А. Богатырёв

Язык Си в системе UNIX

Москва, 1992-1996

0. Напутствие в качестве вступления.

```
Ум подобен желудку.
Важно не то, сколько ты в него вложишь,
а то, сколько он сможет переварить.
```

В этой книге вы найдете ряд задач, примеров, алгоритмов, советов и стилистических замечаний по использованию языка программирования "**C**" (**Си**) в среде операционной системы **UNIX**. Здесь собраны этюды разной сложности и "штрихи к портрету" языка **Си**. Также описаны различные "подводные камни" на которых нередко терпят крушение новички в **Си**. В этом смысле эту книгу можно местами назвать "Как *не надо* программировать на **Си**".

В большинстве случаев в качестве платформы используется персональный компьютер **IBM PC** с какой-либо системой **UNIX**, либо **SPARCstation 20** с системой **Solaris 2** (тоже **UNIX svr4**), но многие примеры без каких-либо изменений (либо с минимумом таковых) могут быть перенесены в среду **MS DOS**£, либо на другой тип машины с системой **UNIX**.

Это ваша ВТОРАЯ книга по Си. Эта книга не учебник, а хрестоматия к учебнику. Она не является ни систематическим курсом по **Си**, ни справочником по нему, и предназначена не для одноразового последовательного прочтения, а для чтения в несколько проходов на разных этапах вашей "зрелости". Поэтому читать ее следует *вместе* с "настоящим" учебником по **Си**, среди которых наиболее известна книга Кернигана и Ритчи.

Эта книга - **не ПОСЛЕДНЯЯ ваша книга по Си**. Во-первых потому, что кое-что в языке все же меняется со временем, хотя и настал час, когда стандарт на язык Си наконец принят... Но появился язык C++, который развивается довольно динамично. Еще есть Objective-C. Во-вторых потому, что есть *библиотеки* и *системные вызовы*, которые развиваются вслед за развитием **UNIX** и других операционных систем. Следующими вашими (настольными) книгами должны стать "Справочное руководство": **man2** (по системным вызовам), **man3** (по библиотечным функциям).

Мощь языка Си - в существующем многообразии библиотек.

Прошу вас с первых же шагов следить за стилем оформления своих программ. Делайте отступы, пишите комментарии, используйте осмысленные имена переменных и функций, отделяйте логические части программы друг от друга пустыми строками. Помните, что "лишние" пробелы и пустые строки в **Си** допустимы везде, кроме изображений констант и имен. Программы на **Си**, набитые в одну колонку (как на **FORTRAN**-е) очень тяжело читать и понимать. Из-за этого бывает трудно находить потерянные скобки { и }, потерянные символы `;' и другие ошибки.

Существует несколько "школ" оформления программ - приглядитесь к примерам в этой книге и в других источниках - и выберите любую! Ничего страшного, если вы будете смешивать эти стили. Но - ПОДАЛЬШЕ ОТ **FORTRAN**-а !!!

Программу можно автоматически сформатировать к "каноническому" виду при помощи, например, программы **cb**.

```
cb < НашФайл.c > /tmp/$$
mv /tmp/$$ НашФайл.c
```

но лучше сразу оформлять программу правильно.

Выделяйте логически самостоятельные ("замкнутые") части программы в функции (даже если они будут вызываться единственный раз). Функции - не просто средство избежать повторения одних и тех же операторов в тексте программы, но и средство структурирования процесса программирования, делающее программу более понятной. Во-первых, вы можете в другой программе использовать текст уже написанной вами ранее функции вместо того, чтобы писать ее заново. Во-вторых, операцию, оформленную в виде функции, можно рассматривать как неделимый примитив (от довольно простого по смыслу, вроде **strcmp**, **strcpy**, до довольно сложного - **qsort**, **malloc**, **gets**) и забыть о его внутреннем устройстве (это хорошо - надо меньше помнить).

Не гонитесь за краткостью в ущерб ясности. **Си** позволяет порой писать такие выражения, над которыми можно полчаса ломать голову. Если же их записать менее мудрено, но чуть длиннее - они самоочевидны (и этим более защищены от ошибок).

В системе **UNIX** вы можете посмотреть описание любой команды системы или функции **Си**, набрав команду

```
man названиеФункции
```

£ **MS DOS** - торговый знак фирмы **Microsoft Corporation**. (читается "Майкрософт"); **DOS** - дисковая операционная система.

(**man** - от слова *manual*, "руководство").

Еще одно напутствие: учите английский язык! Практически все языки программирования используют английские слова (в качестве ключевых слов, терминов, имен переменных и функций). Поэтому лучше понимать значение этих слов (хотя и восприятие их как просто неких символов тоже имеет определенные достоинства). Обратно - программирование на **Си** поможет вам выучить английский.

По различным причинам на территории России сейчас используется много разных восьмибитных русских кодировок. Среди них:

КОИ-8
: Исторически принятая на русских **UNIX** системах - самая ранняя из появившихся. Отличается тем свойством, что если у нее обрезан восьмой бит: *c* & 0177 - то она все же читаема с терминала как транслитерация латинских букв. Именно этой кодировкой пользуется автор этой книги (как и большинство **UNIX**-sites сети RelCom).

ISO 8859/5
: Это американский стандарт на русскую кодировку. А русские программисты к ее разработке не имеют никакого отношения. Ею пользуется большинство коммерческих баз данных.

Microsoft 1251
: Это та кодировка, которой пользуется **Microsoft Windows**. Возможно, что именно к этой кодировке придут и **UNIX** системы (гипотеза 1994 года).

Альтернативная кодировка для *MS DOS*
: Русская кодировка с псевдографикой, использовавшаяся в **MS DOS**.

Кодировка для Macintosh

Это великое "разнообразие" причиняет массу неудобств. Но, господа, это Россия - что значит - широта души и абсолютный бардак. Relax and enjoy.

Многие примеры в данной книге даны вместе с ответами - как образцами для подражания. Однако мы надеемся, что Вы удержитесь от искушения и сначала проверите свои силы, а лишь потом посмотрите в ответ! Итак, читая примеры - делайте по аналогии.

1. Простые программы и алгоритмы. Сюрпризы, советы.

1.1. Составьте программу приветствия с использованием функции **printf**. По традиции принято печатать фразу "*Hello, world !*" ("*Здравствуй, мир !*").

1.2. Найдите ошибку в программе

```
#include <stdio.h>
main(){
    printf("Hello, world\n");
}
```

Ответ: раз не объявлено иначе, функция **main** считается возвращающей целое значение (int). Но функция **main** не возвращает ничего - в ней просто нет оператора **return**. Корректно было бы так:

```
#include <stdio.h>
main(){
    printf("Hello, world\n");
    return 0;
}
```

или

```
#include <stdio.h>
void main(){
    printf("Hello, world\n");
    exit(0);
}
```

а уж совсем корректно - так:

```
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("Hello, world\n");
    return 0;
}
```

1.3. Найдите ошибки в программе

```
#include studio.h
main
{
    int i
    i := 43
    print ('В году i недель')
}
```

1.4. Что будет напечатано в приведенном примере, который является частью полной программы:

```
int n;
n = 2;
printf ("%d + %d = %d\n", n, n, n + n);
```

1.5. В чем состоят ошибки?

```
if( x > 2 )
then    x = 2;
if  x < 1
        x = 1;
```

Ответ: в Си нет ключевого слова **then**, условия в операторах **if**, **while** должны браться в ()-скобки.

1.6. Напишите программу, печатающую ваше имя, место работы и адрес. В первом варианте программы используйте библиотечную функцию **printf**, а во втором - **puts**.

1.7. Составьте программу с использованием следующих постфиксных и префиксных операций:

```
a = b = 5
a + b
a++ + b
++a + b
--a + b
a-- + b
```

Распечатайте полученные значения и проанализируйте результат.

1.8.

Цикл **for**

```
for(INIT; CONDITION; INCR)
        BODY
```

```
        INIT;
repeat:
        if(CONDITION){
                BODY;
        cont:
                INCR;
                goto repeat;
        }
out:    ;
```

Цикл **while**

```
while(COND)
        BODY
```

```
cont:
repeat:
        if(CONDITION){
                BODY;
                goto repeat;
        }
out:    ;
```

Цикл **do**

```
do
        BODY
while(CONDITION)
```

```
cont:
repeat:
        BODY;
        if(CONDITION) goto repeat;
out:    ;
```

В операторах цикла *внутри* тела цикла *BODY* могут присутствовать операторы **break** и **continue**; которые означают на наших схемах следующее:

```
#define break     goto out
#define continue  goto cont
```

1.9. Составьте программу печати прямоугольного треугольника из звездочек

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

используя цикл **for**. Введите переменную, значением которой является размер катета треугольника.

1.10. Напишите операторы Си, которые выдают строку длины *WIDTH*, в которой сначала содержится *x0* символов '-', затем *w* символов '*', и до конца строки - вновь символы '-'. Ответ:

```
int x;
for(x=0; x < x0;     ++x) putchar('-');
for(   ; x < x0 + w; x++) putchar('*');
for(   ; x < WIDTH ; ++x) putchar('-');
putchar('\n');
```

либо

```
for(x=0; x < WIDTH; x++)
        putchar( x < x0     ? '-' :
                 x < x0 + w ? '*' :
                              '-' );
putchar('\n');
```

1.11. Напишите программу с циклами, которая рисует треугольник:

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

Ответ:

```
/* Треугольник из звездочек */
#include <stdio.h>

/* Печать n символов c */
printn(c, n){
        while( --n >= 0 )
                putchar(c);
}

int lines = 10;         /* число строк треугольника */
void main(argc, argv) char *argv[];
{
        register int nline;  /* номер строки */
        register int naster; /* количество звездочек в строке */
        register int i;

        if( argc > 1 )
                lines = atoi( argv[1] );

        for( nline=0; nline < lines ; nline++ ){
                naster = 1 + 2 * nline;

                /* лидирующие пробелы */
                printn(' ', lines-1  - nline);

                /* звездочки */
                printn('*', naster);

                /* перевод строки */
                putchar( '\n' );
        }
        exit(0);        /* завершение программы */
}
```

1.12. В чем состоит ошибка?

```
main(){  /* печать фразы 10 раз */
   int i;
   while(i < 10){
       printf("%d-ый раз\n", i+1);
       i++;
   }
}
```

Ответ: автоматическая переменная *i* не была проинициализирована и содержит не 0, а какое-то произвольное значение. Цикл может выполниться не 10, а *любое* число раз (в том числе и 0 по случайности). Не забывайте инициализировать переменные, возьмите описание с инициализацией за *правило*!

```
int i = 0;
```

Если бы переменная *i* была статической, она бы имела начальное значение 0.

В данном примере было бы еще лучше использовать цикл **for**, в котором все операции над индексом цикла собраны в одном месте - в заголовке цикла:

```
for(i=0; i < 10; i++) printf(...);
```

1.13. Вспомогательные переменные, не несущие смысловой нагрузки (вроде счетчика повторений цикла, не используемого в самом теле цикла) принято по традиции обозначать однобуквенными именами, вроде *i*, *j*. Более того, возможны даже такие курьезы:

```
main(){
  int  _ ;
  for( _ = 0; _ < 10; _++) printf("%d\n", _ );
}
```

основанные на том, что подчерк в идентификаторах - полноправная буква.

1.14. Найдите 2 ошибки в программе:

```
main(){
        int x = 12;

        printf( "x=%d\n" );
        int y;
        y = 2 * x;
        printf( "y=%d\n", y );
}
```

Комментарий: в теле функции все описания должны идти перед всеми выполняемыми операторами (кроме операторов, входящих в состав описаний с инициализацией). Очень часто после внесения правок в программу некоторые описания оказываются после выполняемых операторов. Именно поэтому рекомендуется отделять строки описания переменных от выполняемых операторов *пустыми строками* (в этой книге это часто не делается для экономии места).

1.15. Найдите ошибку:

```
int n;
n = 12;
main(){
        int y;
        y = n+2;
        printf( "%d\n", y );
}
```

Ответ: выполняемый оператор n=12 находится вне тела какой-либо функции. Следует внести его в main() после описания переменной y, либо переписать объявление перед main() в виде

```
int n = 12;
```

В последнем случае присваивание переменной n значения 12 выполнит компилятор еще во время компиляции программы, а не сама программа при своем запуске. Точно так же происходит со всеми статическими данными (описанными как **static**, либо расположенными вне всех функций); причем если их начальное значение не указано явно - то подразумевается 0 ('\0', NULL, ""). Однако нулевые значения не хранятся в скомпилированном выполняемом файле, а требуемая "чистая" память расписывается при старте программы.

1.16. По поводу описания переменной с инициализацией:

```
TYPE x = выражение;
```

является (почти) эквивалентом для

```
TYPE x;          /* описание */
x = выражение;   /* вычисление начального значения */
```

Рассмотрим пример:

```
#include <stdio.h>
extern double sqrt();   /* квадратный корень */
double x   = 1.17;
double s12 = sqrt(12.0);            /* #1 */
double y   = x * 2.0;               /* #2 */
FILE  *fp  = fopen("out.out", "w"); /* #3 */
main(){
  double ss = sqrt(25.0) + x;       /* #4 */
  ...
}
```

Строки с метками #1, #2 и #3 ошибочны. Почему?

Ответ: при инициализации статических данных (а *s12*, *y* и *fp* таковыми и являются, так как описаны вне какой-либо функции) выражение должно содержать только константы, поскольку оно вычисляется КОМПИЛЯТОРОМ. Поэтому ни использование значений переменных, ни вызовы функций здесь недопустимы (но можно брать адреса от переменных).

В строке #4 мы инициализируем автоматическую переменную *ss*, т.е. она отводится уже во время *выполнения* программы. Поэтому выражение для инициализации вычисляется уже не компилятором, а самой программой, что дает нам право использовать переменные, вызовы функций и.т.п., то есть выражения языка Си без ограничений.

