Московский Государственный Университет им. М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Н.Д.Васюкова , И.В.Машечкин , В.В.Тюляева , Е.М.Шляховая

Краткий конспект семинарских занятий по языку Си.

(учебно-методическое пособие для студентов II курса)

Москва
1999

УДК 681.3.06
ББК 32.973-018.1
В20

В пособии представлен конспект семинарских занятий по программированию на языке Си, проводящихся на факультете ВМиК в поддержку основного курса лекций «Системное программное обеспечение». Рассматриваются основные особенности языка, специфика применения. Пособие рекомендуется для студентов, аспирантов и преподавателей.

УДК 681.3.06
ББК 32.973-018.1

Рецензенты:
доцент В.Г.Баула
доцент А.Н.Терехин

Васюкова Н.Д., Машечкин И.В., Тюляева В.В., Шляховая Е.М.

В20 **“Краткий конспект семинарских занятий по языку Си (учебно-методическое пособие)”.**

Издательский отдел факультета ВМиК МГУ
(лицензия ЛР №040777 от 23.07.96), 1999, - 50 с.

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова

ISBN 5-89407-052-X

Введение.

Пособие подготовлено авторами на основании практического опыта использования и преподавания языка Си на факультете ВМиК. В данном пособии авторы сделали попытку рассмотреть наиболее «тонкие», с точки зрения освоения, возможности языка, которые часто опускаются при изучении или используются не совсем верно. К таким особенностям языка отнесены правила работы с типами данных, работа с указателями, структура программы, видимость и существование переменных и т.п. В то же время, в пособии не рассматриваются многие детали языка, хорошо и полно изложенные в доступной литературе, такие как синтаксис языка, редко используемые конструкции (например, volatile), описание библиотечных функций. При написании пособия авторы часто опирались на сведения из популярной книги Б.Кернигана и Д.Ритчи (Язык программирования Си. М., «Финансы и статистика», 1992г.).

ТЕМА 1. Краткая характеристика языка. Базовые типы данных. Квалификаторы. Переменные и константы. Операторы и операции. Таблица приоритетов операций. Выражения. Приведение типов.

Язык Си – язык программирования высокого уровня, тесно связанный с системой UNIX. Разработан Кеном Томпсоном и Денисом Ритчи в 1972 году. В данном пособии рассматривается стандарт ANSI C, принятый в 1989г.

В Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах. Язык Си обеспечивает разнообразие типов данных. Кроме базовых типов: **char**, **int**, **float** и **double** имеется возможность получать целую иерархию выводимых типов данных из указателей, массивов, структур и объединений.

Базовые типы данных могут быть использованы с квалификаторами: **short**, **long**, **signed** и **unsigned**. Всегда выполняется соотношение – размер short не больше размера int, а int, в свою очередь, не больше размера long.

Имена переменных могут состоять из букв (знак подчеркивания считается буквой) и цифр. Первая литера обязательно буква. Большие и маленькие буквы различаются. Для внутренних имен значимыми являются первые 31 литера, а для внешних – 6 литер.

Все переменные должны быть описаны до использования, например,

```
int a;
long b,c,d;
unsigned char s;
```

```
double m[100]; /* m - массив из 100 элементов
                  типа double, индексы принимают
                  значения от 0 до 99 */
```

К любой переменной в декларации может быть применен квалификатор **const**, указывающий на то, что ее значение далее изменяться не будет (для массива это означает, что ни один из его элементов не будет меняться)

```
const char warnmsg[] = "warning: ";
```

Особенности записи констант: целая константа, например, 123 имеет тип int. Целое значение может также иметь восьмеричное или шестнадцатеричное представление. Восьмеричное начинается с нуля (017), а шестнадцатеричное начинается с 0x или 0X (0x1a). Константа типа long завершается буквой l или L, например, 123L. Беззнаковые константы заканчиваются буквой u или U. Константы типа double имеют десятичную точку или экспоненциальную часть, например, 123.4 или 1.234e2. Константы типа float заканчиваются буквой f или F. Константы типа long double заканчиваются буквой l или L, например, 123.4L.

Константа типа char записывается в виде литеры, обрамленной одиночными кавычками (‘*’, ‘s’). Некоторые литеры записываются с помощью заранее определенных эскейп-последовательностей (‘\n’-новая строка, ‘\t’-табуляция, ‘\a’-сигнал звонок и т.д.). Кроме того, произвольный восьмеричный код можно задать в виде ‘\ooo’, где ooo-одна, две или три восьмеричные цифры или ‘\xhh’, где hh-одна или две шестнадцатеричные цифры.

Строковая константа (строка) это ноль или более литер, заключенных в двойные кавычки (“asdf”). Во внутреннем представлении в конце строки присутствует ‘\0’, поэтому, фактически – это массив типа char из n+1 элементов, где n число значащих символов.

В Си имеются средства описания перечислимого типа, для этого используется ключевое слово **enum.** При описании перечислимого типа полностью задается набор его значений:

```
enum palette{white, black=10, red, green=-1,
   blue};
/* white, black, red, green и blue – константы
перечислимого типа palette */
enum palette color;
/*color – переменная типа palette, ей может быть
присвоена любая из констант этого типа*/
color = white;
```

Именование констант перечислимого типа уникально в пределах области видимости. Данные константы ассоциированы с целым типом Си и могут использоваться везде, где используются константы типа int. По умолчанию, константам перечислимого типа присваиваются последовательные значения (0,1,2, ….). Если для константы указано "= значение", то ей присваивается это значение, а следующая за ней константа получает значения на 1 больше и т.д. Таким образом в примере константы перечислимого типа palette имеют следующие значения:

white	-	0
black	-	10
red	-	11
green	-	-1
blue	-	0

В языке Си определены операции следующих типов:

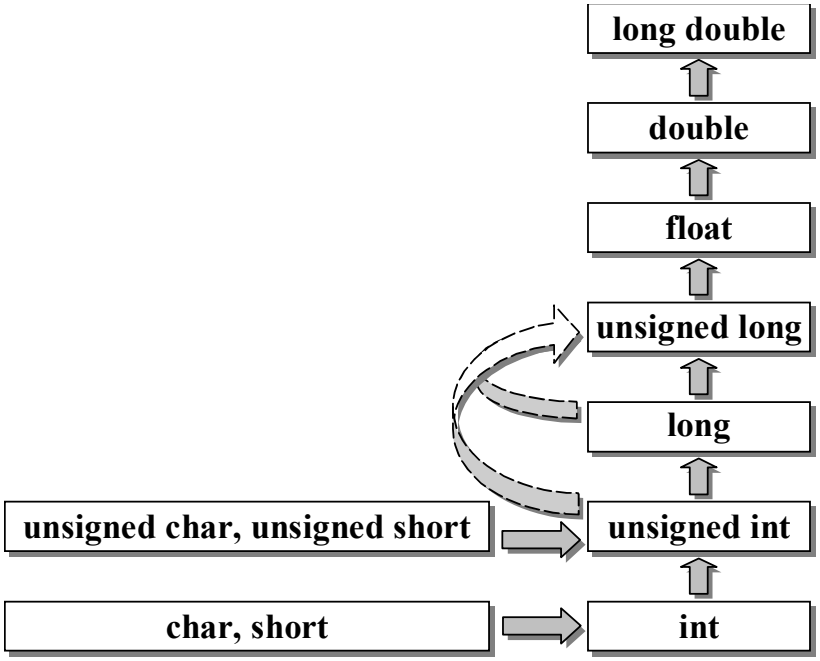
1. Арифметические операции (+, -, *, /, %):
   - унарные операции + и - (изменяет знак стоящей справа величины) имеют более высокий приоритет, чем бинарные операции + и - ;
   - бинарные операции +(сложение) и –(вычитание) имеют одинаковый приоритет, ниже приоритета операций *(умножение), /(деление) и %(остаток от деления);
   - операция / с операндами целых типов выполняется с усечением дробной части;
   - операция % определена только для операндов целых типов;
   - арифметические операции с одинаковым приоритетом выполняются **слева направо.**
2. Логические операции (!, &&, ||)
   - результатом унарной операции ! (логическое НЕ) будет 0 для ненулевого операнда и 1 для нуля;
   - приоритет бинарной операции &&(логическое И) выше приоритета бинарной операции ||(логическое ИЛИ).
3. Операции отношения(<, <=, >, >=, ==, !=)
   - операции <, <=, >, >= имеют одинаковый приоритет, ниже приоритета операций сравнения на равенство == и != ;
   - операции отношения имеют более низкий приоритет, чем арифметические операции, но более высокий, чем логические операции && и ||.
4. Побитовые операции (&, |, ^, <<, >>, ~)
   - все побитовые операции можно применять только к операндам целых типов (знаковым и беззнаковым char, shot, int и long);

- результат поразрядных операций & и | не совпадает с результатом логических операций && и ||, например, для x=2 и y=5 результатом x&y будет 0, тогда как результатом x&&y будет 1;
- операции <<(сдвиг влево) и >>(сдвиг вправо) могут использоваться как эффективное умножение и деление на степени 2: x<<n умножает x на 2 в n-ой степени, а x>>n делит x на 2 в n-ой степени, если число неотрицательное.

5. Операции присваивания (=,+=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=)
   - типом и значением любой операции присваивания является тип и значение ее левого операнда после завершения присваивания.
6. Инкрементная (++) и декрементная(--) операции
   - могут быть как префиксными, так и постфиксными, так например, ++x увеличивает x на 1 до того, как его значение будет использовано, а x++ - после;
   - эти операции можно применять только к переменным: запись ++(x+y) не верна.
7. Условная(тернарная) операция (?:)
   - в выражении x?y:z первым вычисляется значение выражения x, и если он не нуль, то результатом считается значение выражения y, иначе – значение выражения z.

Полная таблица операций с правилами их выполнения приведена в *Приложении 1*.

В выражении могут быть использованы операнды различных типов. При вычислении любой бинарной операции над операндами различных типов предварительно осуществляется автоматическое приведение к объемлющему типу в соответствии со следующей схемой:



Примечание. Читать схему следует следующим образом: «если один из операндов имеет тип long double, то другой операнд приводится к long double» и т.д. Особого внимания требует случай, когда один из операндов имеет тип long, а другой – unsigned int: если представление типа long включает все значения типа unsigned int, то unsigned int приводится к long, если же это не так – оба операнда приводятся к unsigned long (эта особенность вызвана тем, что стандарт языка Си не накладывает ограничений на машинное представление переменных, кроме того, что машинное представление переменной long должно быть не короче машинного представления переменной int).
При выполнении операции присваивания тип правой части приводится к типу левой, например:

```
...int x;
    double y=2.9;
    x=2+y;
```

Тип правой части – double, а переменной x будет присвоено значение 4.

