым пришел, первым ушел).

Основные операции:

1. Добавление элемента в конец очереди.
2. Удаление элемента из начала очереди.



Ниже приводятся примеры функций для очереди (структура элемента очереди совпадает со структурой элемента стека в примере выше):

```
struct Node *head = NULL;  // начало очереди
struct Node *tail = NULL;  // конец очереди

// добавление элемента в конец очереди
struct Node *addQueue(Info add) {
  struct Node *p;
  p = (struct Node *)malloc(sizeof(struct Node));
  if (p) {                // память выделена успешно
    p->data = add;        // заполнение данных
    p->next = NULL;       // элемент будет последний в очереди
    if (tail != NULL)     // в очереди есть элементы
      tail->next = p;     // вставка в конец очереди
    else                  // вставка первого элемента
      head = p;           // в очереди появился элемент
    tail = p;             // новый конец очереди
  }
  return p; // возврат адреса нового элемента или NULL – ошибка добавления
}

// удаление элемента из начала очереди
void delQueue() {
  struct Node *p;
  if (head != NULL) {  // очередь не пустая
    p = head->next;    // запомнить адрес следующего
    free(head);        // освобождение памяти
    head = p;          // новое начало очереди
    if (head == NULL)  // очередь стала пустой
      tail = NULL;     // очередь не имеет конца
  }
}

// печать содержимого очереди
void printQueue() {
  struct Node *p;
  printf("==================\n");
  p = head;               // стать в начало очереди
  while (p != NULL) {
```

```
    printf("%s %d\n", p->data.name, p->data.kol);
    p = p->next;          // переход к следующему элементу
  }
}
```

Очереди также находят многочисленные применения. Примеры прикладных задач – нахождение кратчайшего пути в графе, нахождение максимального потока в сети (поиск в ширину, в отличие от рекурсии – поиска в глубину). Примеры системных задач – очереди к различным ресурсам в операционных системах, компьютерные сети. Процессор в каждый конкретный момент времени может обслуживать только одну задачу. Остальные задачи ставятся в очередь. Очереди также используются в некоторых алгоритмах замещения страниц при страничной организации памяти. Еще одно применение – очереди к устройствам монопольного доступа (принтер). Информационные пакеты в компьютерных сетях также проводят часть времени, ожидая в очередях.

32. ПРЕПРОЦЕССОР ЯЗЫКА С

Препроцессор используется для обработки текста программы до непосредственной ее компиляции. Компилятор вызывает препроцессор автоматически.

Директивы препроцессора нужны для того, чтобы облегчить написание и модификацию программ, а также сделать их более независимыми от аппаратных платформ и операционных систем. Директивы препроцессора позволяют заменять лексемы в тексте программы, вставлять в файл содержимое других файлов, запрещать компиляцию части файла или делать ее зависимой от некоторых условий и т. д. Хотя препроцессор и расширяет возможности языка программирования, его использование не лишено недостатков: использование препроцессора требует дополнительного просмотра текста программы и, как следствие, добавочного времени.

Директивы препроцессора – это инструкции, записанные непосредственно в исходном тексте программы на языке С. Все директивы препроцессора начинаются с символа #, перед которым в строке могут находиться только пробелы. После директив препроцессора точка с запятой не ставится. Директива препроцессора может занимать одну строку, а может продолжаться и на следующей строке программы, если в конце первой строки поставить символ обратной косой черты (\).

Директивы могут быть записаны в любом месте исходного файла. Их действие распространяется от точки программы, в которой они записаны, и до конца исходного файла или же до явной отмены директивы. Часть директив могут содержать аргументы.

Можно выделить следующие основные виды директив:

- вставка файлов;
- определение макрокоманд (макросов);
- условная компиляция программы.

32.1 Директива включения файлов

Директива `#include` включает в текст программы содержимое указанного файла в ту точку исходного файла, где она записана. Включаемый файл также может содержать директивы `#include`. Эта директива может встречаться в любом месте программы, но обычно все включения размещаются в начале исходного текста. Директива имеет две формы:

```
#include "имя_файла"
#include <имя_файла>
```

Имя файла может состоять либо только из имени файла, либо из имени файла с предшествующим ему путем.

Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным путем, а при его отсутствии – в текущем каталоге.

```
#include "my.h"
```

Если имя файла задано в угловых скобках, то поиск файла производится в стандартных каталогах (для их настройки служит опция `Options→Directories...→Include Directories`).

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

Директива `#include` широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций, и поэтому почти всегда программы на языке С начинаются с этой директивы.

32.2. Директива определения макрокоманд (макросов)

Директива `#define` служит для замены часто используемых констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют *именованными константами*. Идентификаторы, заменяющие фрагменты программ, называют *макрокомандами* (или *макросами*), причем макрокоманды могут иметь аргументы. Макрокоманды и именованные константы предназначены для улучшения восприятия программ и упрощения программирования. Директива имеет две формы:

```
#define идентификатор текст
#define идентификатор(список_параметров) текст
```

Эта директива заменяет все последующие вхождения идентификатора на текст. Такой процесс называется макроподстановкой. После того, как макроподстановка выполнена, полученная строка вновь просматривается для поиска новых макросов. При этом ранее произведенные макроподстановки не принимаются во внимание.