1.17. Напишите программу, реализующую эхо-печать вводимых символов. Программа должна завершать работу при получении признака **EOF**. В **UNIX** при вводе с клавиатуры признак **EOF** обычно обозначается одновременным нажатием клавиш **CTRL** и **D** (**CTRL** чуть раньше), что в дальнейшем будет обозначаться **CTRL/D**; а в **MS DOS** - клавиш **CTRL/Z**. Используйте **getchar**() для ввода буквы и **putchar**() для вывода.

1.18. Напишите программу, подсчитывающую число символов поступающих со стандартного ввода. Какие достоинства и недостатки у следующей реализации:

```
#include <stdio.h>
main(){ double cnt = 0.0;
        while (getchar() != EOF) ++cnt;
        printf("%.0f\n", cnt );
}
```

Ответ: и достоинство и недостаток в том, что счетчик имеет тип **double**. Достоинство - можно подсчитать *очень* большое число символов; недостаток - операции с **double** обычно выполняются *гораздо медленнее*, чем с **int** и **long** (до десяти раз), программа будет работать дольше. В повседневных задачах вам вряд ли понадобится иметь счетчик, отличный от **long** *cnt*; (печатать его надо по формату "**%ld**").

1.19. Составьте программу перекодировки вводимых символов со стандартного ввода по следующему правилу:

```
        a -> b
        b -> c
        c -> d
        ...
        z -> a
 другой символ -> *
```

Коды строчных латинских букв расположены подряд по возрастанию.

1.20. Составьте программу перекодировки вводимых символов со стандартного ввода по следующему правилу:

```
        B -> A
        C -> B
        ...
        Z -> Y
 другой символ -> *
```

Коды прописных латинских букв также расположены по возрастанию.

1.21. Напишите программу, печатающую номер и код введенного символа в восьмеричном и шестнадцатеричном виде. Заметьте, что если вы наберете на вводе *строку* символов и нажмете клавишу **ENTER**, то программа напечатает вам на один символ больше, чем вы набрали. Дело в том, что код клавиши **ENTER**, завершившей ввод строки - символ '\n' - *тоже* попадает в вашу программу (на экране он отображается как перевод курсора в начало следующей строки!).

1.22. Разберитесь, в чем состоит разница между символами '**0**' (цифра нуль) и '**0**' (нулевой байт). Напечатайте

```
  printf( "%d %d %c\n", '\0', '0', '0' );
```

Поставьте опыт: что печатает программа?

```
main(){
        int c = 060;  /* код символа '0' */
        printf( "%c %d %o\n", c, c, c);
}
```

Почему печатается 0 48 60? Теперь напишите вместо
```
 int c = 060;
```
строчку
```
 char c = '0';
```

1.23. Что напечатает программа?

```
#include <stdio.h>
void main(){
        printf("ab\0cd\nxyz");
        putchar('\n');
}
```

Запомните, что '\0' служит признаком конца строки в памяти, а '\n' - в файле. Что в строке "abcd\n" на конце неявно уже расположен нулевой байт:

```
'a','b','c','d','\n','\0'
```

Что строка "ab\0cd\nxyz" - это

```
'a','b','\0','c','d','\n','x','y',z','\0'
```

Что строка "abcd\0" - избыточна, поскольку будет иметь на конце два нулевых байта (что не вредно, но зачем?). Что **printf** печатает строку до нулевого байта, а не до закрывающей кавычки.

Программа эта напечатает *ab* и перевод строки.

Вопрос: чему равен **sizeof**("ab\0cd\nxyz")? Ответ: 10.

1.24. Напишите программу, печатающую целые числа от 0 до 100.

1.25. Напишите программу, печатающую квадраты и кубы целых чисел.

1.26. Напишите программу, печатающую сумму квадратов первых n целых чисел.

1.27. Напишите программу, которая переводит секунды в дни, часы, минуты и секунды.

1.28. Напишите программу, переводящую скорость из километров в час в метры в секундах.

1.29. Напишите программу, шифрующую текст файла путем замены значения символа (например, значение символа *C* заменяется на *C*+1 или на ~*C*).

1.30. Напишите программу, которая при введении с клавиатуры буквы печатает на терминале ключевое слово, начинающееся с данной буквы. Например, при введении буквы '**b**' печатает "**break**".

1.31. Напишите программу, отгадывающую задуманное вами число в пределах от 1 до 200, пользуясь подсказкой с клавиатуры "=" (равно), "<" (меньше) и ">" (больше). Для угадывания числа используйте метод деления пополам.

1.32. Напишите программу, печатающую степени двойки

```
1, 2, 4, 8, ...
```

Заметьте, что, начиная с некоторого n, результат становится отрицательным из-за переполнения целого.

1.33. Напишите подпрограмму вычисления квадратного корня с использованием метода касательных (Ньютона):

$$x(0) = a$$

$$x(n+1) = \frac{1}{2} * \left(\frac{a}{x(n)} + x(n) \right)$$

Итерировать, пока не будет $| x(n+1) - x(n) | < 0.001$

Внимание! В данной задаче массив не нужен. Достаточно хранить текущее и предыдущее значения *x* и обновлять их после каждой итерации.

1.34. Напишите программу, распечатывающую простые числа до 1000.

```
1, 2, 3, 5, 7, 11, 13, 17, ...
```

```
/*#!/bin/cc primes.c -o primes -lm
 *        Простые числа.
 */
#include <stdio.h>
#include <math.h>
int debug = 0;

/* Корень квадратный из числа по методу Ньютона */
#define eps 0.0001
double  sqrt (x) double x;
{
    double  sq, sqold, EPS;

    if (x < 0.0)
        return -1.0;
    if (x == 0.0)
        return 0.0;  /* может привести к делению на 0 */
    EPS = x * eps;
    sq = x;
    sqold = x + 30.0;          /* != sq */
    while (fabs (sq * sq - x) >= EPS) {
    /*     fabs( sq - sqold )>= EPS    */
        sqold = sq;
        sq = 0.5 * (sq + x / sq);
    }
    return sq;
}

/* таблица простых чисел */
int is_prime (t) register int    t; {
    register int    i, up;
    int             not_div;

    if (t == 2 || t == 3 || t == 5 || t == 7)
        return 1;               /* prime */
    if (t % 2 == 0 || t == 1)
        return 0;               /* composite */
    up = ceil (sqrt ((double) t)) + 1;
    i = 3;
    not_div = 1;
    while (i <= up && not_div) {
        if (t % i == 0) {
            if (debug)
                fprintf (stderr, "%d поделилось на %d\n",
                                  t,               i);
            not_div = 0;
            break;
        }
        i += 2;  /*
                  * Нет смысла проверять четные,
                  * потому что если делится на 2*n,
                  * то делится и на 2,
                  * а этот случай уже обработан выше.
                  */
    }
    return not_div;
}
```

```
#define COL 6
int     n;
main (argc, argv) char **argv;
{
    int     i,
            j;
    int     n;

    if( argc < 2 ){
        fprintf( stderr, "Вызов: %s число [-]\n", argv[0] );
        exit(1);
    }
    i = atoi (argv[1]); /* строка -> целое, ею изображаемое */
    if( argc > 2 ) debug = 1;

    printf ("\t*** Таблица простых чисел от 2 до %d ***\n", i);
    n = 0;
    for (j = 1; j <= i; j++)
        if (is_prime (j)){

            /* распечатка в COL колонок */
            printf ("%3d%s", j, n == COL-1 ? "\n" : "\t");
            if( n == COL-1 ) n = 0;
            else             n++;
        }
    printf( "\n---\n" );
    exit (0);
}
```

1.35. Составьте программу ввода двух комплексных чисел в виде A + B * I (каждое на отдельной строке) и печати их произведения в том же виде. Используйте **scanf** и **printf**. Перед тем, как использовать **scanf**, проверьте себя: что неверно в нижеприведенном операторе?

```
int x;
scanf( "%d", x );
```

Ответ: должно быть написано "АДРЕС от *x*", то есть **scanf**("%d", **&***x*);

1.36. Напишите подпрограмму вычисления корня уравнения **f**(*x*)=0 методом деления отрезка пополам. Приведем реализацию этого алгоритма для поиска целочисленного квадратного корня из целого числа (этот алгоритм может использоваться, например, в машинной графике при рисовании дуг):

```
/* Максимальное unsigned long число */
#define MAXINT (~0L)
/* Определим имя-синоним для типа unsigned long */
typedef unsigned long ulong;
/* Функция, корень которой мы ищем: */
#define FUNC(x, arg)  ((x) * (x) - (arg))
/* тогда x*x - arg = 0 означает  x*x = arg, то есть
 * x = корень_квадратный(arg)     */
/* Начальный интервал. Должен выбираться исходя из
 * особенностей функции FUNC */
#define  LEFT_X(arg)  0
#define RIGHT_X(arg) (arg > MAXINT)? MAXINT : (arg/2)+1;

/* КОРЕНЬ КВАДРАТНЫЙ, округленный вниз до целого.
 * Решается по методу деления отрезка пополам:
 *    FUNC(x, arg) = 0;       x = ?
 */
ulong i_sqrt( ulong arg ) {
   register ulong   mid,   /* середина интервала    */
                    rgt,   /* правый край интервала */
                    lft;   /* левый край интервала  */
   lft = LEFT_X(arg); rgt = RIGHT_X(arg);

   do{ mid = (lft + rgt + 1 )/2;
/* +1 для ошибок округления при целочисленном делении */
```

```
        if( FUNC(mid, arg) > 0 ){
               if( rgt == mid ) mid--;
               rgt =  mid ;  /* приблизить правый край */
        } else lft =  mid ;  /* приблизить левый край  */
    } while( lft < rgt );
    return mid;
}
void main(){ ulong i;
    for(i=0; i <= 100; i++)
        printf("%ld -> %lu\n", i, i_sqrt(i));
}
```

Использованное нами при объявлении переменных ключевое слово **register** означает, что переменная является ЧАСТО ИСПОЛЬЗУЕМОЙ, и компилятор должен попытаться разместить ее на регистре процессора, а не в стеке (за счет чего увеличится скорость обращения к этой переменной). Это слово используется как

```
register тип переменная;
register переменная; /* подразумевается тип int */
```

От регистровых переменных нельзя брать адрес: **&***переменная* ошибочно.

1.37. Напишите программу, вычисляющую числа треугольника Паскаля и печатающую их в виде треугольника.

```
C(0,n)   = C(n,n)   = 1          n = 0...
C(k,n+1) = C(k-1,n) + C(k,n)     k = 1..n
n - номер строки
```

В разных вариантах используйте циклы, рекурсию.

1.38. Напишите функцию вычисления определенного интеграла методом Монте-Карло. Для этого вам придется написать генератор случайных чисел. Си предоставляет стандартный датчик ЦЕЛЫХ равномерно распределенных псевдослучайных чисел: если вы хотите получить целое число из интервала [A..B], используйте

```
int x = A + rand() % (B+1-A);
```

Чтобы получать *разные* последовательности следует задавать некий начальный параметр последовательности (это называется "рандомизация") при помощи

```
srand( число ); /* лучше нечетное */
```

Чтобы повторить *одну и ту же* последовательность случайных чисел несколько раз, вы должны поступать так:

```
srand(NBEG); x=rand(); ... ; x=rand();
/* и повторить все сначала */
srand(NBEG); x=rand(); ... ; x=rand();
```

Используемый метод получения случайных чисел таков:

```
static unsigned long int next = 1L;
int rand(){
  next = next * 1103515245 + 12345;
  return ((unsigned int)(next/65536) % 32768);
}
void srand(seed) unsigned int seed;
{     next = seed;    }
```

Для рандомизации часто пользуются таким приемом:

```
char t[sizeof(long)];
time(t); srand(t[0] + t[1] + t[2] + t[3] + getpid());
```

1.39. Напишите функцию вычисления определенного интеграла по методу Симпсона.

```
/*#!/bin/cc $* -lm
 * Вычисление интеграла по методу Симпсона
 */
#include <math.h>

extern double integral(), sin(), fabs();
#define PI 3.141593

double myf(x) double x;
{       return sin(x / 2.0);       }

int niter;  /* номер итерации */

void main(){
        double integral();

        printf("%g\n", integral(0.0, PI, myf, 0.000000001));
        /* Заметьте, что myf, а не myf().
         * Точное значение интеграла равно 2.0
         */
        printf("%d итераций\n", niter );
}
```

```
double integral(a, b, f, eps)
        double a, b;    /* концы отрезка */
        double eps;     /* требуемая точность */
        double (*f)();  /* подынтегральная функция */
{
        register long i;
        double fab = (*f)(a) + (*f)(b); /* сумма на краях */
        double h, h2;   /* шаг и удвоенный шаг */
        long n, n2;     /* число точек разбиения и оно же удвоенное */
        double Sodd, Seven;  /* сумма значений f в нечетных и в
                                четных точках */
        double S, Sprev;/* значение интеграла на данной
                           и на предыдущей итерациях */
        double x;       /* текущая абсцисса */

        niter = 0;
        n = 10L;        /* четное число */
        n2 = n * 2;

        h = fabs(b - a) / n2;   h2 = h * 2.0;

        /* Вычисляем первое приближение */
        /* Сумма по нечетным точкам: */
        for( Sodd = 0.0, x = a+h, i = 0;
                                  i < n;
                                  i++, x += h2 )
                Sodd += (*f)(x);

        /* Сумма по четным точкам: */
        for( Seven = 0.0, x = a+h2, i = 0;
                                   i < n-1;
                                   i++, x += h2 )
                Seven += f(x);

        /* Предварительное значение интеграла: */
        S = h / 3.0 * (fab + 4.0 * Sodd + 2.0 * Seven );
        do{
                niter++;
                Sprev = S;

                /* Вычисляем интеграл с половинным шагом */
                h2 = h;       h /= 2.0;
                if( h == 0.0 ) break;   /* потеря значимости */
                n  = n2;      n2 *= 2;

                Seven = Seven + Sodd;
                /* Вычисляем сумму по новым точкам: */
                for( Sodd = 0.0, x = a+h, i = 0;
                                          i < n;
                                          i++, x += h2 )
                Sodd += (*f)(x);

                /* Значение интеграла */
                S = h / 3.0 * (fab + 4.0 * Sodd + 2.0 * Seven );

        } while( niter < 31 && fabs(S - Sprev) / 15.0 >= eps );
        /* Используем условие Рунге для окончания итераций */

        return ( 16.0 * S - Sprev ) / 15.0 ;
        /* Возвращаем уточненное по Ричардсону значение */
}
```

1.40. Где ошибка?

```
struct time_now{
  int  hour, min, sec;
} X = { 13, 08, 00 }; /* 13 часов 08 минут 00 сек.*/
```

Ответ: 08 - восьмеричное число (так как начинается с нуля)! А в восьмеричных числах цифры 8 и 9 не бывают.