Кроме автоматического приведения типов в выражении можно использовать операцию явного приведения типа:

```
(<тип>)<выражение>
```

Примечание. В языке Си нет логического типа, а любое значение отличное от нуля считается истинным. Выражение-условие принимает значение 1, если оно истинно и 0, если ложно. Например:

```
...int c,d;
    c=getchar();
    d=c>='0' && c<='9';
```

Если введенный символ – цифра, то переменной d будет присвоено значение 1, в противном случае 0.

Вычисление логического выражения прекращается как только становится известна истинность или ложность результата.

Задачи.

1) Какое значение примет переменная z?
   int x=1, y=2, z;
   a) z=x & y | x && x==y;
   b) z=x & y | (x && y);
2) Изменятся ли значения переменных x,y,z и какое значение будет присвоено переменной k?
   int x, y, z, k;
   x=y=z=1;
   k=++x && ++y || ++z;
3) Как будет вычисляться выражение z=x+++y ?
4) Какое значение примет переменная k?
   int k;
   k=~(~0<<2);
5) Определить значение выражения и тип результата.
   а) -3*4L-6/5;
   б) (7+6)%5/2;
6) Какие значения примут целочисленные переменные x,y,z после выполнения следующих операторов:
   x=3; y=3; z=1;
   z+=x<y?x++;y++;
7) Реализовать эффективное умножение и деление переменной x на 16, используя операции поразрядного сдвига (указать возможный тип переменной x и диапазон ее начальных значений для корректного выполнения операции).
8) Используя тернарную операцию, присвоить z наименьшее из значений a, b и c.
9) int x;
   Инвертировать 3 бита переменной x, начиная 5-го, а остальные биты оставить без изменения.
10) int z;
    Обнулить в z все разряды, кроме 5 младших, а в 3 младших разрядах установить 1.

ТЕМА 2. Операторы управления: if, switch, while, do while, for, break, continue. Простейшие функции ввода/вывода: getchar, putchar, printf.

Программа на языке Си состоит из набора функций, одна из которых является главной, называется **main** и запускается системой первой. Функции состоят из операторов. Выражение становится оператором, если за ним идет точка с запятой. Точка с запятой заканчивает любой оператор, кроме составного оператора **{ }**[1] Например: x=0 операция присваивания, а x=0; оператор присваивания. Кроме операторов присваивания в языке Си присутствуют операторы управления.

Условный оператор if-else.

Условный оператор применяется для принятия решения в некоторый момент выполнения программы.

```
if(<выражение>)<оператор1>[else<оператор2>]
```

Если выражение истинно, то выполняется оператор1, в противном случае оператор2.Часть else может отсутствовать.

```
if(a>b) z=a; else z=b;
```

Если в одной из веток необходимо выполнить несколько операторов, используется составной оператор **{ }**.

Примечание. После правой фигурной скобки точка с запятой не ставится[2].

Во вложенных операторах if действует правило: else всегда относится к ближайшему незакрытому оператору if. Для изменения этого порядка выполнения операторов обычно используют составной оператор.

Задача 1. Ввести из стандартного входного потока один символ и, если это цифра, то присвоить целой переменной k нулевое значение, если введенный символ окажется буквой a, то k присвоить значение один, во всех остальных случаях переменной k присвоить значение два.

```
...  int c,k;
     c=getchar();
     if(c>='0' && c<='9') k=0;
     else if(c=='a') k=1;
          else k=2;
```

[1] В Си ';' не разделяет операторы, как в других языках программирования, например, в Паскале, а является неотъемлемой частью оператора.

[2] Запись if(x==y){…};else{…} приведет к ошибке компиляции (здесь ; рассматривается компилятором как пустой оператор после "сокращенного" условного, а следовательно, для else нет соответствующего if)

Переключатель switch.

При множественном ветвлении программы иногда вместо вложенных операторов if удобнее использовать переключатель.

```
switch(<выражение>) {
  case <константное выражение 1>: <оп1>...<опN>
                                   [break;]
  .  .  .
  case <константное выражение M>: <оп1>...<опN>
                                   [break;]
  [default: <оп1> . . . <опN>]
}
```

При совпадении значений выражения и константного выражения одной из ветвей, выполняются операторы этой ветви. Оператор **break** вызывает немедленный выход из переключателя. Если оператор break отсутствует в выбранной ветви, то будут выполнены операторы следующей ветви и т.д. до тех пор, пока не встретится оператор break или не закончатся ветви переключателя.

Примечание. Ветви case и default могут быть расположены в любом порядке, при этом ветвь default выполнится, только в том случае если не будет совпадения ни с одним константным выражением.

Далее приводится решение задачи 1 с помощью переключателя.

```
... int c,k;
    switch(c=getchar()) {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9': k=0; break;
        case 'a': k=1; break;
        default: k=2;
    }
```

Операторы цикла while и do-while.

В операторе цикла с предусловием:

```
    while (<выражение>) <оператор>
```

и операторе цикла с постусловием:

```
    do <оператор> while(<выражение>);
```
тело цикла выполняется до тех пор, пока выражение истинно. Если в цикле необходимо выполнить несколько операторов, используется составной оператор {}.

Задача 2. Из стандартного входного потока ввести строку и записать ее в массив str. Длина строки не превышает 80 символов.

```
... int c,i;
    char str[80];
    i=0;
    while((c=getchar())!='\n') str[i++]=c;
    str[i]='\0';
```

Оператор цикла for.

Оператор цикла for состоит из заголовка и тела цикла.

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

Оператор тела цикла выполняется до тех пор, пока выражение2 истинно. При отсутствии выражение2 считается истинным. Выражение1 обычно содержит начальные присваивания, а выражение3 - изменение значения параметра цикла на каждой итерации. В этих выражениях часто используют операцию "," (запятая). Алгоритм выполнения цикла for выглядит следующим образом:

```
< выражение1>;
while(<выражение2>) { <оператор> <выражение3>; }
```

Например, чтобы обнулить элементы целочисленного массива, можно использовать цикл for.

```
 ...int m[10], i;
    for(i=0;i<10;i++) m[i]=0;
```

В вырожденном случае все выражения и оператор тела цикла могут отсутствовать. Например: **for(;;);** - это бесконечный цикл.

Включение в заголовок цикла посторонних вычислений допустимо, но считается плохим стилем программирования.

```
 ...int m[10], i;
    for(i=0;i<10;m[i++]=0);
```

Примечание. Параметр цикла и составляющие условия его окончания могут изменяться в теле цикла. Выходное значение параметра цикла всегда определено.

Задача 3. Определить длину строки. Нулевой байт не считать.

```
 ...char s[100];
    int len, i;
    for(i=0;s[i];i++);
    len=i;
```

Задача 4. Удалить из строки все пробелы.

```
 ...char s[100];
```

```
    int i, j;
    for(i=0,j=0;s[i];i++)
      if(s[i]!=' ') s[j++]=s[i];
    s[j]='\0';
```

Задача 5. Нарисовать горизонтальную гистограмму частоты вхождения введенных символов. Ввести масштабирование по горизонтальной оси так, чтобы гистограмма была развернута на все окно.

```
#include <stdio.h>
main()
{  int c, m[256]={0}, i, j, max;
   while((c=getchar())!='\n') m[c]++;
   for(i=1,max=m[0];i<256;i++)
     if(max<m[i]) max=m[i];
   for(i=0;i<256;i++)
     if(m[i]){
       printf("%c",i);
       for(j=0;j<80*m[i]/max;j++) putchar('*');
       putchar('\n');
     }
}
```

Операторы break и continue.

Оператор **break;** вызывает немедленный выход из тела цикла или из переключателя switch.

Оператор **continue;** вызывает немедленный переход к следующей итерации цикла (while, do-while или for).

Задачи.

1. #define MAXLEN 256
   char str[MAXLEN];
   int i;

Пусть в символьный массив str записана строка, длина которой меньше MAXLEN.

   a) Реверсировать строку, не используя дополнительных массивов.
   b) char str1[MAXLEN];
      Копировать строку str в str1.
   c) Преобразовать строку в целое.
   d) Целое число, записанное в переменной i, преобразовать в строку.

2. Входной поток состоит из слов. Длина слова не превышает 80 символов. Разделители между словами: пробел, запятая, конец строки.
   a) Распечатать входной поток по одному слову в строке.

b) Распечатать размер самой длинной строки.
c) Распечатать самую длинную строку.

3. Построить вертикальную гистограмму частоты вхождения произвольных символов во вводимую из стандартного входного потока строку. Гистограмма должна занимать все окно, т.е. масштабирование должно быть произведено как по вертикали, так и по горизонтали.
4. Ввести строку из стандартного входного потока в массив типа char, удалить из строки комментарии. (/* . . . */) Дополнительные массивы не использовать.
5. Написать машинно-независимую программу, обнуляющую каждую четную двоичную единицу в числе типа int.
6. Написать машинно-независимую программу, транспонирующую двоичный код целого.

ТЕМА 3. Понятие указателя. Работа с адресами. Указатели и массивы.

Одной из отличительных особенностей языка Си является возможность использования в программах указателей. Указателем является переменная, значение которой есть адрес[3] некоторой области памяти Си - программы. Указатель может ссылаться на переменные определенного типа и функции.

Например:

```
char c, *p;
```

где c – это переменная символьного типа, а p – это указатель на объекты символьного типа. Указатель p может содержать адрес любого объекта символьного типа.

Унарная операция **&** - это операция взятия адреса объекта, например: p=&c; а унарная операция * - это операция косвенной адресации (или раскрытия ссылки). Например, оператор *p='a'; интерпретируется следующим образом: объекту, на который ссылается указатель p, присвоить значение, равное коду символа 'a'.

При увеличении значения указателя на единицу на самом деле адрес увеличивается на размер объекта. Например:

```
double x=1.5,y=7.1,*dp;
dp=&x;
dp++;
y=*dp-3;
```

[3] Речь идет об условной адресации в Си-машине. Эта адресация может не совпадать с реальной адресацией ОЗУ ЭВМ.