`Текст` может представлять собой любой фрагмент программы на языке С, а также может и отсутствовать.

Именованная константа – это просто имя, которому присваивается постоянное значение (константа). Такая константа в отличие от значения переменной не может изменяться по мере выполнения программы. Кроме того, что именованные

константы делают программу легче для восприятия, они еще и облегчают модификацию программ.

Следующий оператор определяет именованную константу MAX_N как значение 100:

```
#define MAX_N 100
```

Чтобы отличить именованную константу от переменной, обычно используют для именованных констант буквы верхнего регистра. После определения константы, можно использовать ее значение на протяжении всей программы, просто обращаясь к имени значения константы.

Обратить внимание! Определение константы не следует заканчивать точкой с запятой. Если вы поставите точку с запятой в конце определения, препроцессор включит ее в ваше определение. Например, если в директиве для MAX_N поставить точку с запятой после значения 100, то препроцессор в дальнейшем каждый экземпляр константы MAX_N заменит значением 100 с точкой с запятой: 100;.

Макросы могут использоваться для простой замены в тексте программы:

```
#define begin {
#define end }
```

В то же время, макросы с параметрами позволяют создавать что-то похожее на функции: похожее только внешне, но принципиально разное по реализации.

Параметризованный макрос для возведения в квадрат числа:

```
#define SQR(x) ((x)*(x))
```

Происходит замена в тексте всех идентификатор SQR(…) на ((…)*(…)) с подстановкой вместо x того, что записано в скобках. Эта замена носит чисто текстовый характер. Никаких вычислений или преобразований типа при этом не производится.

```
#define SQR(x) ((x)*(x))

void main() {
  int i=10, x;
  x = SQR(i);       // подстановка: x = ((i)*(i));
  x = SQR(2+i);     // подстановка: x = ((2+i)*(2+i));
  x = 1 / SQR(i);   // подстановка: x = 1 / ((i)*(i));
}
```

Большое количество скобок в параметризованных макросах является только на первый взгляд лишним, ведь препроцессор понимает макрос буквально, т.е. каждый формальный параметр будет просто заменен на фактический:

```
#define SQR(x) (x*x)

void main() {
  int i=10, x;
  x = SQR(i);       // подстановка: x = (i*i);
  x = SQR(2+i);     // подстановка: x = (2+i*2+i); ???
```

```
  x = 1 / SQR(i);  // подстановка: x = 1 / (i*i);
}

#define SQR(x) (x)*(x)

void main() {
  int i=10, x;
  x = SQR(i);      // подстановка: x = (i)*(i);
  x = SQR(2+i);    // подстановка: x = (2+i)*(2+i);
  x = 1 / SQR(i);  // подстановка: x = 1 / (i)*(i); ???
}
```

Как видим, макрос с параметрами – это что-то похожее на функцию, но без вызова функции. Функции безусловно «правильней», но требуют от процессора дополнительных затрат на передачу параметров, вызов и возврат. Если функция часто вызывается, эти затраты (т.е. дополнительное время) могут быть велики и тогда полезно заменить функцию макросом.

В то же время, у макроса есть недостаток, который при определенных условиях может превратится в его достоинство, – отсутствие проверки типов аргументов. Функция, возводящая число в квадрат, принимает аргумент определенного типа. Для параметров типа `int` придется писать одну функцию, для `double` – другую. В то же время макрос, возводящий в квадрат, ничего не знает о типе параметра и его можно использовать для *любых* переменных.

Обратить внимание! Определение макроса не является функцией. Каждый раз при вызове функции программа помещает параметры в стек и затем выполняет переход к коду функции. После завершения функции программа удаляет параметры из стека и переходит обратно к оператору, который следует непосредственно за вызовом функции. В случае с макросом препроцессор просто выполняет замену в тексте программы каждой ссылки на макрос соответствующим определением макроса. Поэтому макросы выполняются быстрее функций, но они увеличивают размер выполняемой программы.

Задача. *Разработать макросы для замены значений двух переменных и для нахождения максимума из двух переменных.*

```
#define SWAP(a,b) \
        a = a - b; \
        b = b + a; \
        a = b - a;

#define MAX((a),(b)) ((a)>(b)?(a):(b))
```

Задача. *Разработать макросы для ускорения записи в программе цикла for.*

```
#define FORin for(i=0; i<n; i++)

#define FOR(i,n) for(i=0; i<n; i++)

void main() {
  int m[5] = {1,2,3,4,5}, s, n = 5, j, i, k ;
  s = 0;
```

```
  FORin            // подстановка: for(i=0; i<n; i++)
    s + =m[i];
  s=0;
  FOR(j,n)         // подстановка: for(j=0; j<n; j++)
    s += m[j];
  s=0;
  FOR(k,5)         // подстановка: for(k=0; k<5; k++)
    s += m[k];
}
```

Макросы в программах можно использовать различным образом. Только надо помнить, что цель использования макросов состоит в упрощении кодирования и улучшении восприятия программ.