1.41. Дан текст:

```
   int i = -2;
   i <<= 2;
   printf("%d\n", i); /* печать сдвинутого i : -8 */
   i >>= 2;
   printf("%d\n", i); /* печатается -2 */
```

Закомментируем две строки (исключая их из программы):

```
   int i = -2;
   i <<= 2;
/*
   printf("%d\n", i); /* печать сдвинутого i : -8 */
   i >>= 2;
*/
   printf("%d\n", i); /* печатается -2 */
```

Почему теперь возникает ошибка? Указание: где кончается комментарий?

Ответ: Си не допускает вложенных комментариев. Вместо этого часто используются конструкции вроде:

```
#ifdef COMMENT
  ... закомментированный текст ...
#endif /*COMMENT*/
```

и вроде

```
/**/ printf("here");/* отладочная выдача включена  */
/*   printf("here");/* отладочная выдача выключена */
```

или

```
/* выключено();    /**/
   включено();     /**/
```

А вот дешевый способ быстро исключить оператор (с возможностью восстановления) - конец комментария занимает отдельную строку, что позволяет отредактировать такой текст редактором почти не сдвигая курсор:

```
   /*printf("here");
    */
```

1.42. Почему программа печатает неверное значение для i2 ?

```
int main(int argc, char *argv[]){
        int i1, i2;

        i1 = 1;         /* Инициализируем i1 /
        i2 = 2;         /* Инициализируем i2 */
        printf("Numbers %d %d\n", i1, i2);
        return(0);
}
```

Ответ: в первом операторе присваивания не закрыт комментарий - весь второй оператор присваивания полностью проигнорировался! Правильный вариант:

```
int main(int argc, char *argv[]){
        int i1, i2;

        i1 = 1;         /* Инициализируем i1 */
        i2 = 2;         /* Инициализируем i2 */
        printf("Numbers %d %d\n", i1, i2);
        return(0);
}
```

1.43. А вот "шальной" комментарий.

```
void main(){
        int n    = 10;
        int *ptr = &n;
        int x, y = 40;

        x = y/*ptr     /* должно быть 4 */  + 1;
        printf( "%d\n", x );    /* пять */
        exit(0);
}

/* или такой пример из жизни - взят из переписки в Relcom */
...
cost = nRecords/*pFactor     /* divided by Factor, and  */
       + fixMargin;          /* plus the precalculated  */
...
```

Результат непредсказуем. Дело в том, что y/*ptr превратилось в начало комментария! Поэтому бинарные операции принято окружать пробелами.

```
x = y / *ptr   /* должно быть 4 */  + 1;
```

1.44. Найдите ошибки в директивах препроцессора Си † (вертикальная черта обозначает левый край файла).

```
|
| #include <stdio.h>
|#include  < sys/types.h >
|#   define inc (x) ((x) + 1)
|#define N 12;
|#define X -2
|
|...  printf( "n=%d\n", N );
|...  p = 4-X;
```

Ответ: в первой директиве стоит пробел перед **#**. Диез должен находиться в первой позиции строки. Во второй директиве в <> находятся лишние пробелы, не относящиеся к имени файла - препроцессор не найдет такого файла! В данном случае "красота" пошла во вред делу. В третьей - между именем макро **inc** и его аргументом в круглых скобках (*x*) стоит пробел, который изменяет весь смысл макроопределения: вместо *макроса с параметром* **inc**(*x*) мы получаем, что *слово* **inc** будет заменяться на (*x*)((*x*)+1). Заметим однако, что пробелы после **#** перед именем директивы вполне допустимы. В четвертом случае показана характерная опечатка - символ **;** после определения. В результате написанный **printf**() заменится на

```
printf( "n=%d\n", 12; );
```

где лишняя **;** даст синтаксическую ошибку.

В пятом случае ошибки нет, но нас ожидает неприятность в строке p=4-X; которая расширится в строку p=4--2; являющуюся синтаксически неверной. Чтобы избежать подобной ситуации, следовало бы написать

```
p = 4 - X;  /* через пробелы */
```

но еще проще (и лучше) взять макроопределение в скобки:

```
#define X (-2)
```

1.45. Напишите функцию **max**(x, y), возвращающую большее из двух значений. Напишите аналогичное макроопределение. Напишите макроопределения **min**(x, y) и **abs**(x) (abs - модуль числа). Ответ:

```
#define abs(x)   ((x)  <  0   ? -(x) : (x))
#define min(x,y) (((x) < (y)) ?  (x) : (y))
```

Зачем x взят в круглые скобки (x)? Предположим, что мы написали

---

† Препроцессор Си - это программа **/lib/cpp**

```
#define abs(x)  (x < 0 ? -x : x )
            вызываем
abs(-z)                  abs(a|b)
            получаем
(-z < 0 ? --z : -z )     (a|b < 0 ? -a|b : a|b )
```

У нас появилась "дикая" операция --z; а выражение a|b<0 соответствует a|(b<0), с совсем другим порядком операций! Поэтому заключение всех аргументов макроса в его теле в круглые скобки позволяет избежать многих неожиданных проблем. Придерживайтесь этого правила!

Вот пример, показывающий зачем полезно брать в скобки *все* определение:

```
#define div(x, y)     (x)/(y)
```

При вызове

```
z = sizeof div(1, 2);
    превратится в
z = sizeof(1) / (2);
```

что равно sizeof(int)/2, а не sizeof(int). Вариант

```
#define div(x, y) ((x) / (y))
```

будет работать правильно.

1.46. Макросы, в отличие от функций, могут порождать непредвиденные побочные эффекты:

```
int sqr(int x){ return x  *  x; }
#define SQR(x)       ((x) * (x))
main(){ int y=2, z;
   z = sqr(y++); printf("y=%d z=%d\n", y, z);
   y = 2;
   z = SQR(y++); printf("y=%d z=%d\n", y, z);
}
```

Вызов функции **sqr** печатает "y=3 z=4", как мы и ожидали. Макрос же **SQR** расширяется в

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

и результатом будет "y=4 z=6", где *z* совсем не похоже на квадрат числа 2.

1.47. ANSI препроцессор† языка Си имеет оператор **##** - "склейка лексем":

```
#define VAR(a, b)       a ## b
#define CV(x)           command_ ## x
main(){
  int VAR(x, 31) = 1;
  /* превратится в int x31 = 1; */
  int CV(a) = 2; /* даст int command_a = 2; */
  ...
}
```

Старые версии препроцессора не обрабатывают такой оператор, поэтому раньше использовался такой трюк:

```
#define VAR(a, b)       a/**/b
```

в котором предполагается, что препроцессор удаляет комментарии из текста, не заменяя их на пробелы. Это не всегда так, поэтому такая конструкция не мобильна и пользоваться ею не рекомендуется.

1.48. Напишите программу, распечатывающую максимальное и минимальное из ряда чисел, вводимых с клавиатуры. Не храните вводимые числа в массиве, вычисляйте *max* и *min* сразу при вводе очередного числа!

---

† **ANSI** - American National Standards Institute, разработавший стандарт на язык Си и его окружение.

```
#include <stdio.h>
main(){
  int max, min, x, n;
  for( n=0; scanf("%d", &x) != EOF; n++)
        if( n == 0 ) min = max = x;
        else{
           if( x > max ) max = x;
           if( x < min ) min = x;
        }
  printf( "Ввели %d чисел: min=%d max=%d\n",
                 n,            min,   max);
}
```

Напишите аналогичную программу для поиска максимума и минимума среди элементов массива, изначально *min=max=array*[0];

1.49. Напишите программу, которая сортирует массив заданных чисел по возрастанию (убыванию) методом пузырьковой сортировки. Когда вы станете более опытны в Си, напишите сортировку методом Шелла.

```
/*
 * Сортировка по методу Шелла.
 * Сортировке подвергается массив указателей на данные типа obj.
 *      v------.-------.------.-------.------0
 *             !       !      !       !
 *             *       *      *       *
 *           элементы типа obj
 * Программа взята из книги Кернигана и Ритчи.
 */

#include <stdio.h>
#include <string.h>
#include <locale.h>
#define obj char

static shsort (v,n,compare)
int n;              /* длина массива */
obj *v[];           /* массив указателей */
int (*compare)();   /* функция сравнения соседних элементов */
{
        int g,      /* расстояние, на котором происходит сравнение */
            i,j;    /* индексы сравниваемых элементов */
        obj *temp;

        for( g = n/2 ; g > 0  ; g /= 2 )
        for( i = g   ; i < n  ; i++    )
        for( j = i-g ; j >= 0 ; j -= g )
        {
                if((*compare)(v[j],v[j+g]) <= 0)
                    break;      /* уже в правильном порядке */

                /* обменять указатели */
                temp = v[j]; v[j] = v[j+g]; v[j+g] = temp;
                /* В качестве упражнения можете написать
                 * при помощи curses-а программу,
                 * визуализирующую процесс сортировки:
                 * например, изображающую эту перестановку
                 * элементов массива */
        }
}
```

```
/* сортировка строк */
ssort(v) obj **v;
{
        extern less();  /* функция сравнения строк */
        int len;

        /* подсчет числа строк */
        len=0;
        while(v[len]) len++;
        shsort(v,len,less);
}

/* Функция сравнения строк.
 * Вернуть целое меньше нуля, если a <  b
 *                       ноль, если a == b
 *                больше нуля, если a >  b
 */
less(a,b) obj *a,*b;
{
        return strcoll(a,b);
        /* strcoll - аналог strcmp,
         * но с учетом алфавитного порядка букв.
         */
}

char *strings[] = {
        "Яша", "Федя", "Коля",
        "Гриша", "Сережа", "Миша",
        "Андрей Иванович", "Васька",
        NULL
};
int main(){
        char **next;

        setlocale(LC_ALL, "");

        ssort( strings );
        /* распечатка */
        for( next = strings ; *next ; next++ )
                printf( "%s\n", *next );
        return 0;
}
```

1.50. Реализуйте алгоритм быстрой сортировки.

```
/* Алгоритм быстрой сортировки. Работа алгоритма "анимируется"
 * (animate-оживлять) при помощи библиотеки curses.
 *      cc -o qsort qsort.c -lcurses -ltermcap
 */
#include "curses.h"

#define N 10     /* длина массива */

/* массив, подлежащий сортировке */
int target [N] = {
        7, 6, 10, 4, 2,
        9, 3,  8, 5, 1
};

int maxim;      /* максимальный элемент массива */

/* quick sort */
qsort (a, from, to)
        int a[];      /* сортируемый массив */
        int from;     /* левый начальный индекс */
        int to;       /* правый конечный индекс */
{
        register i, j, x, tmp;

        if( from >= to ) return;
        /* число элементов <= 1 */

        i = from; j = to;
        x = a[ (i+j) / 2 ];     /* значение из середины */

        do{
                /* сужение вправо */
                while( a[i] < x ) i++ ;

                /* сужение влево */
                while( x < a[j] ) j--;

                if( i <= j ){   /* обменять */
                        tmp = a[i]; a[i] = a[j] ; a[j] = tmp;
                        i++;  j--;

                        demochanges();  /* визуализация */
                }
        } while( i <= j );

        /* Теперь обе части сошлись в одной точке.
         * Длина левой части = j - from + 1
         *       правой      = to - i   + 1
         * Все числа в левой части меньше всех чисел в правой.
         * Теперь надо просто отсортировать каждую часть в отдельности.
         * Сначала сортируем более короткую (для экономии памяти
         * в стеке ). Рекурсия:
         */
        if( (j - from)  <  (to - i) ){
                qsort( a, from, j );
                qsort( a, i, to   );
        } else {
                qsort( a, i,   to );
                qsort( a, from, j );
        }
}

int main (){
        register i;

        initscr();      /* запуск curses-а */
```

```
        /* поиск максимального числа в массиве */
        for( maxim = target[0], i = 1 ;  i < N ; i++ )
                if( target[i] > maxim )
                    maxim = target[i];

        demochanges();
        qsort( target, 0, N-1 );
        demochanges();

        mvcur( -1, -1, LINES-1, 0);
        /* курсор в левый нижний угол */

        endwin();       /* завершить работу с curses-ом */
        return 0;
}

#define GAPY  2
#define GAPX 20

/* нарисовать картинку */
demochanges(){
        register i, j;
        int h = LINES - 3 * GAPY - N;
        int height;

        erase();        /* зачистить окно */
        attron( A_REVERSE );

        /* рисуем матрицу упорядоченности */
        for( i=0 ; i < N ; i++ )
                for( j = 0; j < N ; j++ ){
                    move( GAPY + i , GAPX + j * 2 );
                    addch( target[i] >= target[j] ? '*' : '.' );
                    addch( ' ' );
                    /* Рисовать '*' если элементы
                     * идут в неправильном порядке.
                     * Возможен вариант проверки target[i] > target[j]
                     */
                }
        attroff( A_REVERSE );

        /* массив */
        for( i = 0 ; i < N ; i++ ){
                move( GAPY + i , 5 );
                printw( "%4d", target[i] );

                height = (long) h * target[i] / maxim ;
                for( j = 2 * GAPY + N + (h - height) ;
                                        j < LINES - GAPY; j++ ){
                        move( j, GAPX + i * 2 );
                        addch( '|' );
                }
        }
        refresh();      /* проявить картинку */
        sleep(1);
}
```

1.51. Реализуйте приведенный фрагмент программы без использования оператора **goto** и без меток.

```
        if ( i > 10 ) goto M1;
        goto M2;
        M1: j = j + i; flag = 2; goto M3;
        M2: j = j - i; flag = 1;
        M3:       ;
```

Заметьте, что помечать можно только оператор (может быть пустой); поэтому не может встретиться фрагмент

```
{ ..... Label:  }  а только { ..... Label: ; }
```

1.52. В каком случае оправдано использование оператора **goto**?