Для примера будем считать, что переменные x и y размещены в памяти рядом (подавляющее большинство компиляторов разместит их рядом, но в стандарте языка Си это не зафиксировано), тогда после операции dp++, значение указателя dp увеличится на sizeof(double) и он будет указывать уже на переменную y. В следующем операторе присваивания значение переменной y уменьшится на три.

Примечание. Унарные операторы * и & имеют более высокий приоритет, чем арифметические операторы, и одинаковый приоритет с унарными операторами ++ и -- . Выражение *p++ интерпретируется следующим образом: взять значение по адресу, хранящемуся в указателе p, а затем увеличить адрес в p на размер объекта, на который ссылается указатель. Операции * и ++ имеют одинаковый приоритет, но выполняются справа налево. Поэтому сначала будет выполнена постфиксная операция p++, но при этом увеличение значения адреса будет отложено до окончания вычисления всего выражения. Аналогично, в случае *--p адрес в p уменьшится прежде, чем по этому указателю будет получено число, т.к. здесь -- - префиксная операция.

В языке Си можно также определить указатель на неопределенный тип:

```
void  *vp;
```

Такому указателю может быть присвоен адрес объекта любого типа или значение любого указателя. Обратное неверно.

```
int *ip, k;
void vp;
ip=&k;
vp=ip;        // Правильно
ip=vp;        // Неправильно
ip=(int*)vp; // Правильно
```

Над указателем типа void* нельзя выполнять операцию косвенной адресации без явного приведения типа.

```
k+=*(int*)vp;     // Правильно
(int*)vp++;       // Неправильно
((int*)vp)++;     // Правильно
```

Указатели используются чаще всего при работе с массивами. Имя массива есть константный указатель, содержащий адрес нулевого элемента.

```
double  m[10], *dp;
```

Чтобы присвоить указателю dp адрес начала массива m, можно написать dp=m; или dp=&m[0]; что эквивалентно. Увеличение значения указателя на единицу приводит к следующему элементу массива.

```
dp=m;    - адрес m[0]
dp+1     - адрес m[1]
dp+i     - адрес m[i]
*(dp+i)  - m[i]
m++;     /* ошибка, константный указатель
         изменять нельзя */
```

Задача1. В вещественном массиве найти максимальное значение.

```
double m[100], *p, max;
.  .  .
for( p=m+1,max=*m;p<m+100;p++)
  if(max<*p) max=*p;
```

Указатели можно присваивать, сравнивать, вычитать и складывать с целыми числами.

Задача 2. Определить длину строки.

```
char str[100],*p, len;
.  .  .
p=str;
while(*p) p++;
len=p-str;
```

Задача 3. Скопировать строку s в t.

```
char s[100], t[100], *p1, *p2;
.  .  .
p1=s; p2=t;
while( *p2++=*p1++);
```

Задачи.

1. Конкатенировать строки (аналог функции strcat).
2. Сравнить строки (аналог функции strcmp).
3. Присвоить переменной единицу, если одна строка содержится в конце другой, и ноль в противном случае.
4. Поменять местами первый отрицательный элемент вещественного массива m1 с последним положительным элементом вещественного массива m2.
5. Изменить знак у всех отрицательных элементов вещественного массива X.
6. Определить, у скольких элементов целочисленного массива X равные соседи(предыдущий и последующий элементы). Записать ответ в переменную n.
7. Проверить на равенство строки s1 и s2 при условии, что пробелы не учитываются.
8. Описаны 2 строки:

char s[255], t[10];
Подсчитать количество вхождений строки t в строку s, ответ записать в переменную count, а переменной last присвоить индекс начала последнего вхождения.

9. Описаны 2 строки:
   char s[200], t[200];
   Все цифры строки s записать в начало строки t, а остальные символы – в конец (в любом порядке).
10. Упорядочить вещественный массив x по неубыванию, используя метод сортировки выбором (найти максимальный элемент массива и переставить его с последним элементом; затем применить этот же метод ко всем элементам, кроме последнего).
11. Упорядочить вещественный массив x по неубыванию, используя метод сортировки обменом ("метод пузырька") (последовательно сравнивать пары соседних элементов: x1 с x2, x2 с x3 и т.д., и если первый элемент пары больше второго, то переставлять их – наибольший элемент окажется в конце массива; затем применить этот же метод ко всем элементам, кроме последнего).
12. Упорядочить вещественный массив x по неубыванию, используя метод сортировки вставками (пусть первые k элементов уже упорядочены по неубыванию; взять (k+1)-ый элемент и разместить его между первыми k, не нарушая порядок).

ТЕМА 4. Понятие функции. Оператор return. Понятие прототипа функции. Механизм передачи параметров.

В стандарте ANSI C представлено два способа описания и определения функций, так называемые «новая» (введенная стандартом) и «старая» (используемая до принятия стандарта) нотации.

Согласно новой нотации определение любой функции имеет следующий вид:

```
[<тип результата>]<имя функции>([список параметров])
{ [декларации]
  [операторы]
}
```

Согласно старой нотации функция определяется следующим образом:

```
[<тип результата>]<имя функции>([список имен парам.])
[описание параметров]
{ [декларации]
  [операторы]
}
```

В этом случае в круглых скобках после имени функции перечисляются имена параметров, а их типы задаются перед первой от-

крывающей фигурной скобкой. Функция может возвращать одно значение, тип которого указывается в заголовке, а само значение задается выражением в операторе **return**:

```
return <выражение>;
```

Оператор return возвращает управление вызвавшей функции.
Если возвращаемое значение имеет тип int, то тип результата можно не указывать. Функция может и не возвращать значение, тогда вместо типа результата следует писать void.

В стандарт Си было добавлено средство контроля за соответствием фактических параметров в вызове функции ее формальным параметрам. В начале программного файла, как правило, для всех функций, кроме main, задаются **прототипы**. Синтаксически прототип повторяет заголовок функции. При отсутствии прототипа функции, если в выражении встретилось имя, нигде ранее не описанное, за которым следует открывающая скобка, такое имя считается именем функции, возвращающей значение типа int. При несоответствии типов формальных и фактических параметров автоматическое приведение типов не осуществляется, что приводит к ошибочному результату. А при наличии прототипа функции все приведения типов осуществляются автоматически. Согласованность имен параметров в прототипе и в определении функции не требуется, могут быть указаны только типы параметров без имен.
В случае старой нотации в прототипе функции параметры не указываются, и компилятор не может обнаружить ошибок при несоответствии количества формальных и фактических параметров или их типов. Пример описания и определения функции в новой и в старой нотациях:

```
/*прототип */
int f(char);
     ...
/* определение */
int f(char c)
{
 /*…тело функции…*/
}
```

новая нотация

```
/*прототип */
int f();
     ...
/* определение */
int f(c)
char c;
{
 /*…тело функции…*/
}
```

старая нотация

Аргументы функции передаются по “значению”. Если аргументом является имя массива, то функции передается копия адреса начала этого массива, а сам массив не копируется. Нельзя передать целиком сам массив функции и таким образом:

```
int arr[10];
```

```
    ...
f(*arr);
```

В этом случае функция f получит в качестве фактического значения аргумента значение нулевого элемента массива.

Задача1. Написать функцию, суммирующую два вещественных значения в двух вариантах: сумма – возвращаемое значение функции и сумма – параметр функции.

```
  . . .
double sum1(double,double);/* прототип
                                функции sum1*/
void sum2(double,double,double*); /* прототип
                                функции sum2*/
main()
{ double a1=2.5, u;
  int a2=3;
  u=sum1(a1,a2);
  printf("u=%f\n",u); /* u=5.5 */
  sum2(a1,a2,&u);
  printf("u=%f\n",u); /* u=5.5 */
}
double sum1(double x,double y)
{ return x+y; }
void sum2(double x,double y,double *z)
{ *z=x+y; }
```

Задача 2. Написать функцию суммирования двух целочисленных векторов.

```
  . . .
void sum_vec(int*,int*,int*,int);
main()
{ int m1[20],m2[20],m3[20],i;
  . . .
  sum_vec(m1,m2,m3,20);
  for(i=0;i<20;i++)
    printf("m3[%d]=%d\n",i,m3[i]);
}
void sum_vec(int x[],int *y,int z[],int k)
{ int i;
  for(i=0;i<k;i++)
    *z++=x[i]+y[i]; /*z[i]=x[i]+y[i];*/
}
```

Список формальных параметров функции может быть переменной длины – в этом случае он заканчивается “, …” и должен содержать хотя бы один именованный параметр. Примером такой функции является функция печати printf, описанная в stdio.h:

```
int printf(const char *format, …);
```

Для работы с параметрами, количество и тип которых заранее не известны, используется механизм макросов из файла стандартной библиотеки <stdarg.h>. В теле функции заводится переменная типа va_list – указатель на очередной неименованный аргумент. Доступ к неименованным параметрам через переменную типа va_list возможен только после обращения к макросу va_start. Макрос va_start получает два параметра - переменную типа va_list и последний именованный параметр функции и инициализирует переменную типа va_list так, чтобы она указывала на первый неименованный параметр:

```
int f(p1, p2, ...)
{
  va_list parg;
  va_start(parg, p2);/* p2 – последний
        именованный параметр функции f */
  int i = va_arg(parg, int);
     ...
  va_end(parg);
}
```

Далее, каждое обращение к макросу va_arg выдает очередной параметр и передвигает указатель (переменную типа va_list) на следующий параметр. К макросу va_arg обращаются с двумя параметрами – переменной типа va_list, проинициализированной с помощью va_start, и именем типа возвращаемого параметра. Макрос va_end нужен для корректного завершения работы с переменной типа va_list.

Поскольку функции не известны ни типы передаваемых неименованных параметров, ни их количество, то для корректной работы с такими параметрами может использоваться различная дополнительная информация. Например, как в случае функции printf, параметр format полностью определяет типы и число последующих параметров. Другой способ – передавать через фиксированный параметр количество неименованных параметров.

Пример: Написать функцию, возвращающую максимальный из полученных ею фактических параметров (типа unsigned int).

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

```
unsigned int unsgn_max(unsigned int count, ...);
     /* Число неименованных параметров передается
     через параметр count и не учитывается при по-
     иске максимума. */
main()
{
  printf("max = %d\n",
          unsgn_max (3, 10, 20, 30));
}
unsigned int unsgn_max(unsigned int count, ...)
{
  va_list ap;
  unsigned int res = 0, cur;
  int i;
  va_start(ap, count);
  for(i=1; i<=count; i++)
    if(res < (cur = va_arg(ap,uinsigend int)))
      res = cur;
  va_end(ap);
  return res;
}
```

Задачи.