Одиночный символ **#**, помещаемый перед параметром макроса, указывает на то, что параметр должен быть преобразован в символьную строку, то есть, конструкция вида *#формальный_параметр* будет заменена на конструкцию *"фактический_параметр"*. Например, можно сделать макрос `print` для печати значения переменных в формате «`имя = значение`»:

```
#define print(a) printf(#a " = %d\n",a)
int abc = 10;
print(abc);    // подстановка: printf("abc" " = %d\n", abc);
```

Наконец, возможна директива препроцессора для удаления из текста программы заданного идентификатора:

```
#define DEBUG
```

Препроцессор умеет не только создавать макросы, но и уничтожать их. Для уничтожения макросов используется директива `#undef` вида:

```
#undef идентификатор
```

Например:

```
#define MY_NULL 0
x = MY_NULL;           // подстановка:       x = 0;
#undef MY_NULL
y = MY_NULL;           // нет подстановки:   y = MY_NULL;
```

32.3 Директива условной компиляции

Исходный файл можно компилировать не целиком, а частями, используя директивы условной компиляции:

```
#define LEVEL 2
...
#if LEVEL > 3
  текст1
#elif LEVEL > 1
  текст2
```

```
#else
  текст3
#endif
```

где LEVEL – это макроимя, поэтому выражение в директивах #if и #elif можно вычислить во время обработки исходного текста препроцессором.

Вычисляется константное целое выражение, заданное в строке #if. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до ближайшей директивы #endif, #elif (действует как else if) или #else. Блок условной компиляции должен завершаться директивой #endif.

Получаем, если LEVEL больше 3, то компилироваться будет текст1, если LEVEL больше 1, то компилироваться будет текст2, в противном случае компилируется текст3.

Текст может занимать более одной строки. Он может представлять собой фрагмент программного кода, но может использоваться и для обработки произвольного текста. Если текст содержит другие директивы препроцессора, они выполняются. Обработанный препроцессором текст передается на компиляцию. Все участки текста, не выбранные препроцессором, игнорируются и не порождают компилируемого кода.

Директив #elif может быть несколько (либо вообще ни одной), директива #else также может быть опущена.

Если все выражения, следующие за #if, #elif на данном уровне вложенности ложны (равны нулю), выбирается текст, следующий за #else. Если при этом ветвь #else отсутствует, никакой текст не выбирается.

В каком-то смысле директива #if похожа на условный оператор if. Однако, в отличие от него, условие – это константа, которая вычисляется на стадии препроцессора, и куски текста, не удовлетворяющие условию, просто игнорируются.

Можно использовать специальную препроцессорную операцию defined. Операция defined(идентификатор) дает ненулевое значение, если заданный идентификатор в данный момент определен; в противном случае выражение равно нулю (ложно). Операция может использоваться в сложном выражении и неоднократно:

```
#if defined(name1) || defined(name2)
```

Пример:

```
#if defined (COLOR)
  color();
#elif defined (MONO)
  mono();
#else
  error();
#endif
```

Здесь условная директива управляет компиляцией одного из трех вызовов функции. Вызов функции color() компилируется, если определена именованная константа COLOR. Если определена константа MONO, компилируется вызов функции mono(), если ни одна из двух констант не определена, компилируется вызов функции error().

Чтобы застраховаться от повторного включения заголовочного файла my.h, этот файл можно оформить следующим образом:

```
#if !defined(MY_H)
#define MY_H
// здесь содержимое файла my.h
#endif
```

При первом включении файла my.h будет определено имя MY_H, а при последующих включениях препроцессор обнаружит, что имя MY_H уже определено, и перескочит сразу на #endif. Этот прием полезен, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый заголовочный файл будет сам включать заголовочные файлы, от которых он зависит, освободив от этого занятия пользователя.

Директива #ifdef – модификация условия компиляции. Условие считается выполненным, если указанное после нее макроимя определено. Соответственно, для директивы #ifndef условие выполнено, если имя не определено.

```
#define DEBUG 1
...
#if DEBUG
  printf("%d", x);
#endif
```

32.4 Дополнительные директивы препроцессора

Указания компилятору или *прагмы* предназначены для выдачи дополнительных указаний компилятору. Например, не выдавать предупреждений при компиляции, или вставить дополнительную информацию для отладчика. Они имеют общий синтаксис вида:

```
#pragma текст
```

где текст задает определенную инструкцию, возможно, имеющую аргументы.

Конкретные возможности директивы #pragma у разных компиляторов различные.

Директива #error выдает сообщение и завершает компиляцию. Например, следующая конструкция выдаст сообщение и не даст откомпилировать исходный файл, если макроимя unix не определено:

```
#ifndef unix
  #error "Программу можно компилировать только для Unix!"
```

```
#endif
```

Кроме директив, у препроцессора есть одна операция `##`, которая соединяет строки, например `A ## B`.