Ответ: при выходе из вложенных циклов, т.к. оператор **break** позволяет выйти только из самого внутреннего цикла (на один уровень).

1.53. К какому **if**-у относится **else**?

```
if(...) ... if(...) ... else ...
```

Ответ: ко второму (к ближайшему предшествующему, для которого нет другого **else**). Вообще же лучше явно расставлять скобки (для ясности):

```
if(...){ ... if(...) ...   else ... }
if(...){ ... if(...) ... } else ...
```

1.54. Макроопределение, чье тело представляет собой последовательность операторов в {...} скобках (блок), может вызвать проблемы при использовании его в условном операторе **if** с **else**-частью:

```
#define MACRO   { x=1; y=2; }

if(z)   MACRO;
else    .......;
```

Мы получим после макрорасширения

```
if(z)   { x=1; y=2; } /* конец if-а */ ;
else    .......;      /* else ни к чему не относится */
```

то есть синтаксически ошибочный фрагмент, так как должно быть либо

```
if(...)  один_оператор;
else     .....
         либо
if(...){ последовательность; ...; операторов; }
else     .....
```

где точка-с-запятой после **}** не нужна. С этим явлением борются, оформляя блок {...} в виде **do**{...}**while**(0)

```
#define MACRO do{ x=1; y=2; }while(0)
```

Тело такого "цикла" выполняется единственный раз, при этом мы получаем правильный текст:

```
if(z)   do{ x=1; y=2; }while(0);
else    .......;
```

1.55. В чем ошибка (для знающих язык "Паскаль")?

```
int  x = 12;
     if( x < 20 and x > 10 ) printf( "O'K\n");
else if( x > 100 or x < 0  ) printf( "Bad x\n");
else                         printf( "x=%d\n", x);
```

Напишите

```
#define and &&
#define or  ||
```

1.56. Почему программа зацикливается? Мы хотим подсчитать число пробелов и табуляций в начале строки:

```
int i = 0;
char *s = "   3 spaces";
while(*s == ' ' || *s++ == '\t')
        printf( "Пробел %d\n", ++i);
```

Ответ: логические операции || и && выполняются слева направо; как только какое-то условие в || оказывается истинным (а в && ложным) - дальнейшие условия просто *не вычисляются*. В нашем случае условие *s==' ' сразу же верно, и операция *s++ из второго условия не выполняется! Мы должны были написать хотя бы так:

```
while(*s == ' ' || *s == '\t'){
        printf( "Пробел %d\n", ++i); s++;
}
```

С другой стороны, это свойство || и && черезвычайно полезно, например:

```
if( x != 0.0 && y/x < 1.0 ) ... ;
```

Если бы мы не вставили проверку на 0, мы могли бы получить деление на 0. В данном же случае при *x*==0 деление просто не будет вычисляться. Вот еще пример:

```
int a[5], i;
for(i=0; i < 5 && a[i] != 0; ++i) ...;
```

Если *i* выйдет за границу массива, то сравнение *a*[*i*] с нулем уже не будет вычисляться, т.е. попытки прочесть элемент не входящий в массив не произойдет.

Это свойство && позволяет писать довольно неочевидные конструкции, вроде

```
if((cond) && f());
    что оказывается эквивалентным
if( cond ) f();
```

Вообще же

```
if(C1 && C2 && C3) DO;
        эквивалентно
if(C1) if(C2) if(C3) DO;
```

и для "или"

```
if(C1 || C2 || C3) DO;
        эквивалентно
     if(C1) goto ok;
else if(C2) goto ok;
else if(C3){ ok: DO; }
```

Вот еще пример, пользующийся этим свойством ||

```
#include <stdio.h>
main(argc, argv) int argc; char *argv[];
{  FILE *fp;
   if(argc < 2 || (fp=fopen(argv[1], "r")) == NULL){
        fprintf(stderr, "Плохое имя файла\n");
        exit(1); /* завершить программу */
   }
   ...
}
```

Если *argc*==1, то *argv*[1] не определено, однако в этом случае попытки открыть файл с именем *argv*[1] просто не будет предпринято!

Ниже приведен еще один содержательный пример, представляющий собой одну из возможных схем написания "двуязычных" программ, т.е. выдающих сообщения на одном из двух языков по вашему желанию. Проверяется переменная окружения **MSG** (или **LANG**):

```
                              ЯЗЫК:
1)  "MSG=engl"               английский
2)   MSG нет в окружении     английский
3)  "MSG=rus"                русский
```

Про окружение и функцию **getenv**() смотри в главе "Взаимодействие с UNIX", про **strchr**() - в главе "Массивы и строки".

```
#include <stdio.h>
int _ediag = 0; /* язык диагностик: 1-русский */
extern char *getenv(), *strchr();
#define ediag(e,r) (_ediag?(r):(e))
main(){  char *s;
_ediag = ((s=getenv("MSG"))   != NULL   &&
          strchr("rRpP", *s) != NULL);
printf(ediag("%d:english\n", "%d:русский\n"), _ediag);
}
```

Если переменная **MSG** не определена, то *s*==NULL и функция **strchr**(*s*,...) *не вызывается* (ее первый фргумент не должен быть NULL-ом). Здесь ее можно было бы упрощенно заменить на *s=='r'; тогда если *s* равно NULL, то обращение **s* было бы незаконно (обращение по указателю NULL дает непредсказуемые

результаты и, скорее всего, вызовет крах программы).

1.57. Иногда логическое условие можно сделать более понятным, используя *правила де-Моргана*:

```
a && b     =     ! ( !a || !b )
a || b     =     ! ( !a && !b )
```

а также учитывая, что

```
! !a       =     a
! (a == b) =    (a != b)
```

Например:

```
if( c != 'a' && c != 'b' && c != 'c' )...;
        превращается в
if( !(c == 'a' || c == 'b' || c == 'c')) ...;
```

1.58. Пример, в котором используются побочные эффекты вычисления *выражений*. Обычно значение выражения присваивается некоторой переменной, но это не необходимо. Поэтому можно использовать свойства вычисления && и || в выражениях (хотя это не есть самый понятный способ написания программ, скорее некоторый род извращения). Ограничение тут таково: все части выражения должны возвращать значения.

```
#include <stdio.h>
extern int errno;       /* код системной ошибки */
FILE *fp;

int openFile(){
        errno = 0;
        fp = fopen("/etc/inittab", "r");
        printf("fp=%x\n", fp);
        return(fp == NULL ? 0 : 1);
}
int closeFile(){
        printf("closeFile\n");
        if(fp) fclose(fp);
        return 0;
}

int die(int code){
        printf("exit(%d)\n", code);
        exit(code);
        return 0;
}

void main(){
        char buf[2048];

        if( !openFile()) die(errno); closeFile();
        openFile()    || die(errno); closeFile();
          /* если файл открылся, то die() не вычисляется */
        openFile() ? 0 : die(errno); closeFile();

        if(openFile()) closeFile();
        openFile()  && closeFile();
          /* вычислить closeFile() только если openFile() удалось */
        openFile()  && (printf("%s", fgets(buf, sizeof buf, fp)), closeFile());
}
```

В последней строке использован оператор "запятая": (a,b,c) возвращает значение выражения *c*.

1.59. Напишите функцию, вычисляющую сумму массива заданных чисел.

1.60. Напишите функцию, вычисляющую среднее значение массива заданных чисел.

1.61. Что будет напечатано в результате работы следующего цикла?

```
for ( i = 36; i > 0; i /= 2 )
      printf ( "%d%s", i,
                       i==1 ? ".\n":", ");
```

Ответ: 36, 18, 9, 4, 2, 1.

1.62. Найдите ошибки в следующей программе:

```
main {
     int i, j, k(10);

     for ( i = 0, i <= 10, i++   ){
           k[i] = 2 * i + 3;
           for ( j = 0, j <= i, j++ )
                 printf ("%i\n", k[j]);
     }
}
```

Обратите внимание на формат **%i**, существует ли такой формат? Есть ли это тот формат, по которому следует печатать значения типа **int**?

1.63. Напишите программу, которая распечатывает элементы массива. Напишите программу, которая распечатывает элементы массива по 5 чисел в строке.

1.64. Составьте программу считывания строк символов из стандартного ввода и печати номера введенной строки, адреса строки в памяти ЭВМ, значения строки, длины строки.

1.65. Стилистическое замечание: в операторе **return** возвращаемое выражение не обязательно должно быть в ()-скобках. Дело в том, что **return** - не функция, а *оператор*.

```
return  выражение;
return (выражение);
```

Однако если вы вызываете *функцию* (например, **exit**) - то аргументы должны быть в круглых скобках: **exit**(1); но не **exit** 1;

1.66. Избегайте ситуации, когда функция в разных ветвях вычисления то возвращает некоторое значение, то не возвращает ничего:

```
int func (int x) {
    if( x > 10  ) return x*2;
    if( x == 10 ) return (10);
    /* а здесь - неявный return; без значения */
}
```

при $x < 10$ функция вернет непредсказуемое значение! Многие компиляторы распознают такие ситуации и выдают предупреждение.

1.67. Напишите программу, запрашивающую ваше имя и "приветствующую" вас. Напишите функцию чтения строки. Используйте **getchar**() и **printf**().

Ответ:

```
#include <stdio.h>  /* standard input/output */
main(){
   char buffer[81]; int i;
   printf( "Введите ваше имя:" );
   while((i = getstr( buffer, sizeof buffer )) != EOF){
          printf( "Здравствуй, %s\n", buffer );
          printf( "Введите ваше имя:" );
   }
}

getstr( s, maxlen )
 char *s;    /* куда поместить строку */
 int maxlen; /* длина буфера:
                макс. длина строки = maxlen-1 */
```

```
{  int c;    /* не char! (почему ?) */
   register int i = 0;

   maxlen--; /* резервируем байт под конечный '\0' */
   while(i < maxlen && (c = getchar()) != '\n'
                    &&  c != EOF )
          s[i++] = c;
   /* обратите внимание, что сам символ '\n'
    * в строку не попадет */
   s[i] = '\0';  /* признак конца строки */
   return (i == 0 && c == EOF) ? EOF : i;
   /* вернем длину строки  */
}
```

Вот еще один вариант функции чтения строки: в нашем примере ее следует вызывать как
fgetstr(*buffer*,sizeof(*buffer*),**stdin**);
Это подправленный вариант стандартной функции **fgets** (в ней строки @1 и @2 обменяны местами).

```
char *fgetstr(char *s, int maxlen, register FILE *fp){
  register c; register char *cs = s;

  while(--maxlen > 0 && (c = getc(fp)) != EOF){
    if(c == '\n') break;  /* @1 */
    *cs++ = c;            /* @2 */
  }
  if(c == EOF && cs == s) return NULL;
  /* Заметьте, что при EOF строка s не меняется! */
  *cs = '\0'; return s;
}
```

Исследуйте поведение этих функций, когда входная строка слишком длинная (длиннее *maxlen*). Замечание: вместо нашей "рукописной" функции **getstr**() мы могли бы использовать стандартную библиотечную функцию **gets**(*buffer*).

1.68. Объясните, почему d стало отрицательным и почему %X печатает больше F, чем в исходном числе? Пример выполнялся на 32-х битной машине.

```
main(){
  unsigned short u = 65535;  /* 16 бит: 0xFFFF */
           short d = u;      /* 15 бит + знаковый бит */
  printf( "%X %d\n", d, d);  /* FFFFFFFF -1 */
}
```

Указание: рассмотрите двоичное представление чисел (смотри приложение). Какие приведения типов здесь происходят?

1.69. Почему 128 превратилось в отрицательное число?

```
main()
{
  /*signed*/ char c  = 128; /* биты: 10000000 */
  unsigned   char uc = 128;
  int  d = c;   /* используется 32-х битный int */
  printf( "%d %d %x\n", c, d, d );
  /*       -128 -128 ffffff80 */
       d = uc;
  printf( "%d %d %x\n", uc, d, d );
  /*       128 128 80       */
}
```

Ответ: при приведении **char** к **int** расширился знаковый бит (7-ой), заняв всю старшую часть слова. Знаковый бит int-а стал равен 1, что является признаком отрицательного числа. То же будет происходить со всеми значениями *c* из диапазона 128..255 (содержащими бит 0200). При приведении **unsigned char** к **int** знаковый бит не расширяется.

Можно было поступить еще и так:

```
printf( "%d\n", c & 0377 );
```

Здесь *c* приводится к типу **int** (потому что при использовании в аргументах функции тип **char** ВСЕГДА приводится к типу **int**), затем &0377 занулит старший байт полученного целого числа (состоящий из битов 1),

снова превратив число в положительное.