1. Дан массив, содержащий заданное количество вещественных чисел. Написать функцию, возвращающую максимальное значение этого массива.
2. Даны целочисленные массивы X и Y , содержащие по 20 элементов. Написать функцию, которая возвращает значение u.

$$U = \begin{cases} \sum_{i=1}^{20} x_i^{\,2}, \text{при} \sum_{i=1}^{15} x_i y_i > 0 \\ \sum_{i=10}^{20} y_i^{\,2}, \text{иначе} \end{cases}$$

3. Написать аналог функции strstr , которая возвращает указатель на первое вхождение одной строки в другую и нулевой указатель в противном случае. Используя функцию strstr, найти последнее вхождение строки “end” во введенной строке. Распечатать символы, которые следуют за этим вхождением.
4. Написать функцию, упорядочивающую вещественный массив. Число элементов массива и сам массив передать в качестве параметров.

5. Написать функцию getletter(), читающую одну литеру из стандартного ввода с помощью getchar() и возвращающую введенную литеру, если это большая или малая латинская буква, и –1 в противном случае.
6. Описать функцию, определяющую, сколько элементов вещественного массива X из n элементов равны числу y.
7. Написать функцию, которой передаются 2 100-элементных вещественных массива, определяющую, составлены ли эти 2 массива из одних и тех же чисел без учета порядка их следования, но с учетом повторяющихся чисел(функция возвращает 1 в случае положительного ответа).
8. Написать функцию, которая за 1 просмотр циклически сдвигает полученный вещественный массив на 20 позиций влево (функция для работы должна использовать вспомогательный массив).
9. Написать функцию, вычисляющую n-ое число Фибоначчи (n>=0) по правилу:

$$F(n) = \begin{cases} 1, \text{ при } n=0 \text{ или } n=1 \\ F(n-1) + F(n-2), n>=2 \end{cases}$$

ТЕМА 5. Общая структура программного файла. Области видимости и существования переменных. Препроцессор.

Программа может размещаться как в одном, так и в нескольких файлах, содержать одну или несколько функций, одна из которых считается головной (**main**), с нее начинается выполнение программы. Определение каждой функции размещается полностью в одном файле, но файл может содержать несколько определений различных функций.

В соответствии со структурой программы переменные могут обладать следующими качествами:

- **видимость переменной (область видимости);**
- **существование переменной (область существования).**

Область видимости переменной определяет текстовое пространство программы (оно не обязательно непрерывно), из любой точки которого доступна данная переменная с данным именем. С точки зрения видимости можно выделить следующие группы переменных:

- видимые в пределах блоков,
- видимые в пределах файла,
- видимые в пределах программы.

Для переменных, определенных внутри функции в начале любого блока, областью видимости является весь этот блок. В случае вложенных блоков переменные, определенные внутри вложенного блока, “перекрывают” переменные с такими же именами, определенные в объемлющем блоке (и так для любой степени вложенности). Например:

```
main()
{ int x = 1;
  if(x>0)
  {  int x = 2;
     printf("x = %d\n", ++x); /*выводит: x = 3 */
  }
  printf("x = %d\n", x); /*выводит: x = 1 */
}
```

Переменные, определенные внутри функции, “перекрывают” формальные параметры с теми же именами[4]:

```
int f(int f)
{
  int f = 1;
     …
}
```

Переменные, определенные вне блоков, доступны с точки определения до конца файла. Если на такую переменную нужно сослаться до того, как она определена, должно быть ее описание со спецификатором **extern**, например:

```
int x;
main()
{
  extern int y;
  x = y = 10;
  printf("x=%d, y=%d\n",x,y); /* x=10, y=10 */
}
 …
int y;
```

В файле вне функций не может встречаться несколько определений переменных (возможно, разных типов) с одним и тем же именем:

```
int x;
```

[4] Следует заметить, что в приведенном ниже примере коллизии имени функции f с именем формального параметра или с переменной f, определенной внутри функции, не будет из-за различия областей видимости.

```
main()
{
  ...
}
float x;/* ошибка: повторное определение x */
```

Указание **static**, примененное к переменной или функции, ограничивает область их видимости концом файла[5].

Если используемая переменная определена в другом программном файле, она также должна быть описана со спецификатором **extern**. При таком описании переменных память не отводится, а только декларируется тип переменной.

Примечание. Переменные, определенные внутри блока, как и формальные параметры функций "перекрывают" переменные с теми же именами, видимые в пределах файла и в пределах программы.

Область существования переменной – это множество всех точек программы, при приходе управления на которые переменная существует, т.е. для нее выделена память. С этой точки зрения можно выделить две группы переменных:

1. Статические переменные

Переменные, являющиеся статическими, существуют на всем протяжении работы программы. Память под эти переменные выделяется на этапе редактирования внешних связей и загрузки программы, тогда же происходит и инициализация статических переменных (следует отметить, что статические переменные по умолчанию инициализируются нулем). Правила определения статических переменных различаются в зависимости от конкретного места программы, в котором это определение встретилось. Для определения статической переменной, локализованной в блоке, используется ключевое слово static, например:

```
int max; /*статическая переменная вне блока*/
int f(int param)
{
  static int min; /* статическая переменная,
                     определенная внутри блока */
    ...
}
```

Основным свойством статических переменных, определенных внутри блока, является сохранение их значений при выходе из блока. Например:

[5] Здесь возникает некая терминологическая путаница, поскольку, как будет рассказано ниже, ключевое слово static используется для определения статических переменных, а в данном контексте с помощью static задаются границы области видимости.

```
#include <stdio.h>
void print_a();
main()
{ int i;
  for(i=0; i<5; i++)
    print_a();
}

void print_a()
{
  static int a = 1;
  printf("a = %d\n", a++);

}
```

в результате напечатается:

```
a = 1
a = 2
a = 3
a = 4
a = 5
```

Все переменные, определенные вне функций, являются статическими.

2. Автоматические переменные

Автоматическими переменными являются все переменные определенные внутри блока (функции) и не являющиеся статическими. Автоматические переменные существуют на протяжении работы блока, в котором они определены, включая блоки, вложенные в данный. Выделение памяти под автоматические переменные и их инициализация осуществляется каждый раз при входе в блок (начальное значение по умолчанию для автоматических переменных не определено).

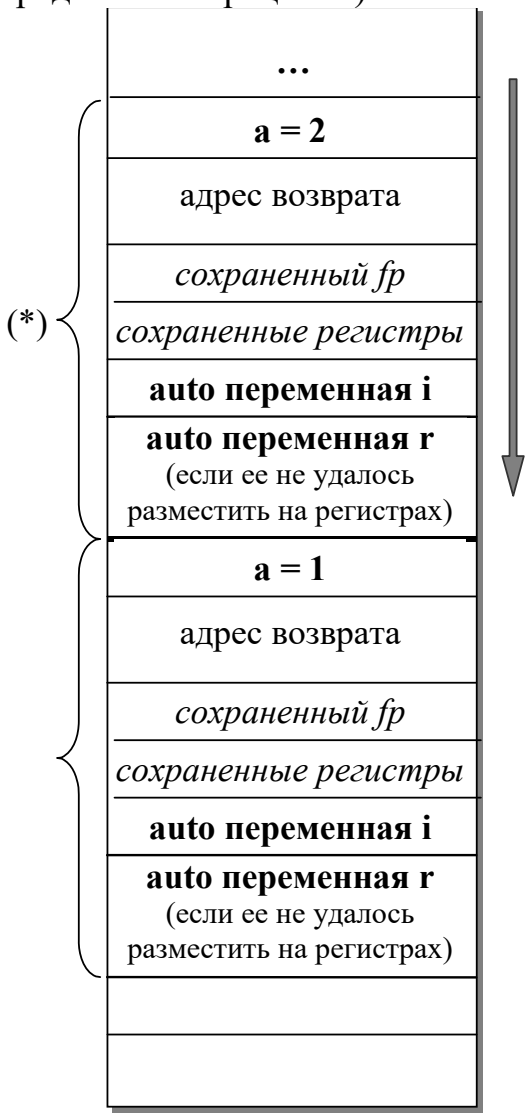
```
#include <stdio.h>
void print_a();
main()
{ int i;
  for(i=0; i<5; i++)
    print_a();
}

void print_a()
{
  int a = 1;
  printf("a = %d\n", a++);

}
```

в результате напечатается:

```
a = 1
a = 1
a = 1
a = 1
a = 1
```

При выходе из блока память, выделенная под автоматические переменные, освобождается. Таким образом, автоматические пере-

менные являются более эффективным средством при работе с памятью, чем статические переменные. Однако, надо учитывать тот факт, что инициализация автоматических переменных, фактически, эквивалентна присваиванию значений и требует определенных временных затрат на исполнение этого кода. В качестве средств оптимизации Си программ предлагается возможность использования так называемых регистровых переменных. Это достигается за счет использования квалификатора register в определении переменных, что указывает компилятору, что данную переменную в целях ускорения программы имеет смысл разместить на регистрах, однако компилятор может проигнорировать это указание. Квалификатор register может применяться только к автоматическим переменным и формальным параметрам функций. Независимо от того, была ли переменная, описанная с квалификатором register, действительно размещена на регистрах или нет, для нее не определено понятие адреса (т.е. не определена операция &).




```
int f(int a){
  int i;
  register int r;
     ...
  r = a--;
  if(r){
     return f(r);
  }
  return r;
}

main(){
  ...
  int b = f(2);
  ...
}
```

(*) – стековый кадр – непрерывная область памяти стека, используемая функцией для своей работы

fp (*frame pointer*) – указатель стекового кадра

Рис.1

Обычно автоматические переменные, как и формальные параметры функций, реализуются с использованием стека. Параметрам функции отводится память в стеке, в которую копируются фактические значения. По окончании работы функции память освобождается. Рассмотрим состояние стека на примере рекурсивной функции (см. Рис.1). В примере каждый раз при вызове функции f в стеке отводится место для параметра a и автоматической переменной i, а также для автоматической переменной r в случае, если ее не удалось разместить на регистрах. Заметим, что при рекурсивном вызове функции f еще раз выделяется память в стеке под автоматические переменные и параметры нового вызова. При выходе из функции выделенная в стеке память освобождается.

Препроцессор[6]. Стандартная схема трансляции Си-программы состоит из двух этапов: препроцессирование и собственно компиляция. Препроцессор выполняет *макроподстановку*, *условную компиляцию*, *подключение файлов* и тем самым формирует текст программы, поступающий на вход компилятору.

Программный файл может содержать директивы препроцессору. Директивам препроцессору предшествует знак #. Например:

```
#include <stdio.h>    /* подключение файла */
#define  a "max=%d\n" /* макроподстановка */
int x=15;
max(int);
main()
{ int y, u;
  scanf("%d",&y);
  u=max(y);
  printf(a,u);
}
max(int f)
{ int k;
  k=(x>f)?x:f;
  return k;
}
```

Включение файлов. На место директивы #include препроцессор подставляет содержимое указанного в ней файла. Порядок поиска подключаемого файла зависит от реализации Си. Если имя файла заключено в < >, то в UNIX–системах он ищется в стандартном каталоге подключаемых файлов /usr/include, как например в случае:

[6] Мы не рассматриваем особенности функционирования конкретных трансляторов, а описываем наиболее часто используемые возможности препроцессора.

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

Если же имя указано в “ ”, то препроцессор рассматривает его как относительный путь от местонахождения того файла, в котором встретился #include или, если имя начинается с “/” – как абсолютный путь от корня. Например:

```
#include "myfile.h"
#include "/usr/stuff/myfile.h"
```

Включаемые файлы, в свою очередь, могут содержать #include-директивы.

Макроподстановка.

```
#define имя подставляемый_текст
```

Начиная от места появления #define и до конца файла, везде, где встречается имя, указанное в #define, вместо него препроцессор подставляет заданный текст (кроме случаев, когда имя встречается внутри текста в кавычках).

Примеры:

```
#define NUMBER 10
#define DOUBLED_NUMBER NUMBER*2 /*в #define-
определении можно использовать более ранние
определения[7] */
#define PrintHello printf("Hello,\
world"); /* "\" используется для продолжения
определения на следующей строке */
```

Можно определить макроподстановку с параметрами, что позволяет изменять подставляемый текст в зависимости от фактических параметров. Например:

```
#define SQR(x) ((x)*(x))
#define Print(x) printf(#x " = %d\n", x)
/* имя формального параметра, встретившееся в "",
не заменяется на фактический параметр, но если
перед именем формального параметра стоит #, то в
макрорасширении #имя_формального_параметра будет
заменено на строку "фактический_параметр", после
чего в нашем примере строки конкатенируются */
main()
{
  int n = 4;
  Print(n);       /* выводит на печать: n = 4 */
  Print(SQR(n)); /* SQR(n) = 16 */
```

[7] при этом в текст программы вместо DOUBLED_NUMBER подставится 10*2, а не 20

```
/* следующие два примера демонстрируют необходи-
мость скобок в определении SQR(x) для обеспечения
нужного порядка вычислений */
  Print(SQR(n+2));   /* SQR(n+2) = 36 */
  Print(256/SQR(n)); /* 256/SQR(n) = 16 */
}
```

Оператор ## позволяет «склеивать» аргументы в макроподстановке, например: `#define bond(left, right) left##right`

Таким образом для `bond(name, 123)` будет сгенерировано `name123`

Примечание. Повторный #define для того же имени является ошибкой, если подставляемые тексты не совпадают с точностью до разделяющих пробельных символов. Препроцессору можно дать указание «забыть» определенное имя с помощью директивы

```
    #undef имя
```

(если имя не было ранее определено, ошибки не произойдет)

Условная компиляция. Другой пример использования препроцессора – управление выборочным включением того или иного текста в программу в зависимости от вычисляемого на этапе препроцессирования условия.

```
if-директива выражение_или_идентификатор
    ...
[{#elif выражение}
    ...
#else
    ... ]
#endif
```

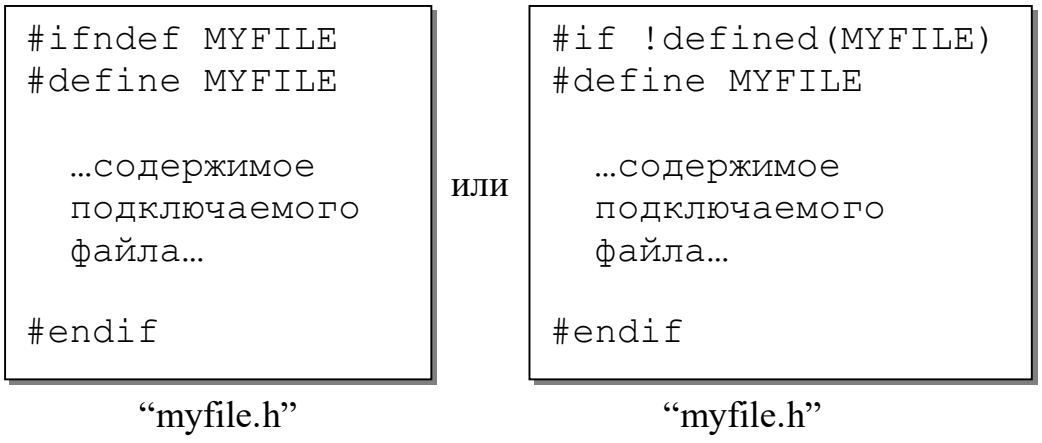
где `if-директива` – это:

```
#if константное_выражение
```

(вычислить выражение и если получилось ненулевое значение, то включить весь текст до строки else-директивы или `#endif`)

```
#ifdef имя /* 1, если имя уже определено */
#ifndef имя /*1, если имя не было определено */
```

Например, во избежание повторного включения файла "myfile.h" его оформляют следующим образом:



ТЕМА 6. Инициализация одномерных массивов. Массивы указателей и их инициализация. Аргументы командной строки. Указатели на функции.

При инициализации массивов инициализирующие значения заключаются в фигурные скобки и, если их не хватает, то оставшимся элементам присваиваются нулевые значения. Например:

```
int m1[5]={0,1,2,3}, m2[10]={0};
```

Последний элемент массива m1 и все элементы массива m2 будут обнулены. Если размер массива не указан, то он определяется по количеству инициализирующих значений.

Массив типа char (строка) может быть проинициализирован двумя способами:

```
char str1[ ]={ 'a','b','c','\0'};
char str2[ ]="abc";
```

Под строки str1 и str2 будет отведено по 4 байта. Строка str2 будет автоматически дополнена нулевым байтом.

При инициализации указателя на строку, в него записывается адрес константной строки, элементы которой изменять нельзя.

```
char *p="abc";
*p='A';       /* Неправильно */
*str2='A';    /* Правильно */
```

Язык Си позволяет инициализировать массив указателей, например:

```
char *mas[]={"for","while","do","return",NULL};
```

Элементами массива mas являются адреса строковых констант.

`mas[i]` – адрес i-ой строки (адрес её нулевого символа).

`mas[i]+j` – адрес j-го символа i-ой строки.

Задача 1. Написать функцию, которой в качестве аргумента передается массив указателей на строки (признак конца - нулевой указатель). Распечатать последний символ каждой строки.

```
void fp(char *s[])/* void fp(char **s)
                        тождественно */
{ int i=0;
  while(s[i]!=NULL) {
    printf("%c\n",
           *(s[i]+ strlen(s[i])-1));
    i++;
  }
}
```

Задача 2. Написать фрагмент программы, размещающий в динамической памяти вводимые из стандартного входного потока вещественные числа. Количество вводимых чисел вводится первым.

```
... int k,i;
    double *p;
    . . .
    scanf("%d",&k);
    p=(double*)malloc(k*sizeof(double));
    for(i=0;i<k;i++) scanf("%lf",p+i);
```

Задача 3. Ввести строку из стандартного входного потока длиной не более ста символов и разместить ее в динамической памяти.

```
... char str[100],*p;
    ...
    if(gets(str)!=NULL) {
        p=(char*)malloc(strlen(str)+1);
        strcpy(p,str);
    }
```

В операционной системе имеется возможность передавать аргументы запускаемой программе при помощи командной строки. Стандарт ANSI C определяет только 2 аргумента функции main():

```
main(int argc, char *argv[])
```

В первом (argc) передается количество аргументов командной строки, во втором (argv) – адрес массива указателей на строки-аргументы. По соглашению argv[0] всегда содержит адрес строки, в которой записано имя вызванной программы. Поэтому если argc равен 1, то в командной строке после имени программы никаких аргументов нет. Кроме того, всегда выполняется соглашение argv[argc]=NULL.

Стандарт POSIX.1 определяет третий аргумент функции main():

```
main(int argc, char *argv[], char *envp[])
```

Массив envp[] содержит указатели на переменные окружения, передаваемые программе. Каждая переменная – строка вида

```
имя_переменной = значение_переменной
```

POSIX.1 также рекомендует доступ к окружению из программы через глобальную переменную `environ`:

```
extern char **environ;
```

В языке Си можно определять указатели на функции, которые ничем не отличаются от обычных указателей. Указатель на функцию можно присваивать, размещать в массиве, передавать в функцию в качестве параметра. Например:

```
double (*fp)(double);
```

fp – это указатель на функцию с одним параметром типа double, возвращающую значение типа double. Теперь указателю fp может быть присвоен адрес любой функции, имеющей такой же прототип, после чего его можно будет использовать наравне с именем функции.

```
double x=1.57,y;
    fp=sin;
    y=fp(x); /* y=sin(x) – эквивалентно */
```

Задача 1. Написать программу вычисляющую y=f(x), где имя функции(sin, cos, exp или sqrt) и ее параметр задаются в качестве аргументов командной строки.

```
#include<stdio.h>
#include<string.h>
#include<math.h>
typedef double (*tf)(double);
main(int argc,char * argv[])
{ char *str[4]={"sin","cos","exp","sqrt"};
  tf m[4]={sin,cos,exp,sqrt};
  double x;
  int i;
  for(i=0;i<4;i++)
    if(!strcmp(argv[1],str[i])) break;
  if(i==4) {
    printf("Имя функции задано неверно\n");
    return(-1);
  }
  x=atof(argv[2]);
  printf("%s(%f)=%f\n",str[i], x, m[i](x));
}
```

При передаче функции в качестве параметра ее имя трактуется как адрес этой функции, поэтому оператор & перед ним не нужен (также как & был не нужен перед именем массива).