1.70. Почему

```
printf("%d\n", '\377' == 0377 );
printf("%d\n", '\xFF' == 0xFF );
```

печатает 0 (ложь)? Ответ: по той же причине, по которой

```
printf("%d %d\n", '\377', 0377);
```

печатает -1 255, а именно: char '\377' приводится в выражениях к целому расширением знакового бита (а 0377 - уже целое).

1.71. Рассмотрим программу

```
#include <stdio.h>
int main(int ac, char **av){
        int c;

        while((c = getchar()) != EOF)
                switch(c){
                case 'ы': printf("Буква ы\n"); break;
                case 'й': printf("Буква й\n"); break;
                default:  printf("Буква с кодом %d\n", c); break;
                }
        return 0;
}
```

Она работает так:

```
% a.out
йфыв
Буква с кодом 202
Буква с кодом 198
Буква с кодом 217
Буква с кодом 215
Буква с кодом 10
^D
%
```

Выполняется всегда **default**, почему не выполняются **case** *'ы'* и **case** *'й'*?

Ответ: русские буквы имеют восьмой бит (левый) равный 1. В **case** такой байт приводится к типу **int** расширением знакового бита. В итоге получается *отрицательное число*. Пример:

```
void main(void){
        int c = 'й';
        printf("%d\n", c);
}
печатает -54
```

Решением служит подавление расширения знакового бита:

```
#include <stdio.h>
/* Одно из двух */
#define U(c)    ((c) & 0xFF)
#define UC(c)   ((unsigned char) (c))
int main(int ac, char **av){
        int c;

        while((c = getchar()) != EOF)
                switch(c){
                case U('ы'):    printf("Буква ы\n"); break;
                case UC('й'):   printf("Буква й\n"); break;
                default:        printf("Буква с кодом %d\n", c); break;
                }
        return 0;
}
```

Она работает правильно:

```
% a.out
йфыв
Буква й
Буква с кодом 198
Буква ы
Буква с кодом 215
Буква с кодом 10
^D
%
```

Возможно также использование кодов букв:

```
case 0312:
```

но это гораздо менее наглядно. Подавление знакового бита необходимо также и в операторах **if**:

```
int c;
        ...
if(c == 'й') ...

        следует заменить на

if(c == UC('й')) ...
```

Слева здесь - **signed int**, правую часть компилятор тоже приводит к **signed int**. Приходится явно говорить, что справа - **unsigned**.

1.72. Рассмотрим программу, которая должна напечатать числа от 0 до 255. Для этих чисел в качестве счетчика достаточен один байт:

```
int main(int ac, char *av[]){
    unsigned char ch;

    for(ch=0; ch < 256; ch++)
        printf("%d\n", ch);
    return 0;
}
```

Однако эта программа зацикливается, поскольку в момент, когда *ch*==255, это значение меньше 256. Следующим шагом выполняется *ch*++, и *ch* становится равно 0, ибо для **char** вычисления ведутся по модулю 256 (2 в 8 степени). То есть в данном случае 255+1=0

Решений существует два: первое - превратить **unsigned char** в **int**. Второе - вставить явную проверку на последнее значение диапазона.

```
int main(int ac, char *av[]){
    unsigned char ch;

    for(ch=0; ; ch++){
        printf("%d\n", ch);
        if(ch == 255) break;
    }
    return 0;
}
```

1.73. Подумайте, почему для

```
unsigned a, b, c;
a < b + c    не эквивалентно   a - b < c
```

(первое - более корректно). Намек в виде примера (он выполнялся на 32-битной машине):

```
a = 1; b = 3; c = 2;
printf( "%u\n", a - b );     /* 4294967294, хотя в
                    нормальной арифметике 1 - 3 = -2 */
printf( "%d\n", a < b + c ); /* 1 */
printf( "%d\n", a - b < c ); /* 0 */
```

Могут ли **unsigned** числа быть отрицательными?

1.74. Дан текст:

```
short x = 40000;
printf("%d\n", x);
```

Печатается -25536. Объясните эффект. Указание: каково наибольшее представимое короткое целое (16 битное)? Что на самом деле оказалось в x? (лишние слева биты - обрубаются).

1.75. Почему в примере

```
double x = 5 / 2;
printf( "%g\n", x );
```

значение x равно 2 а не 2.5 ?

Ответ: производится целочисленное деление, затем в присваивании целое число 2 приводится к типу double. Чтобы получился ответ 2.5, надо писать одним из следующих способов:

```
double x = 5.0 / 2;
x = 5 / 2.0;
x = (double) 5 / 2;
x = 5 / (double) 2;
x = 5.0 / 2.0;
```

то есть в выражении должен быть хоть один операнд типа double.

Объясните, почему следующие три оператора выдают такие значения:

```
double g = 9.0;
int t = 3;
double dist = g *  t * t / 2;    /* 40.5 */
       dist = g * (t * t / 2);   /* 36.0 */
       dist = g * (t * t / 2.0); /* 40.5 */
```

В каких случаях деление целочисленное, в каких - вещественное? Почему?

1.76. Транслируйте пример на машине с длиной слова int равной 16 бит:

```
long n  = 1024 * 1024;
long nn =  512 *  512;
printf( "%ld %ld\n", n, nn );
```

Почему печатается 0 0 а не 1048576 262144?

Ответ: результат умножения (2**20 и 2**18) - это *целое* число; однако оно слишком велико для сохранения в 16 битах, поэтому старшие биты обрубаются. Получается 0. Затем в присваивании это уже обрубленное значение приводится к типу long (32 бита) - это все равно будет 0.

Чтобы получить корректный результат, надо чтобы выражение справа от = уже имело тип long и сразу сохранялось в 32 битах. Для этого оно должно иметь хоть один операнд типа long:

```
long n = (long) 1024 * 1024;
long nn =        512 *  512L;
```

1.77. Найдите ошибку в операторе:

```
x - = 4;   /* вычесть из x число 4 */
```

Ответ: между `-' и `=' не должно быть пробела. Операция вида

```
x @= expr;
```

означает

```
x = x @ expr;
```

(где @ - одна из операций **+ - * / % ^ >> << & |**), причем *x* здесь вычисляется единственный раз (т.е. такая форма не только короче и понятнее, но и экономичнее).

Однако имеется тонкое отличие a=a+n от a+=n; оно заключается в том, сколько раз вычисляется *a*. В случае a+=n единожды; в случае a=a+n два раза.

```
#include <stdio.h>

static int x = 0;
int *iaddr(char *msg){
        printf("iaddr(%s) for x=%d evaluated\n", msg, x);
        return &x;
}
int main(){
        static int a[4];
        int *p, i;

        printf( "1: "); x = 0; (*iaddr("a"))++;
        printf( "2: "); x = 0; *iaddr("b") += 1;
        printf( "3: "); x = 0; *iaddr("c") = *iaddr("d") + 1;

        for(i=0, p = a; i < sizeof(a)/sizeof(*a); i++) a[i] = 0;
        *p++ += 1;
        for(i=0; i < sizeof(a)/sizeof(*a); i++)
                printf("a[%d]=%d ", i, a[i]);
        printf("offset=%d\n", p - a);

        for(i=0, p = a; i < sizeof(a)/sizeof(*a); i++) a[i] = 0;
        *p++ = *p++ + 1;
        for(i=0; i < sizeof(a)/sizeof(*a); i++)
                printf("a[%d]=%d ", i, a[i]);
        printf("offset=%d\n", p - a);

        return 0;
}
```

Выдача:

```
1: iaddr(a) for x=0 evaluated
2: iaddr(b) for x=0 evaluated
3: iaddr(d) for x=0 evaluated
iaddr(c) for x=0 evaluated
a[0]=1 a[1]=0 a[2]=0 a[3]=0 offset=1
a[0]=1 a[1]=0 a[2]=0 a[3]=0 offset=2
```

Заметьте также, что

```
a[i++] += z;

        это
a[i] = a[i] + z; i++;

        а вовсе не
a[i++] = a[i++] + z;
```

1.78. Операция *y* = ++*x*; эквивалентна

```
y = (x = x+1, x);
```

а операция *y* = *x*++; эквивалентна

```
y = (tmp = x, x = x+1, tmp);
или
y = (x += 1) - 1;
```

где *tmp* - временная псевдопеременная того же типа, что и *x*. Операция `,' выдает значение последнего выражения из перечисленных (подробнее см. ниже).

Пусть *x*=1. Какие значения будут присвоены *x* и *y* после выполнения оператора

```
y = ++x + ++x + ++x;
```

1.79. Пусть *i*=4. Какие значения будут присвоены *x* и *i* после выполнения оператора

```
x = --i + --i + --i;
```

1.80. Пусть *x*=1. Какие значения будут присвоены *x* и *y* после выполнения оператора

```
y = x++ + x++ + x++;
```

1.81. Пусть *i*=4. Какие значения будут присвоены *i* и *y* после выполнения оператора

```
y = i-- + i-- + i--;
```

1.82. Корректны ли операторы

```
char *p = "Jabberwocky"; char s[] = "0123456789?";
int i = 0;

s[i] = p[i++]; или *p   = *++p;
                или s[i] = i++;
          или даже *p++ = f( *p );
```

Ответ: нет, стандарт не предусматривает, какая из частей присваивания вычисляется первой: левая или правая. Поэтому все может работать так, как мы и подразумевали, но может и иначе! Какое *i* используется в *s*[*i*]: 0 или уже 1 (++ уже сделан или нет), то есть

```
int i = 0; s[i] = i++;     это
s[0]  = 0;      или же     s[1] = 0;  ?
```

Какое *p* будет использовано в левой части **p*: уже продвинутое или старое? Еще более эта идея драматизирована в

```
s[i++] = p[i++];
```

Заметим еще, что в

```
int i=0, j=0;
s[i++] = p[j++];
```

такой проблемы не возникает, поскольку индексы обоих в частях присваивания независимы. Зато аналогичная проблема встает в

```
if( a[i++] < b[i] )...;
```

Порядок вычисления операндов не определен, поэтому неясно, что будет сделано прежде: взято значение *b*[*i*] или значение *a*[*i*++] (тогда будет взято *b*[*i*+1]). Надо писать так, чтобы не полагаться на особенности вашего компилятора:

```
if( a[i] < b[i+1] )...;  или  *p = *(p+1);
i++;                           ++p;
```

Твердо усвойте, что *i*++ и ++*i* не только выдают значения *i* и *i*+1 соответственно, но и изменяют значение *i*. Поэтому эти операторы НЕ НАДО использовать там, где по смыслу требуется *i*+1, а не *i*=*i*+1. Так для сравнения соседних элементов массива

```
if( a[i] < a[i+1] ) ... ;  /* верно */
if( a[i] < a[++i] ) ... ;  /* неверно */
```

1.83. Порядок вычисления операндов в бинарных выражениях не определен (что раньше вычисляется - левый операнд или же правый ?). Так пример

```
int f(x,s) int x; char *s;
{  printf( "%s:%d ", s, x ); return x; }

main(){
   int x = 1;
   int y = f(x++, "f1") + f(x+=2, "f2");
   printf("%d\n", y);
}
```

может печатать либо

```
f1:1 f2:4 5
   либо
f2:3 f1:3 6
```

в зависимости от особенностей поведения вашего компилятора (какая из двух **f**() выполнится первой: левая или правая?). Еще пример:

```
int y = 2;
int x = ((y = 4) * y );
printf( "%d\n", x );
```

Может быть напечатано либо 16, либо 8 в зависимости от поведения компилятора, т.е. данный оператор немобилен. Следует написать

```
y = 4;  x = y * y;
```

1.84. Законен ли оператор

```
f(x++, x++);   или  f(x, x++);
```

Ответ: Нет, порядок вычисления аргументов функций не определен. По той же причине мы не можем писать

```
f( c = getchar(), c );
```

а должны писать

```
c = getchar();  f(c, c);
```

(если мы именно это имели в виду). Вот еще пример:

```
...
case '+':
        push(pop()+pop()); break;
case '-':
        push(pop()-pop()); break;
...
```

следует заменить на

```
...
case '+':
        push(pop()+pop()); break;
case '-':
        { int x = pop(); int y = pop();
          push(y - x); break;
        }
...
```

И еще пример:

```
int x = 0;
printf( "%d %d\n", x = 2, x );   /* 2 0 либо 2 2 */
```

Нельзя также

```
struct pnt{ int x; int y; }arr[20]; int i=0;
...
scanf( "%d%d", & arr[i].x, & arr[i++].y );
```

поскольку *i*++ может сделаться раньше, чем чтение в *x*. Еще пример:

```
main(){
   int i = 3;
   printf( "%d %d %d\n", i += 7, i++, i++ );
}
```

который показывает, что на **IBM PC** † и **PDP-11** £ аргументы функций вычисляются справа налево (пример печатает 12 4 3). Впрочем, другие компиляторы могут вычислять их слева направо (как и подсказывает нам здравый смысл).

1.85. Программа печатает либо x=1 либо x=0 в зависимости от КОМПИЛЯТОРА - вычисляется ли раньше правая или левая часть оператора вычитания:

---

† **IBM** ("Ай-би-эм") - International Buisiness Machines Corporation. Персональные компьютеры **IBM PC** построены на базе микропроцессоров фирмы **Intel**.

£ **PDP-11** - (Programmed Data Processor) - компьютер фирмы **DEC** (**Digital Equipment Corporation**), у нас известный как **СМ-1420**. Эта же фирма выпускает машину **VAX**.

```
#include <stdio.h>
void main(){
        int c = 1;
        int x = c - c++;

        printf( "x=%d c=%d\n", x, c );
        exit(0);
}
```

Что вы имели в виду ?

```
left = c; right = c++; x = left - right;
        или
right = c++; left = c; x = left - right;
```

А если компилятор еще и распараллелит вычисление left и right - то одна программа в разные моменты времени сможет давать разные результаты.