Например:

```
void sort(void *vector[],
          int (*comp)(void*, void*));
/*функция сортировки получает функцию
  сравнения в качестве параметра */
int complen(char *, char *);
int lexcomp(char *, char *);
main()
{ int lexorder;
  char * strings[];
   ... /* заполнение strings */
  sort((void **)strings,
(int(*)(void*,void*))(lexorder?lexcomp:complen));
/* явные преобразования нужны для проверки компи-
лятором согласованности типов */
}
```

Задачи.

1. Ввести набор слов. Слов не более 20-ти. Разделители между словами: пробел, запятая, конец строки. Длина слова не превышает 20 символов. Разместить слова в динамической памяти (лишнего места не занимать) и сформировать массив указателей на слова. Использовать в качестве признака конца массива нулевой указатель NULL. Упорядочить слова в неубывающем порядке. Распечатать слова, каждое на отдельной строке.
2. Написать функции, которым в качестве параметра передается массив указателей на строки. Признак конца – нулевой указатель.
   a) Написать функцию, которая распечатывает первые три символа из каждой строки.
   b) Написать функцию, которая печатает самую длинную строку.
   c) Написать функцию, которая печатает строки, включающие в себя строку “begin”.
3. Ввести набор строк. Количество строк не более 15-ти. Длина каждой строки не должна превышать 80 символов. Под каждую введенную строку зарезервировать в динамической памяти место под реальную ее длину, скопировать туда строку и записать адрес в массив указателей . Признак конца в массиве указателей – NULL(нулевой указатель). Написать функцию, которой передается в качестве параметра массив указателей, а выходное значение – адрес самой большой (лексикографически) строки. Используя этот адрес, распечатать самую большую строку.
4. Распечатать все аргументы командной строки (без имени вызываемой программы).

5. Если среди параметров командной строки, есть строки, содержащие “end” более одного раза, то распечатать символы этих строк, находящиеся после второго вхождения “end”.
6. Напечатать таблицу значений функции на отрезке [a,b] с шагом h. Имя функции(sin, cos, tan, log, fabs, sqrt или exp), a,b и h ввести как параметры командной строки. Вызов функций реализовать через массив указателей на функции.
7. Написать функцию вычисления интеграла f(x) на отрезке [a,b] методом прямоугольников. f(x) – любая функция, интегрируемая на заданном отрезке. Функцию f(x) передавать через параметр-указатель на функцию.
8. Получить польскую инверсную запись (ПОЛИЗ) выражения, содержащего цифры 0,1, . . . 9, скобки и арифметические операции: *, /, +, - .
9. Вычислить значение выражения по его ПОЛИЗ’у.
10. Ввести в качестве аргумента командной строки произвольное скобочное выражение, содержащее числа (целые и вещественные), знаки арифметических операций (+, -, *, /) и стандартные функции из математической библиотеки. Выражение может содержать параметр X, тогда и только тогда должен быть предусмотрен ввод конкретного значения. Вычислить это выражение и результат вывести на печать. В качестве усложнения задачи предлагается ввести контроль правильности введенного выражения или не ограничивать число параметров.
11. Написать программу, печатающую имена и значения всех переменных окружения.

ТЕМА 7. Многомерные массивы и их инициализация. Передача многомерных массивов в качестве параметров функций. Указатель на массив. Интерпретация сложных деклараций. Оператор typedef.

В языке Си массив представляет собой совокупность элементов одного и того же типа. При определении массива указывается тип элементов и их количество в квадратных скобках. Массив можно конструировать из объектов арифметического типа, указателей, структур и объединений, а также других массивов(генерируя при этом многомерные массивы). Например, int m[3][4]; - это определение массива из трех элементов, каждый элемент которого в свою очередь является массивом, состоящим из четырех целочисленных объектов.

В общем случае декларация массива имеет вид:

`T D`, где D имеет вид

`D1[<константное выражение целого типа>]`.

Аналогично определяется D1 и т.д.

Доступ к элементам массивов осуществляется через индексирование, например: m[i][j], где i – номер строки, а j – номер столбца. Операция индексирования E1[E2] определена как *(E1+E2), следовательно m[i][j] тождественно *(*(m+i)+j).

Массивы хранятся построчно, т.е. в общем виде быстрее всего изменяется самый правый индекс.

Инициализация многомерных массивов осуществляется аналогично одномерным массивам. Например:

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

Если инициализирующих значений меньше, чем указанная размерность, то соответствующим элементам присваиваются нулевые значения. Например:

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

Элементы: w[1][2], w[2][0], w[2][1], w[2][2] примут нулевые значения.

Для многомерных массивов пустой может быть только первая размерность, реальное значение которой определяется по количеству инициализирующих значений. Например, если в выше приведенном примере опустить число строк, то тогда будет определена матрица 2x3.

На примере двумерного целочисленного массива m рассмотрим понятие указателя на массив.

```
int *p1;
int (*p2)[4];
```

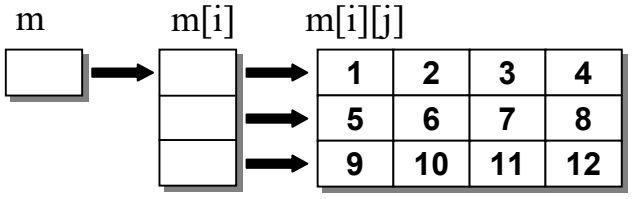
p1 – это указатель на объект целого типа, ему может быть присвоен адрес любого элемента матрицы m. Например, адрес 0-го элемента 1-ой строки можно присвоить тремя эквивалентными способами:

```
p1=&m[1][0];      p1=m[1];      p1=*(m+1);
```

p2 – это указатель на массив из четырех целочисленных элементов и ему может быть присвоен адрес любой строки матрицы, например:

```
p2=m+1;
```

Соответственно, оператор p1++; вызывает переход к следующему элементу 1-ой строки, а оператор p2++; вызывает переход к следующей строке матрицы. Тогда *p1=6, а **p2=9.



При передаче двумерного массива в качестве параметра функции возможно три варианта записи заголовка функции:

```
1) f(int  mas[3][4]);
```

```
2) f(int  mas[][4]);
3) f(int  (*p)[4]);
```

Все три варианта эквивалентны и позволяют пользоваться внутри функции как индексной записью, так и указателями.

Задача 1. Написать функцию, которая вычисляет и распечатывает сумму элементов каждого столбца вещественной матрицы 5x6.

```
void sum(double (*p)[6])
{ int i,j;
  double s,(*pp)[6];
  for(i=0;i<6;i++) {
    pp=p;
    for(j=0,s=0.0; j<5; j++) {
        s+=*(*pp+i);
        pp++;
    }
    printf("%f\n",s);
  }
}
```

Интерпретация сложных деклараций. В декларациях обычно используется имя (идентификатор) и один из модификаторов *, [ ] и ( ), причем разрешается использовать более одного модификатора в одной декларации[8]. Для раскрытия этих деклараций применяются следующие правила:

1. Чем ближе модификатор стоит к идентификатору, тем выше его приоритет.
2. Приоритет ( ) и [ ] выше, чем приоритет *.
3. Приоритет повышается заключением в скобки ().

Примеры:

1) Массивы и указатели

`int matrix[10][10];`	matrix – массив массивов типа int
`char **argv;`	argv – указатель на указатель на char
`int (*ip)[10];`	ip – указатель на массив из 10 элементов типа int
`int *ip[10];`	ip - 10-элементный массив указателей на int
`int *ipp[3][4];`	ipp - 3-элементный массив указателей на 4-элементный массив типа int
`int (*ipp)[3][4];`	ipp – указатель на 3-элементный массив, каждый элемент которого - 4-

[8] В этой теме рассматривается не строгий синтаксис деклараций, а способ разобраться в том, как понимать декларируемое имя.

	элементный массив типа int

2) Функции и указатели

`int *f();`	f – функция, возвращающая указатель на int
`int (*pf)();`	pf – указатель на функцию, возвращающую int
`char (*(*x())[])();`	x – функция, возвращающая указатель на массив указателей на функцию, возвращающую char
`char (*(*x[3])())[5];`	x – массив из 3 указателей на функцию, возвращающую указатель на массив из 5 элементов типа char

Для упрощения прочтения сложных деклараций, а также для именования, типам данных можно задавать новые имена с помощью оператора **typedef**[9]. Например:

```
typedef double (*PFD)();
```

определяет тип PFD как “указатель на функцию, возвращающую double”; typedef не создает новый тип, а декларирует новое имя уже существующего типа.

Задачи[10].

1. Дано множество точек на плоскости, заданных координатами (x,y). Написать функцию, возвращающую минимальное расстояние между двумя точками.
2. Написать функцию обработки вещественной матрицы (50x20). Функция должна находить строку, содержащую наибольшее количество отрицательных элементов и распечатывать все элементы этой строки. Если таких строк несколько, то печатать все такие строки.
3. Написать функцию, осуществляющую умножение двух матриц порядка MxN и NxK.
4. Что будет выведено на печать?
   int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
   int *pa[3]={a[0],a[1],a[2]};

[9] Спецификация языка не очень четко выделяет класс синтаксической конструкции typedef, мы остановимся на термине «оператор»

[10] Задачи №4 и №5 взяты из первого издания книги Б.Керниган, Д.Ритчи «Язык программирования Си»

```
   int *p=a[0];
 main()
 { int i;
   for(i=0;i<3;i++)
   printf("%d %d %d %d %d\n",a[i][2-i],*a[i],*(*(a+i)+i),*pa[i],p[i]);
 }
```

5. Что будет выведено на печать?

```
char *c[]={"ENTER","NEW","POINT","FIRST"};
char **cp[]={c+3,c+2,c+1,c};
char ***cpp=cp;
main()
{  printf("%s",**++cpp);
   printf("%s",*--*++cpp+3);
   printf(" %s",*cpp[-2]+3);
   printf("%s\n",cpp[-1][-1]+1);
}
```

ТЕМА 8. Структуры, объединения, битовые поля. Программирование динамических структур данных.

Структура – это тип данных, позволяющий сгруппировать несколько переменных (возможно различного типа) под одним именем. В общем случае декларация структуры имеет следующий вид:

```
struct[<тег структуры>]{<список деклараций полей>};
```

Например, для задания комплексных переменных удобно использовать структуру:

```
struct point { int x;
               int y;
} a,b;
```

В данном примере переменные a и b определены сразу за декларацией структуры, но т.к. структура имеет тег point, переменные можно определить и иначе:

```
struct point c,d,*p;
```

Например, используя **typedef** при декларации структуры, задается её новое имя:

```
typedef struct point { int x; int y; } sp;
```

Теперь тип sp – это синоним типа struct point, и следующие определения переменной z эквивалентны:

```
struct point z;
sp z;
```

Структуры могут быть проинициализированы списком константных значений:

```
    struct point k={3,5};
```
Доступ к полям структуры осуществляется с помощью операции «точка»: c.x, c.y, если же определен указатель на структуру, то используется операция ->. Например:
```
    p=&c;
    p->x=2;  /* (*p).x  - эквивалентно */
```
Структуры могут быть вложенными:
```
 struct stud {
    char fio[15];/* фамилия студента*/
    struct data { int year;
                  int mon;
                  int day;
    } d;         /* дата рождения */
    int m[3];    /* оценки в сессию */
    };
```
При такой вложенной декларации структур, тип struct stud и struct data имеют одинаковую область действия.

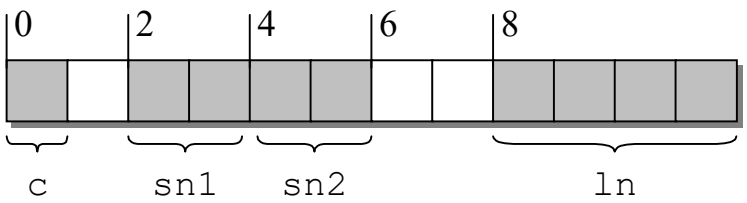
В Си разрешается присваивать и копировать структуры, что позволяет передавать их в функцию в качестве аргумента и передавать из функции в качестве возвращаемого значения, но структуры нельзя сравнивать. Например:
```
  struct stud s={"Ivanov",{1980,6,30},{5,3,4}};
  struct data ss;
  ss=s.d; /*ss.year=1980; ss.mon=6; ss.day=30;*/
  if(ss==s.d) {…} /* ошибка */
```
Задача 1. Написать функцию, параметрами которой являются массив анкет студентов (struct stud) и их количество. Функция печатает фамилии отличников и даты рождения.
```
void f(struct stud g[],int n)
{ int i;
  for(i=0;i<n;i++) {
  if(g[i].m[0]==5 && g[i].m[1]==5 && g[i].m[2]==5)
    printf("%s %d.%d.%d\n",
         g[i].fio,g[i].d.day,g[i].d.mon,g[i].d.year)
  }
}
```
Структуры могут содержать поля - указатели на такие же структуры. Это свойство структур используется при программировании динамических структур данных (стек, очередь, список, дерево), которые будут рассмотрены ниже. Структуры также могут содержать поля – указатели на еще не определенные структуры, что позволяет описывать структуры, ссылающиеся друг на друга, например:
```
    struct s1 {
```

```
        ...
        struct s2 *ps2;
    };
    struct s2 {
        ...
        struct s1 *ps1;
    };
```

Приведенные выше определения корректны, т.к. размер области памяти, резервируемой под любой указатель, всегда известен для конкретной машинной реализации Си.

Рассмотрим подробнее размещение структуры в памяти генератором кода. Здесь возникают две проблемы – собственно размещение объекта типа «структура» в памяти и размещение элементов структуры в области памяти, выделенной под этот объект. Адрес, с которого размещается сама структура, называется *базовым адресом*, а адрес элемента структуры относительно ее базового адреса называется *смещением.* Заметим, что каждый тип данных транслируется в определенный машинный тип (например, “байт”, “слово”, “двойное слово”), которому соответствует некоторое правило выравнивания. Рассмотрим архитектуру с байтовой адресацией и правилами размещения, согласно которым слово располагается, начиная с байта с адресом, кратным 2, а двойное слово – начиная с байта с адресом, кратным 4. В различных реализациях языка применяются различные подходы к определению базового адреса структуры. Это может быть выравнивание по типу первого элемента структуры или по максимально длинному машинному типу и т.п. Смещения элементов структуры зависят от их типов, представления их типов на конкретной архитектуре и правил выравнивания, поэтому при трансляции структур могут оставаться «пропуски».

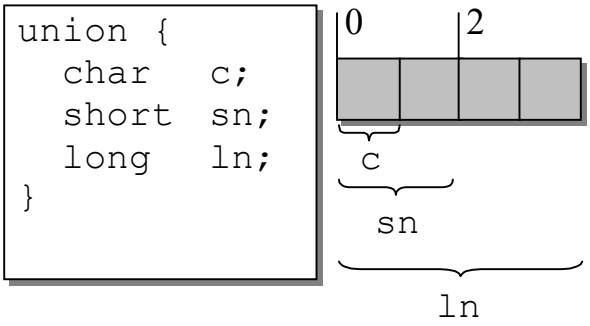
```
struct {
  char   c;
  short  sn1;
  short  sn2;
  long   ln;
}
```



Считаем, что под переменную типа char выделяется 1 байт, под переменную типа short – слово (2 байта), а под переменную типа long – двойное слово(4 байта).

Объединение (**union**) – это тип данных, позволяющий хранить разнородные данные (поля) в одной и той же области памяти. Синтаксис объединений аналогичен синтаксису структур. Фактически, объ-

единение – это структура, все поля которой имеют нулевое смещение относительно ее базового адреса, а размер выделенной области памяти позволяет разместиться в ней самому большому полю. При выравнивании учитывается требование для максимально длинного машинного типа, используемого для представления полей объединения, т.е. если одно поле можно разместить, начиная с адреса двойного слова, а второе – с адреса байта, то компилятор следует первому требованию.



В Си есть средство прямого доступа к группам подряд идущих битов. Внутри некоторой зависящей от реализации единицы памяти («слове») определяются именованные поля, к которым можно осуществлять прямой доступ по имени. Поля определяются при помощи определения структуры, в котором указывается имя поля и его размер, например:

```
struct {
    unsigned int flag1: 1 /*1–ширина поля*/
    unsigned int flag2: 1
    unsigned int code1: 8
    unsigned int code1: 8
} flags;
```

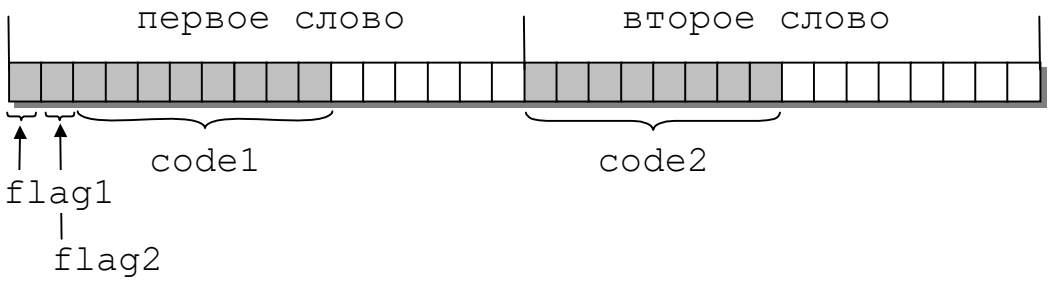
Поле может иметь тип `int`, `unsigned int` или `signed int` и интерпретируется как объект целочисленного типа с заданным числом битов. От реализации языка зависит, рассматривается ли поле типа int как знаковое.

Для присвоения полям значений используется тот же синтаксис, что и для элементов структур:

```
flags.flag1 = 1;
flags.code1 = 5;
```

Под описанную таким образом переменную отводится область памяти размером в слово. Если суммарный размер всех полей превысит размер слова, то отводится 2 соседних слова памяти, причем поле не может перекрывать границу между ними, и компилятор автоматически сдвигает поле, чтобы оно было выровнено по границе слова (в этом случае в первом слове остается безымянное пустое ме-

сто). В приведенном примере при условии, что машинное слово занимает 16 битов, память распределится следующим образом:



Поле длины 0 без имени указывает на то, что оставшиеся в слове разряды нужно пропустить и начать размещение следующих полей с начала следующего слова.

Как уже было замечено ранее, при программировании динамических структур данных (стек, очередь, список, дерево) используются структуры, содержащие поля - указатели на такие же структуры.

Задача 2. Ввести строку символов из стандартного входного потока и распечатать ее в обратном порядке, построив при этом в динамической памяти стек.

```
... int q;
    struct st { char c;
                struct st *s;
    } *p,*n;

    p=n=NULL;
    while((q=getchar())!='\n') {
        /* построение стека */
        n=(struct st*)malloc(sizeof(struct st));
        n->c=q; n->s=p;
        p=n;
    }
    while(n!=NULL) { /* печать строки */
        printf("%c",n->c);
        n=n->s;
    }
```

Задачи.

1. Дана строка символов. Написать функцию создания бинарного дерева (рекурсивный и нерекурсивный варианты). Узлы дерева имеют следующую структуру:

```
struct tree { char c;
              int n;
              struct tree* left;
```

```
        struct tree* right;
    };
```

где с символ входной строки, а n число вхождений данного символа во входную строку. При построении дерева в качестве ключа использовать код символа.

2. Написать рекурсивную функцию подсчета количества узлов дерева.
3. Написать рекурсивную функцию определения высоты дерева.
4. Написать рекурсивную функцию печати узлов дерева в убывающем порядке.
5. Написать функцию печати узлов дерева по слоям (рекурсивный и нерекурсивный варианты).
6. Ввести набор слов. Разделители между словами: пробел, запятая, точка с запятой, конец строки. Признак конца текста - точка. Длина каждого слова не должна превышать 20 символов. Сформировать двоичное дерево, каждый узел которого содержал бы указатель на слово и число вхождений этого слова во входной поток. Ключом при построении дерева должно являться само слово. По окончании формирования дерева распечатать слова в убывающем порядке, в возрастающем порядке и по уровням.

ТЕМА 9. Файлы. Библиотечные функции для работы с файлами.

Язык Си предоставляет программисту широкие возможности по работе с произвольными файлами. Любой файл предварительно должен быть открыт. Для этого используется функция **fopen**:

```
FILE* fopen(char* name, char* mode);
```

Функция получает в качестве аргументов внешнее имя файла (name) и режим доступа (mode), а возвращает файловый указатель, используемый в дальнейшем для работы с файлом. Функция возвращает нулевой указатель, если файл не может быть открыт по каким либо причинам.