Вот еще достойная задачка:

```
x = c-- - --c;  /* c-----c */
```

1.86. Напишите программу, которая устанавливает в 1 бит 3 и сбрасывает в 0 бит 6. Биты в слове нумеруются с нуля справа налево. Ответ:

```
        int x = 0xF0;

        x |=  (1 << 3);
        x &= ~(1 << 6);
```

В программах часто используют битовые маски как флаги некоторых параметров (признак - есть или нет). Например:

```
#define A  0x08             /* вход свободен  */
#define B  0x40             /* выход свободен */
установка флагов            : x |=    A|B;
сброс флагов                : x &=  ~(A|B);
проверка флага A            : if( x & A ) ...;
проверка, что оба флага есть: if((x & (A|B)) == (A|B))...;
проверка, что обоих нет     : if((x & (A|B)) == 0 )...;
проверка, что есть хоть один: if( x & (A|B))...;
проверка, что есть только A : if((x & (A|B)) == A)...;
проверка, в каких флагах
  различаются x и y         : diff = x ^ y;
```

1.87. В программах иногда требуется использовать "**множество**": каждый допустимый элемент множества имеет *номер* и может либо присутствовать в множестве, либо отсутствовать. Число вхождений не учитывается. Множества принято моделировать при помощи битовых шкал:

```
#define   SET(n,a) (a[(n)/BITS] |=  (1L <<((n)%BITS)))
#define   CLR(n,a) (a[(n)/BITS] &= ~(1L <<((n)%BITS)))
#define ISSET(n,a) (a[(n)/BITS] &   (1L <<((n)%BITS)))
#define BITS 8 /* bits per char (битов в байте) */
/* Перечислимый тип */
enum fruit { APPLE, PEAR, ORANGE=113,
             GRAPES, RAPE=125, CHERRY};
/* шкала:   n из интервала 0..(25*BITS)-1 */
static char fr[25];
main(){
  SET(GRAPES, fr);  /* добавить в множество */
  if(ISSET(GRAPES, fr)) printf("here\n");
  CLR(GRAPES, fr);  /* удалить из множества */
}
```

1.88. Напишите программу, распечатывающую все возможные перестановки массива из *N* элементов. Алгоритм будет рекурсивным, например таким: в качестве первого элемента перестановки взять *i*-ый элемент массива. Из оставшихся элементов массива (если такие есть) составить все перестановки порядка *N*-1. Выдать все перестановки порядка *N*, получающиеся склейкой *i*-ого элемента и всех (по очереди) перестановок порядка *N*-1. Взять следующее *i* и все повторить.

Главная проблема здесь - организовать *оставшиеся* после извлечения *i*-ого элемента элементы массива в удобную структуру данных (чтобы постоянно не копировать массив). Можно использовать, например, битовую шкалу уже выбранных элементов. Воспользуемся для этого макросами из предыдущего параграфа:

```
/* ГЕНЕРАТОР ПЕРЕСТАНОВОК ИЗ n ЭЛЕМЕНТОВ ПО m */

extern void *calloc(unsigned nelem, unsigned elsize);
/* Динамический выделитель памяти, зачищенной нулями.
 * Это стандартная библиотечная функция.
 * Обратная к ней - free();
 */
extern void free(char *ptr);
static int N, M, number;
static char *scale;     /* шкала выбранных элементов */
       int  *res;       /* результат */

/* ... текст определений SET, CLR, ISSET, BITS ... */

static void choose(int ind){
  if(ind == M){ /* распечатать перестановку */
     register i;
     printf("Расстановка #%04d", ++number);
     for(i=0; i < M; i++) printf(" %2d", res[i]);
     putchar('\n'); return;
  } else
  /* Выбрать очередной ind-тый элемент перестановки
   * из числа еще не выбранных элементов.
   */
  for(res[ind] = 0; res[ind] < N; ++res[ind])
     if( !ISSET(res[ind], scale)) {
        /* элемент еще не был выбран */
        SET(res[ind], scale);      /* выбрать */
        choose(ind+1);
        CLR(res[ind], scale);      /* освободить */
     }
}
void arrange(int n, int m){
  res   = (int *)  calloc(m, sizeof(int));
  scale = (char *) calloc((n+BITS-1)/BITS, 1);
  M = m; N = n; number = 0;
  if( N >= M ) choose(0);
  free((char *) res); free((char *) scale);
}
void main(int ac, char **av){
   if(ac != 3){ printf("Arg count\n"); exit(1); }
   arrange(atoi(av[1]), atoi(av[2]));
}
```

Программа должна выдать *n*!/(*n*-*m*)! расстановок, где *x*! = 1*2*...**x* - функция "факториал". По определению 0! = 1. Попробуйте переделать эту программу так, чтобы очередная перестановка печаталась по запросу:

```
res = init_iterator(n, m);
/* печатать варианты, пока они есть */
while( next_arrangement (res))
       print_arrangement(res, m);
clean_iterator(res);
```

1.89. Напишите макроопределения циклического сдвига переменной типа **unsigned int** на *skew* бит влево и вправо (ROL и ROR). Ответ:

```
#define BITS 16     /* пусть целое состоит из 16 бит */
#define ROL(x,skew) x=(x<<(skew))|(x>>(BITS-(skew)))
#define ROR(x,skew) x=(x>>(skew))|(x<<(BITS-(skew)))
```

```
Вот как работает ROL(x, 2) при BITS=6
        |abcdef|        исходно
       abcdef00          << 2
         0000abcdef      >> 4
         ------          операция |
         cdefab          результат
```

В случае **signed int** потребуется накладывать маску при сдвиге вправо из-за того, что левые биты при >> не заполняются нулями. Приведем пример для сдвига переменной типа **signed char** (по умолчанию все **char** - знаковые) на 1 бит влево:

```
#define CHARBITS 8
#define ROLCHAR1(x) x=(x<<1)|((x>>(CHARBITS-1)) & 01)
     соответственно для сдвига
на 2 бита надо делать &  03
на 3                  &  07
на 4                  & 017
на skew               & ~(~0 << skew)
```

1.90. Напишите программу, которая инвертирует (т.е. заменяет 1 на 0 и наоборот) N битов, начинающихся с позиции P, оставляя другие биты без изменения. Возможный ответ:

```
unsigned x, mask;
mask = ~(~0 << N) << P;
x = (x & ~mask) | (~x & mask);
             /*   xnew   */
```

Где маска получается так:

```
  ~0            = 11111....11111
  ~0 << N       = 11111....11000  /* N нулей */
~(~0 << N)      = 00000....00111  /* N единиц */
~(~0 << N) << P = 0...01110...00
/* N единиц на местах P+N-1..P */
```

1.91. Операции умножения * и деления / и % обычно достаточно медленны. В критичных по скорости функциях можно предпринять некоторые ручные оптимизации, связанные с представлением чисел в двоичном коде (хороший компилятор делает это сам!) - пользуясь тем, что операции +, &, >> и << гораздо быстрее. Пусть у нас есть

```
unsigned int x;
```

(для **signed** операция >> может не заполнять освобождающиеся левые биты нулем!) и **2****n означает 2 в степени *n*. Тогда:

```
x * (2**n)  = x << n
x / (2**n)  = x >> n
x % (2**n)  = x - ((x >> n) << n)
x % (2**n)  = x & (2**n - 1)
              это  11...111  n двоичных единиц
```

Например:

```
x * 8   = x << 3;
x / 8   = x >> 3; /* деление нацело     */
x % 8   = x &  7; /* остаток от деления */
x * 80  = x*64 + x*16  = (x << 6) + (x << 4);
x * 320 = (x * 80) * 4 = (x * 80) << 2 =
                         (x << 8) + (x << 6);
x * 21  = (x << 4) + (x << 2) + x;

x & 1   = x % 2 = четное(x)? 0:1 = нечетное(x)? 1:0;

x & (-2) = x & 0xFFFE = | если x = 2*k      то 2*k
                        | если x = 2*k + 1  то 2*k
                        | то есть округляет до четного
```

Или формула для вычисления количества дней в году (високосный/простой):

```
days_in_year = (year % 4 == 0) ? 366 : 365;
        заменяем на
days_in_year = ((year & 0x03) == 0) ? 366 : 365;
```

Вот еще одно полезное равенство:

```
x = x & (a|~a) = (x & a) | (x & ~a) = (x&a) + (x&~a)
        из чего вытекает, например
x - (x % 2**n) = x - (x &  (2**n - 1)) =
               =      x & ~(2**n - 1)  = (x>>n) << n
x - (x%8) = x-(x&7) = x & ~7
```

Последняя строка может быть использована в функции **untab**() в главе "Текстовая обработка".

1.92. Обычно мы вычисляем min(a,b) так:

```
#define min(a, b) (((a) < (b)) ? (a) : (b))
```

или более развернуто

```
if(a < b) min = a;
else      min = b;
```

Здесь есть операция сравнения и условный переход. Однако, если (a < b) эквивалентно условию (a - b) < 0, то мы можем избежать сравнения. Это предположение верно при

```
(unsigned int)(a - b) <= 0x7fffffff.
```

что, например, верно если a и b - оба неотрицательные числа между 0 и 0x7fffffff. При этих условиях

```
min(a, b) = b + ((a - b) & ((a - b) >> 31));
```

Как это работает? Рассмотрим два случая:

```
Случай 1: a < b

        Здесь (a - b) < 0, поэтому старший (левый, знаковый) бит
        разности (a - b) равен 1.
        Следовательно, (a - b) >> 31 == 0xffffffff,
        и мы имеем:

        min(a, b)       = b + ((a - b) & ((a - b) >> 31))
                        = b + ((a - b) & (0xffffffff))
                        = b + (a - b)
                        = a
        что корректно.

Случай 2: a >= b

        Здесь (a - b) >= 0, поэтому старший бит разности
        (a - b) равен 0.  Тогда (a - b) >> 31 == 0, и мы имеем:

        min(a, b)       = b + ((a - b) & ((a - b) >> 31))
                        = b + ((a - b) & (0x00000000))
                        = b + (0)
                        = b

        что также корректно.
```

Статья предоставлена by Jeff Bonwick.

1.93. Есть ли быстрый способ определить, является ли X степенью двойки? Да, есть.

```
int X является степенью двойки
тогда и только тогда, когда

        (X & (X - 1)) == 0
```

(в частности 2 здесь окажется степенью двойки). Как это работает? Пусть X != 0. Если X - целое, то его двоичное представление таково:

```
X = bbbbbbbbbb10000...
```

где 'bbb' представляет некие биты, '1' - младший бит, и все остальные биты правее - нули. Поэтому:

```
X               = bbbbbbbbbb10000...
X - 1           = bbbbbbbbbb01111...
-----------------------------------
X & (X - 1)     = bbbbbbbbbb00000...
```

Другими словами, X & (X-1) имеет эффект обнуления последнего единичного бита. Если X - степень двойки, то он содержит в двоичном представлении ровно ОДИН такой бит, поэтому его гашение обращает результат в ноль. Если X - не степень двойки, то в слове есть хотя бы ДВА единичных бита, поэтому X & (X-1) должно содержать хотя бы один из оставшихся единичных битов - то есть не равняться нулю.

Следствием этого служит программа, вычисляющая число единичных битов в слове X:

```
int popc;
for (popc = 0; X != 0; X &= X - 1)
        popc++;
```

При этом потребуется не 32 итерации (число бит в int), а ровно столько, сколько единичных битов есть в X. Статья предоставлена by Jeff Bonwick.

1.94. Функция для поиска номера позиции старшего единичного бита в слове. Используется бинарный поиск: позиция находится максимум за 5 итераций (двоичный логарифм 32x), вместо 32 при линейном поиске.

```
int highbit (unsigned int x)
{
        int i;
        int h = 0;

        for (i = 16; i >= 1; i >>= 1) {
                if (x >> i) {
                        h += i;
                        x >>= i;
                }
        }
        return (h);
}
```

Статья предоставлена by Jeff Bonwick.

1.95. Напишите функцию, округляющую свой аргумент вниз до степени двойки.

```
#include <stdio.h>
#define INT short
#define INFINITY (-999)

/* Функция, выдающая число, являющееся округлением вниз
 * до степени двойки.
 * Например:
 *      0000100010111000110
 *      заменяется на
 *      0000100000000000000
 * то есть остается только старший бит.
 * В параметр power2 возвращается номер бита,
 * то есть показатель степени двойки. Если число == 0,
 * то эта степень равна минус бесконечности.
 */
```

```
unsigned INT round2(unsigned INT x, int *power2){
/* unsigned - чтобы число рассматривалось как
 * битовая шкала, а сдвиг >> заполнял левые биты
 * нулем, а не расширял вправо знаковый бит.
 * Идея функции: сдвигать число >> пока не получится 1
 * (можно было бы выбрать 0).
 * Затем сдвинуть << на столько же разрядов, при этом все правые
 * разряды заполнятся нулем, что и требовалось.
 */
        int n = 0;
        if(x == 0){
                *power2 = -INFINITY; return 0;
        }
        if(x == 1){
                *power2 = 0; return 1;
        }
        while(x != 1){
            x >>= 1;
            n++;
            if(x == 0 || x == (unsigned INT)(-1)){
                printf("Вижу %x: похоже, что >> расширяет знаковый бит.\n"
                       "Зациклились!!!\n", x);
                return (-1);
            }
        }
        x <<= n;

        *power2 = n; return x;
}

int counter[ sizeof(unsigned INT) * 8];
int main(void){
        unsigned INT i;
        int n2;
        for(i=0; ; i++){
            round2(i, &n2);
            if(n2 == -INFINITY) continue;
            counter[n2]++;

            /* Нельзя писать for(i=0; i < (unsigned INT)(-1); i++)
             * потому что такой цикл бесконечен!
             */
            if(i == (unsigned INT) (-1)) break;
        }
        for(i=0; i < sizeof counter/sizeof counter[0]; i++)
                printf("counter[%u]=%d\n", i, counter[i]);
        return 0;
}
```

1.96. Если некоторая вычислительная функция будет вызываться много раз, не следует пренебрегать возможностью построить таблицу решений, где значение вычисляется один раз для каждого входного значения, зато потом берется непосредственно из таблицы и не вычисляется вообще. Пример: подсчет числа единичных бит в байте. Напоминаю: байт состоит из 8 бит.

```
#include <stdio.h>

int nbits_table[256];

int countBits(unsigned char c){
    int nbits = 0;
    int bit;

    for(bit = 0; bit < 8; bit++){
            if(c & (1 << bit))
                    nbits++;
    }
    return nbits;
}
void generateTable(){
    int c;
    for(c=0; c < 256; c++){
        nbits_table[ (unsigned char) c ] = countBits(c);
        /* printf("%u=%d\n", c, nbits_table[ c & 0377 ]); */
    }
}

int main(void){
    int c;
    unsigned long bits  = 0L;
    unsigned long bytes = 0L;

    generateTable();

    while((c = getchar()) != EOF){
            bytes++;
            bits += nbits_table[ (unsigned char) c ];
    }
    printf("%lu байт\n", bytes);
    printf("%lu единичных бит\n", bits);
    printf("%lu нулевых бит\n", bytes*8 - bits);
    return 0;
}
```

1.97. Напишите макрос **swap**(x, y), обменивающий значениями два своих аргумента типа int.

```
#define swap(x,y) {int tmp=(x);(x)=(y);(y)=tmp;}
 ... swap(A, B); ...
```

Как можно обойтись без временной переменной? Ввиду некоторой курьезности последнего способа, приводим ответ:

```
        int x,  y;      /*  A    B   */

        x = x ^ y;       /*  A^B  B   */
        y = x ^ y;       /*  A^B  A   */
        x = x ^ y;       /*    B  A   */
```

Здесь используется тот факт, что A^A дает 0.

1.98. Напишите *функцию* **swap**(x, y) при помощи указателей. Заметьте, что в отличие от макроса ее придется вызывать как

```
... swap(&A, &B); ...
```

Почему?

1.99. Пример объясняет разницу между формальным и фактическим параметром. Термин "*формальный*" означает, что имя параметра можно произвольно заменить другим (во всем теле функции), т.е. *само имя* не существенно. Так

```
f(x,y)       { return(x   +    y); }     и
f(муж,жена)  { return(муж + жена); }
```

воплощают одну и ту же функцию. "*Фактический*" - означает значение, даваемое параметру в момент

вызова функции:

```
f(xyz, 43+1);
```

В Си это означает, что формальным параметрам (в качестве локальных переменных) присваиваются *начальные значения*, равные значениям фактических параметров:

```
x = xyz; y = 43 + 1; /*в теле ф-ции их можно менять*/
```

При выходе из функции формальные параметры (и локальные переменные) разопределяются (и даже уничтожаются, см. следующий параграф). Имена формальных параметров могут "перекрывать" (делать невидимыми, *override*) одноименные глобальные переменные на время выполнения данной функции.

Что печатает программа?

```
char str[] = "строка1";
char lin[] = "строка2";
f(str) char str[];      /* формальный параметр. */
{       printf( "%s %s\n", str, str );         }

main(){
        char *s = lin;
                        /* фактический параметр: */
        f(str);         /* массив str            */
        f(lin);         /* массив lin            */
        f(s);           /* переменная s          */
        f("строка3");   /* константа             */
        f(s+2);         /* значение выражения    */
}
```

Обратите внимание, что параметр *str* из **f**(*str*) и массив *str*[] - это две совершенно РАЗНЫЕ вещи, хотя и называющиеся одинаково. Переименуйте аргумент функции **f** и перепишите ее в виде

```
f(ss) char ss[];      /* формальный параметр. */
{       printf( "%s %s\n", ss, str );         }
```

Что печатается теперь? Составьте аналогичный пример с целыми числами.

1.100. Поговорим более подробно про область видимости имен.

```
int x = 12;
f(x){  int y = x*x;
       if(x) f(x - 1);
}
main(){ int x=173, z=21; f(2); }
```

Локальные переменные и аргументы функции отводятся в *стеке* при вызове функции и уничтожаются при выходе из нее:

```
       -+               +- вершина стека
        |локал    y=0   |
        |аргумент x=0   |  f(0)
        |---------------|---------
"кадр"  |локал    y=1   |
 frame  |аргумент x=1   |  f(1)
        |---------------|---------
        |локал    y=4   |
        |аргумент x=2   |  f(2)
        |---------------|---------
        |локал    z=21  |
auto:   |локал    x=173 |  main()
================================== дно стека
static:  глобал   x=12
==================================
```

Автоматические *локальные переменные* и *аргументы* функции видимы только в том вызове функции, в котором они отведены; но не видимы ни в вызывающих, ни в вызываемых функциях (т.е. видимость их ограничена рамками своего "кадра" стека). Статические *глобальные переменные* видимы в любом кадре, если только они не "перекрыты" (заслонены) одноименной локальной переменной (или формалом) в данном кадре.

Что напечатает программа? Постарайтесь ответить на этот вопрос не выполняя программу на машине!

```
                                          x1 x2 x3 x4 x5
int x = 12;                     /* x1 */ |  .  .  .  .
f(){                                     |___  .  .  .
  int x = 8;       /* x2, перекрытие */ :  |  .  .  .
  printf( "f: x=%d\n", x );    /* x2 */ :  |  .  .  .
  x++;                          /* x2 */ :  |  .  .  .
}                                        :--+  .  .  .
g(x){                           /* x3 */ :_____  .  .
  printf( "g: x=%d\n", x );    /* x3 */ :     |  .  .
  x++;                          /* x3 */ :     |  .  .
}                                        :-----+  .  .
h(){                                     :________  .
  int x = 4;                    /* x4 */ :        |  .
  g(x);                         /* x4 */ :        |___
  { int x = 55; }               /* x5 */ :        :  |
  printf( "h: x=%d\n", x );    /* x4 */ :        |--+
}                                        :--------+
main(){                                  |
  f(); h();                              |
  printf( "main: x=%d\n", x ); /* x1 */ |
}                                         ----
```

Ответ:

```
f: x=8
g: x=4
h: x=4
main: x=12
```

Обратите внимание на функцию **g**. Аргументы функции служат *копиями* фактических параметров (т.е. являются локальными переменными функции, проинициализированными значениями фактических параметров), поэтому их изменение не приводит к изменению фактического параметра. Чтобы изменять фактический параметр, надо передавать его адрес!

1.101. Поясним последнюю фразу. (Внимание! Возможно, что данный пункт вам следует читать ПОСЛЕ главы про указатели). Пусть мы хотим написать функцию, которая обменивает свои аргументы *x* и *y* так, чтобы выполнялось *x* < *y*. В качестве значения функция будет выдавать (*x*+*y*)/2. Если мы напишем так:

```
int msort(x, y) int x, y;
{ int tmp;
  if(x > y){ tmp=x; x=y; y=tmp; }
  return (x+y)/2;
}
int x=20, y=8;
main(){
  msort(x,y); printf("%d %d\n", x, y); /* 20 8 */
}
```

то мы не достигнем желаемого эффекта. Здесь переставляются *x* и *y*, которые являются локальными переменными, т.е. *копиями* фактических параметров. Поэтому *вне* функции эта перестановка никак не проявляется!

Чтобы мы могли изменить аргументы, копироваться в локальные переменные должны не сами *значения* аргументов, а их *адреса*:

```
int msort(xptr, yptr) int *xptr, *yptr;
{ int tmp;
  if(*xptr > *yptr){tmp= *xptr;*xptr= *yptr;*yptr=tmp;}
  return (*xptr + *yptr)/2;
}
int x=20, y=8, z;
main(){
  z = msort(&x,&y);
  printf("%d %d %d\n", x, y, z); /* 8 20 14 */
}
```

Обратите внимание, что теперь мы передаем в функцию не значения *x* и *y*, а их адреса &*x* и &*y*.

Именно поэтому (чтобы *x* смог измениться) стандартная функция **scanf**() требует указания адресов:

```
int x; scanf("%d", &x); /* но не scanf("%d", x); */
```

Заметим, что адрес от арифметического выражения или от константы (а не от переменной) вычислить нельзя, поэтому законны:

```
int xx=12, *xxptr = &xx, a[2] = { 13, 17 };
int *fy(){ return &y; }
        msort(&x, &a[0]);       msort(a+1, xxptr);
        msort(fy(), xxptr);
```

но незаконны

```
        msort(&(x+1), &y);   и  msort(&x, &17);
```

Заметим еще, что при работе с адресами мы можем направить указатель в неверное место и получить непредсказуемые результаты:

```
        msort(&xx - 20, a+40);
```

(указатели указывают неизвестно на что).

Резюме: если аргумент служит только для передачи значения *В* функцию - его не надо (хотя и можно) делать указателем на переменную, содержащую требуемое значение (если только это уже не указатель). Если же аргумент служит для передачи значения *ИЗ* функции - он **должен** быть указателем на переменную возвращаемого типа (лучше возвращать значение как значение функции - **return**-ом, но иногда надо возвращать несколько значений - и этого главного "окошка" не хватает).

Контрольный вопрос: что печатает фрагмент?

```
int a=2, b=13, c;
int f(x, y, z) int x, *y, z;
{
        *y += x; x *= *y; z--;
        return (x + z - a);
}
main(){ c=f(a, &b, a+4); printf("%d %d %d\n",a,b,c); }
```

(Ответ: 2 15 33)

1.102. Формальные аргументы функции - это такие же локальные переменные. Параметры как бы описаны в самом внешнем блоке функции:

```
char *func1(char *s){
        int s;        /* ошибка: повторное определение имени s */
        ...
}
int func2(int x, int y){
        int z;
        ...
}
соответствует
int func2(){
        int x = безымянный_аргумент_1_со_стека;
        int y = безымянный_аргумент_2_со_стека;
        int z;
        ...
}
```

Мораль такова: формальные аргументы можно смело изменять и использовать как локальные переменные.

1.103. Все параметры функции можно разбить на 3 класса:

- **in** - входные;
- **out** - выходные, служащие для возврата значения из функции; либо для изменения данных, находящихся по этому адресу;
- **in/out** - для передачи значения в функцию и из функции.

Два последних типа параметров должны быть указателями. Иногда (особенно в прототипах и в документации) бывает полезно указывать класс параметра в виде комментария:

```
int f( /*IN*/    int  x,
       /*OUT*/   int *yp,
       /*INOUT*/ int *zp){
   *yp = ++x + ++(*zp);
   return (*zp *= x) - 1;
}
int x=2, y=3, z=4, res;
main(){  res = f(x,  &y,  &z);
 printf("res=%d  x=%d y=%d z=%d\n",res,x,y,z);
         /*  14     2    8   15   */
}
```

Это полезно потому, что иногда трудно понять - зачем параметр описан как указатель. То ли по нему *выдается* из функции информация, то ли это просто указатель на данные (массив), *передаваемые в* функцию. В первом случае указуемые данные будут изменены, а во втором - нет. В первом случае указатель должен указывать на зарезервированную *нами* область памяти, в которой будет размещен результат. Пример на эту тему есть в главе "Текстовая обработка" (функция **bi_conv**).

1.104. Известен такой стиль оформления аргументов функции:

```
void func(   int    arg1
         ,   char  *arg2      /* argument 2 */
         ,   char  *arg3[]
         ,   time_t time_stamp
         ){ ... }
```

Суть его в том, что запятые пишутся в столбик и в одну линию с (и) скобками для аргументов. При таком стиле легче добавлять и удалять аргументы, чем при версии с запятой в конце. Этот же стиль применим, например, к перечислимым типам:

```
enum {  red
     ,  green
     ,  blue
     };
```

Напишите программу, форматирующую заголовки функций таким образом.

1.105. В чем ошибка?

```
char *val(int x){
     char str[20];
     sprintf(str, "%d", x);
     return str;
}
void main(){
     int x = 5; char *s = val(x);
     printf("The values:\n");
     printf("%d %s\n", x, s);
}
```

Ответ: **val** возвращает указатель на *автоматическую* переменную. При выходе из функции **val**() ее локальные переменные (в частности *str*[]) в стеке *уничтожаются* - указатель *s* теперь указывает на испорченные данные! Возможным решением проблемы является превращение *str*[] в *статическую* переменную (хранимую не в стеке):

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

Однако такой способ не позволит писать конструкции вида

```
printf("%s %s\n", val(1), val(2));
```

так как под оба вызова **val**() используется *один и тот же* буфер *str*[] и будет печататься "1 1" либо "2 2", но не "1 2". Более правильным будет задание буфера для результата **val**() как аргумента:

```
char *val(int x, char str[]){
      sprintf(str, "%d", x);
      return str;
}
void main(){
     int x=5, y=7;
     char s1[20], s2[20];
     printf("%s %s\n", val(x, s1), val(y, s2));
```

```
}
```

1.106. Каковы ошибки (не синтаксические) в программе†?

```
main() {
   double y; int x = 12;
   y = sin (x);
   printf ("%s\n", y);
}
```

Ответ:

- стандартная библиотечная функция **sin**() возвращает значение типа **double**, но мы нигде не информируем об этом компилятор. Поэтому он считает по умолчанию, что эта функция возвращает значение типа **int** и делает в присваивании *y*=**sin**(*x*) приведение типа **int** к типу левого операнда, т.е. к **double**. В результате возвращаемое значение (а оно на самом деле - **double**) интерпретируется неверно (как **int**), подвергается приведению типа (которое портит его), и результат получается совершенно не таким, как надо. Подобная же ошибка возникает при использовании функций, возвращающих указатель, например, функций **malloc**() и **itoa**(). Поэтому если мы пользуемся библиотечной функцией, *возвращающей не int*, мы должны предварительно (до первого использования) описать ее, например£:

  ```
  extern double sin();
  extern long   atol();
  extern char  *malloc(), *itoa();
  ```

  Это же относится и к нашим собственным функциям, которые мы используем прежде, чем определяем (поскольку из заголовка функции компилятор обнаружит, что она выдает не целое значение, уже *после* того, как транслирует обращение к ней):

  ```
/*extern*/ char *f();
           main(){
                char *s;
                s = f(1); puts(s);
           }
           char *f(n){ return "knights" + n; }
  ```

  Функции, возвращающие *целое*, описывать не требуется. Описания для некоторых стандартных функций уже помещены в системные include-файлы. Например, описания для математических функций (**sin**, **cos**, **fabs**, ...) содержатся в файле */usr/include/math.h*. Поэтому мы могли бы написать перед **main**

  ```
           #include <math.h>
  ```
  вместо
  ```
           extern double sin(), cos(), fabs();
  ```

- библиотечная функция **sin**() требует аргумента типа **double**, мы же передаем ей аргумент типа **int** (который короче типа **double** и имеет иное внутреннее представление). Он будет неправильно проинтерпретирован функцией, т.е. мы вычислим синус отнюдь НЕ числа 12. Следует писать:

  ```
     y = sin( (double) x );
  и  sin(12.0); вместо sin(12);
  ```

- в **printf** мы печатаем значение типа **double** по неправильному формату: следует использовать формат **%g** или **%f** (а для ввода при помощи **scanf**() - **%lf**). Очень частой ошибкой является печать значений типа **long** по формату **%d** вместо **%ld** .