При запуске Си-программы операционная система всегда открывает три стандартных файла: входной (ему соответствует файловый указатель stdin), выходной (stdout) и файл ошибок (stderr). Обычно stdin соотнесен с клавиатурой, stdout и stderr с экраном.

Файловый указатель ссылается на структуру типа FILE, содержащую следующую информацию о файле:

```
typedef struct {
  int cnt;     /* количество оставшихся литер */
  char *ptr;   /* позиция следующей литеры */
  char *base;  /* адрес буфера */
  int flag;    /* режим доступа */
```

```
  int fd;      /* дескриптор файла */
} FILE;
```

При открытии файла режим доступа может принимать следующие значения:

“r”	- файл открывается только для чтения;
“w”	- файл создается только для записи, при этом, если он уже существовал, его содержимое теряется;
“a”	- файл создается или открывается для записи в конец файла;
“r+”	- файл открывается для чтения и для записи;
“w+”	- файл создается для чтения и для записи, при этом, если он уже существовал, его содержимое теряется;
“a+”	- файл создается или открывается для чтения и для записи в конец файла

После окончания работы файл должен быть закрыт с помощью функции **fclose**, при этом освобождаются все буфера. Если программа завершается нормально, все открытые файлы автоматически закрываются системой.

При работе с файлами существуют различия, определяемые набором функций, используемых для ввода/вывода данных. В частности, можно выделить форматированный ввод – функция

```
int fscanf(FILE *f,const char *format, . . .)
```

читает текстовые данные из входного потока, преобразовывает их в соответствии со спецификациями, содержащимися в формате и присваивает по порядку аргументам, каждый из которых должен быть указателем. Обратные действия производит форматированный вывод

```
int fprintf(FILE *f,const char *format, . . .),
```

который преобразует аргументы в текстовый вид в соответствии с форматом (format) и пишет в выходной поток.

Особо следует рассмотреть работу с файлами, содержащими текстовую информацию, разбитую на строки. Для записи в файл текстовой информации используются функции - fputc и fputs, а для чтения - fgetc и fgets.

Примечание. До сих пор мы рассматривали строку, как последовательность символов, оканчивающуюся нулевым байтом-‘\0’. Строка же в текстовом файле заканчивается специальным символом –‘\n’ и не содержит нулевого байта.

Например:

```
  FILE *fp;
  if((fp=fopen("a.a","w"))!=NULL)
    fputs("str",fp);
```

В данном фрагменте создается файл “a.a” и в него записывается строка “str”. Следует заметить, что функция fputs пишет в файл

символы строки до нулевого байта. Строка может и не содержать символа ‘\n’, и тогда следующий fputs “приклеит” записываемую строку к предыдущей. Например, после fputs(“ing\n”,fp) в файле будет записана строка “string” с ‘\n’ на конце.

Задача 1. Переписать начальные строки (не более 10-ти) из текстового файла “aaa” в конец файла “bbb”. Длина строки не превышает 80 символов (включая ‘\n’).

```
main()
{ FILE *f1,*f2;
  char s[81];
  int i=0;
  if((f1=fopen(“aaa”,”r”))==NULL) exit(1);
  if((f2=fopen(“bbb”,”a”))==NULL) exit(1);
  while(fgets(s,81,f1)!=NULL && i<10) {
    i++;
    fputs(s,f2);
  }
  fclose(f1); fclose(f2);
}
```

Следует помнить, что

```
char *fgets(char *s,int n,FILE *f);
```

читает не более n-1 литер в массив s, прекращая чтение, если встретился ‘\n’, который включается в массив. В любом случае массив дополняется нулевым байтом. Если в решении задачи 1 заменить второй параметр при вызове функции fgets на 80, то строки будут переписываться без искажения, но их количество может оказаться меньше десяти, если в исходном файле окажутся строки максимальной длины.

Примечание. Для текстовых файлов использование функции

```
int feof(FILE *f)
```

имеет некоторые особенности. Если при чтении из файлового потока f достигнут конец файла, то возвращается ненулевое значение, в противном случае возвращается ноль. При этом, если не предпринималась попытка прочитать информацию за концом файла, то функция feof не будет сигнализировать о том, что достигнут конец файла. Например, чтобы скопировать содержимое одного файла в другой можно было бы произвести следующие действия:

```
FILE f1,f2;
char str[80];
    .  .  .
while(!feof(f1)) {
    fgets(str,80,f1);
    fputs(str,f2);
```

```
}
```

Но тогда последняя строка исходного файла будет переписана дважды.

В языке Си предусмотрен также, так называемый прямой ввод-вывод, осуществляемый с помощью функций: fread и fwrite.

Функция

```
size_t fread(void *p,size_t size,size_t nobj,
             FILE *f)
```

читает из входного потока в массив p nobj объектов размером size безо всякого преобразования. Функция возвращает количество прочитанных объектов и ноль, если исчерпан файл.

Обратная ей функция:

```
size_t fwrite(void *p,size_t size,size_t nobj,
              FILE *f)
```

пишет в выходной поток из массива p nobj объектов размером size в непреобразованном виде. Возвращает количество записанных объектов, которое в случае ошибки меньше nobj.

Задача 2. В непустом файле с именем “sss” в непреобразованном(бинарном) виде записаны вещественные числа (double). Поменять местами первое максимальное число с последним минимальным.

```
main()
{  FILE *f;
   double max,min,a;
   long max_p,min_p;
   if((f=fopen("sss","r+"))==NULL) exit(1);
   fread(&a,sizeof(double),1,f);
   max=min=a;
   max_p=min_p=0;
   while (fread(&a,sizeof(double),1,f)) {
     if(a>max){
       max=a;
       max_p=ftell(f)-sizeof(double);
     }
     if(a<=min){
       min=a;
       min_p= ftell(f)-sizeof(double);
     }
   }
   fseek(f,min_p,SEEK_SET);
   fwrite(&max,sizeof(double),1,f);
   fseek(f,max_p,SEEK_SET);
```

```
  fwrite(&min,sizeof(double),1,f);
  fclose(f);
}
```

При работе с однородными бинарными файлами часто возникает задача определения количества объектов, записанных в файл. Например, пусть в бинарном файле записаны числа типа long double.

Для определения количества этих чисел достаточно поставить указатель на конец файла:

```
FILE *f;
long pos,n;
 . . .
fseek(f,0L,SEEK_END);
```

с помощью функции ftell определить номер текущей позиции (нумерация начинается с нуля):

```
pos=ftell(f);
```

и разделить его на размер объекта:

```
n=pos/sizeof(long double);
```

Задачи.

1. Написать функцию сортировки файла, содержащего числа типа long, записанные в файл с помощью функции fwrite, не используя дополнительных файлов. Используя эту функцию отсортировать два файла и слить их в один упорядоченный файл. Результат записать в первый файл.
2. Инвертировать файл, содержащий числа типа int, не используя дополнительных файлов.

   а) Написать функцию для записи в файл анкет студентов. Каждая анкета есть структура:

   struct anketa { char fio[20];
   struct { int day, month, year; } data;
   short m[3];
   };

   в которой поле fio – фамилия студента, поле data – дата рождения, а поле m – оценки, полученные студентом в последнюю сессию.

   b) Написать функцию, которая из файла анкет студентов удаляет всех “двоичников”.
3. Оставить в файле только строки, содержащие некоторую подстроку. Имя файла и подстроку задавать в виде параметров командной строки.

   а)Длина строк не превышает 255 символов.

   б) Длина строк не ограничена.

4. Удалить из файла все строки длиной более пятидесяти символов. Длина строк не ограничена. Имя файла задается в командной строке.
5. Преобразовать файл, усекая строки до 50-ти символов.
6. Упорядочить текстовый файл (лексикографически сравнить строки, длина строки не более 80-ти символов), используя один из методов внешней сортировки.
7. Написать функцию-упаковщик файлов. Коды символов лежат в диапазоне от 0 до 127. Последовательность из n одинаковых символов заменяется на байт со значением –n и байт, содержащий код символа. Рассмотреть два варианта: a) n<=127; b) n – любое.
8. Написать функцию-распаковщик файлов, см. задачу 8.
9. Заменить в файле все “begin” на ‘{‘ ,а “end” на ‘}’.

Приложение 1. Приоритеты и порядок вычисления операторов.

Обозначение	Операция	Порядок выполнения
`( )`	вызов функции	Слева направо
`[ ]`	индексное выражение	
`->  .`	операции доступа к полям записей	
`!`	логическое “НЕ”	**Справа налево**
`~`	поразрядное “НЕ”	
`++  --`	инкрементная и декрементная операции	
`-`	унарный минус	
`(тип)`	приведение к типу	
`*`	косвенная адресация	
`&`	взятие адреса	
`sizeof`	получение размера объекта	
`*  /` `%`	умножение, деление, остаток от деления	Слева направо
`+  -`	сложение, вычитание	Слева направо
`<<  >>`	поразрядные сдвиги	Слева направо
`<  <=` `>  >=`	операции отношения	Слева направо
`==  !=`	операции отношения	Слева направо
`&`	поразрядное “И”	Слева направо
`^`	поразрядное исключающее “ИЛИ”	Слева направо
`|`	поразрядное “ИЛИ”	Слева направо
`&&`	логическое “И”	Слева направо
`||`	логическое “ИЛИ”	Слева направо
`? :`	тернарный оператор	**Справа налево**
`=` `+=  -=` `*=  /=  %=` `<<=  >>=` `&=  ^=  |=`	операции присваивания	**Справа налево**
`,`	операция запятая	Слева направо

Приоритет операций убывает сверху вниз.
Операции из одной ячейки таблицы имеют одинаковый приоритет.

ЛИТЕРАТУРА.

1. Б.Керниган, Д.Ритчи. Язык программирования Си.
   2-ое издание. Москва, Финансы и статистика. 1992г.
2. Б.И.Березин, С.Б.Березин. Начальный курс С и С++.
   Москва, Диалог-МИФИ, 1996г.
3. С.О.Бочков, Д.М.Субботин. Язык программирования Си для персонального компьютера. Москва, СП “Диалог”, Радио и связь, 1990г.
4. И.В.Машечкин, М.И.Петровский, П.Д.Скулачев, А.Н.Терехин. Системное Программное Обеспечение: файловые системы ОС Unix и Windows NT. Москва, Диалог-МГУ,1997г.