Первых двух проблем в современном Си удается избежать благодаря заданию прототипов функций (о них подробно рассказано ниже, в конце главы "Текстовая обработка"). Например, **sin** имеет прототип

```
double sin(double x);
```

Третяя проблема (ошибка в формате) не может быть локализована средствами Си и имеет более-менее приемлемое решение лишь в языке C++ (*streams*).

---

† Для трансляции программы, использующей стандартные математические функции **sin**, **cos**, **exp**, **log**, **sqrt**, и.т.п. следует задавать ключ компилятора **-lm**

cc *file.c* **-o** *file* **-l***m*

£ Слово **extern** ("внешняя") не является обязательным, но является признаком хорошего тона - вы сообщаете программисту, читающему эту программу, что данная функция реализована в другом файле, либо вообще является стандартной и берется из библиотеки.

1.107. Найдите ошибку:

```
int sum(x,y,z){  return(x+y+z); }
main(){
        int s = sum(12,15);
        printf("%d\n", s);
}
```

Заметим, что если бы для функции **sum**() был задан прототип, то компилятор поймал бы эту нашу оплошность! Заметьте, что сейчас значение *z* в **sum**() непредсказуемо. Если бы мы вызывали

```
s = sum(12,15,17,24);
```

то лишние аргументы были бы просто проигнорированы (но и тут может быть сюрприз - аргументы могли бы игнорироваться с ЛЕВОГО конца списка!).

А вот пример опасной ошибки, которая не ловится даже прототипами:

```
int x;   scanf("%d%d", &x );
```

Второе число по формату **%d** будет считано неизвестно по какому адресу и разрушит память программы. Ни один компилятор не проверяет соответствие числа **%**-ов в строке формата числу аргументов **scanf** и **printf**.

1.108. Что здесь означают внутренние („) в вызове функции f() ?

```
f(x, y, z){
        printf("%d %d %d\n", x, y, z);
}
main(){ int t;
        f(1, (2, 3, 4), 5);
        f(1, (t=3,t+1), 5);
}
```

Ответ: (2,3,4) - это оператор "запятая", выдающий значение *последнего выражения* из списка перечисленных через запятую выражений. Здесь будет напечатано 1 4 5. Кажущаяся двойственность возникает из-за того, что *аргументы функции* тоже перечисляются через запятую, но это совсем другая синтаксическая конструкция. Вот еще пример:

```
int y = 2, x;
x = (y+4, y, y*2); printf("%d\n", x);       /*  4 */
x =  y+4, y, y*2 ; printf("%d\n", x);       /*  6 */
x = (x=y+4, ++y, x*y); printf("%d\n", x);   /* 18 */
```

Сначала обратим внимание на первую строку. Это - объявление переменных *x* и *y* (причем *y* - с инициализацией), поэтому запятая здесь - не ОПЕРАТОР, а просто разделитель объявляемых переменных! Далее следуют три строки выполняемых операторов. В первом случае выполнилось x=y*2; во втором x=y+4 (т.к. приоритет у присваивания выше, чем у запятой). Обратите внимание, что выражение *без присваивания* (которое может вообще не иметь эффекта или иметь только побочный эффект) вполне законно:

```
x+y;   или   z++;   или   x == y+1;   или   x;
```

В частности, все вызовы функций-процедур именно таковы (это выражения без оператора присваивания, имеющие побочный эффект):

```
f(12,x);        putchar('Ы');
```

в отличие, скажем, от x=**cos**(0.5)/3.0; или c=**getchar**();

Оператор "запятая" разделяет *выражения*, а не просто операторы, поэтому если хоть один из перечисленных операторов не выдает значения, то это является ошибкой:

```
main(){ int i, x = 0;
  for(i=1; i < 4; i++)
      x++, if(x > 2) x = 2; /* используй { ; } */
}
```

оператор **if** не выдает значения. Также логически ошибочно использование функции типа **void** (не возвращающей значения):

```
void f(){}
  ...
  for(i=1; i < 4; i++)
      x++, f();
```

хотя компилятор может допустить такое использование.

Вот еще один пример того, как можно переписать один и тот же фрагмент, применяя разные синтаксические конструкции:

```
if( условие ) { x = 0; y = 0; }
if( условие )   x = 0, y = 0;
if( условие )   x = y = 0;
```

1.109. Найдите опечатку:

```
switch(c){
case 1:
        x++; break;
case 2:
        y++; break;
defalt:
        z++; break;
}
```

Если c=3, то z++ не происходит. Почему? (Потому, что defalt: - это метка, а не ключевое слово **default**).

1.110. Почему программа зацикливается и печатает совсем не то, что нажато на клавиатуре, а только 0 и 1?

```
while ( c = getchar() != 'e')
        printf("%d %c\n, c, c);
```

Ответ: данный фрагмент должен был выглядеть так:

```
while ((c = getchar()) != 'e')
        printf("%d %c\n, c, c);
```

Сравнение в Си имеет высший приоритет, нежели присваивание! Мораль: надо быть внимательнее к приоритетам операций. Еще один пример на похожую тему:

```
       вместо
if(  x & 01  == 0 ) ...     if( c&0377  > 0300)...;
       надо:
if( (x & 01) == 0 ) ...     if((c&0377) > 0300)...;
```

И еще пример с аналогичной ошибкой:

```
FILE *fp;
if( fp = fopen( "файл", "w" ) == NULL ){
    fprintf( stderr, "не могу писать в файл\n");
    exit(1);
}
fprintf(fp,"Good bye, %s world\n","cruel"); fclose(fp);
```

В этом примере файл открывается, но *fp* равно 0 (логическое значение!) и функция **fprintf**() не срабатывает (программа падает по защите памяти†).

Исправьте аналогичную ошибку (на приоритет операций) в следующей функции:

```
/* копирование строки from в to */
char *strcpy( to, from ) register char *from, *to;
{
     char *p = to;
     while( *to++ = *from++ != '\0' );
     return p;
}
```

1.111. Сравнения с нулем (**0**, **NULL**, '**\0**') в Си принято опускать (хотя это не всегда способствует ясности).

---

† "Падать" - программистский жаргон. Означает "аварийно завершаться". "Защита памяти" - обращение по некорректному адресу. В **UNIX** такая ошибка ловится аппаратно, и программа будет убита одним из сигналов: **SIGBUS**, **SIGSEGV**, **SIGILL**. Система сообщит нечто вроде "ошибка шины". Знайте, что это не ошибка аппаратуры и не сбой, а ВАША ошибка!

```
if( i == 0 ) ...;     -->    if( !i ) ... ;
if( i != 0 ) ...;     -->    if(  i ) ... ;
```

например, вместо

```
char s[20], *p ;
for(p=s; *p != '\0'; p++ ) ... ;
     будет
for(p=s; *p; p++ ) ... ;
```

и вместо

```
char s[81], *gets();
while( gets(s) != NULL ) ... ;
     будет
while( gets(s)) ... ;
```

Перепишите **strcpy** в этом более лаконичном стиле.

1.112. Истинно ли выражение

```
if( 2 < 5 < 4 )
```

Ответ: да! Дело в том, что Си не имеет логического типа, а вместо "истина" и "ложь" использует целые значения "не 0" и "0" (логические операции выдают 1 и 0). Данное выражение в условии **if** эквивалентно следующему:

```
((2 < 5) < 4)
```

Значением (2 < 5) будет 1. Значением (1 < 4) будет тоже 1 (истина). Таким образом мы получаем совсем не то, что ожидалось. Поэтому вместо

```
if( a < x < b )
```

надо писать

```
if( a < x && x < b )
```

1.113. Данная программа должна печатать коды вводимых символов. Найдите опечатку; почему цикл сразу завершается?

```
int c;
for(;;) {
    printf("Введите очередной символ:");
    c = getchar();
    if(c = 'e') {
       printf("нажато e, конец\n"); break;
    }
    printf( "Код %03o\n", c & 0377 );
}
```

Ответ: в **if** имеется опечатка: использовано `=' вместо `=='.

Присваивание в Си (а также операции **+=**, **-=**, ***=**, и.т.п.) выдает *новое значение левой части*, поэтому синтаксической ошибки здесь нет! Написанный оператор равносилен

```
c = 'e'; if( c ) ... ;
```

и, поскольку 'e'!= 0, то условие оказывается истинным! Это еще и следствие того, что в Си нет специального логического типа (истина/ложь). Будьте внимательны: компилятор не считает ошибкой использование оператора = вместо == внутри условий **if** и условий циклов (хотя некоторые компиляторы выдают *предупреждение*).

Еще аналогичная ошибка:

```
for( i=0; !(i = 15) ; i++ ) ... ;
```

(цикл не выполняется); или

```
static char s[20] = "   abc"; int i=0;
while(s[i] = ' ') i++;
printf("%s\n", &s[i]); /* должно напечататься abc */
```

(строка заполняется пробелами и цикл не кончается).

То, что оператор присваивания имеет значение, весьма удобно:

```
int x, y, z;            это на самом деле
x = y = z = 1;           x = (y = (z = 1));
```

или†

```
y=f( x += 2 );       // вместо x+=2; y=f(x);
if((y /= 2) > 0)...; // вместо y/=2; if(y>0)...;
```

Вот пример упрощенной игры в "очко" (упрощенной - т.к. не учитывается ограниченность числа карт каждого типа в колоде (по 4 штуки)):

```
#include <stdio.h>
main(){
  int sum = 0, card; char answer[36];
  srand( getpid());  /* рандомизация */
  do{  printf( "У вас %d очков. Еще? ", sum);
       if( *gets(answer) == 'n' ) break;
       /* иначе маловато будет */
       printf( "  %d очков\n",
                card = 6 + rand() % (11 - 6 + 1));
  } while((sum += card) < 21);      /* SIC ! */
  printf ( sum == 21 ? "очко\n"   :
           sum >  21 ? "перебор\n":
                       "%d очков\n", sum);
}
```

Вот еще пример, использующийся для подсчета правильного размера таблицы. Обратите внимание, что присваивания используются в сравнениях, в аргументах вызова функции (**printf**), т.е. везде, где допустимо *выражение*:

```
#include <stdio.h>
int width = 20; /* начальное значение ширины поля */
int len; char str[512];
main(){
  while(gets(str)){
    if((len = strlen(str)) > width){
fprintf(stderr,"width увеличить до %d\n", width=len);
    }
    printf("|%*.*s|\n", -width, width, str);
  }
}
```

Вызывай эту программу как
a.out < *входнойФайл* > /**dev**/**null**

1.114. Почему программа "зависает" (на самом деле - зацикливается) ?

```
int x = 0;
while( x < 100 );
        printf( "%d\n", x++ );
printf( "ВСЕ\n" );
```

Указание: где кончается цикл **while**?

Мораль: не надо ставить **;** где попало. Еще мораль: даже отступы в оформлении программы не являются гарантией отсутствия ошибок в группировке операторов.

1.115. Вообще, приоритеты операций в Си часто не соответствуют ожиданиям нашего здравого смысла. Например, значением выражения:

```
x = 1 << 2 + 1 ;
```

будет 8, а не 5, поскольку сложение выполнится первым. Мораль: в затруднительных и неочевидных случаях лучше явно указывать приоритеты при помощи круглых скобок:

```
x = (1 << 2) + 1 ;
```

Еще пример: увеличивать x на 40, если установлен флаг, иначе на 1:

---

† Конструкция //*текст*, которая будет изредка попадаться в дальнейшем - это комментарий в стиле языка **C++**. Такой комментарий простирается от символа // до конца строки.

```
int bigFlag = 1, x = 2;
x = x + bigFlag ? 40 : 1;
printf( "%d\n", x );
```

ответом будет 40, а не 42, поскольку это

```
x = (x + bigFlag) ? 40 : 1;
```

а не

```
x = x + (bigFlag ? 40 : 1);
```

которое мы имели в виду. Поэтому вокруг условного выражения **?:** обычно пишут круглые скобки.

Заметим, что () указывают только приоритет, но не порядок вычислений. Так, компилятор имеет полное право вычислить

```
long a = 50, x; int b = 4;
x = (a * 100) / b;
  /* деление целочисленное с остатком ! */
и как   x = (a * 100)/b = 5000/4 = 1250
и как   x = (a/b) * 100 = 12*100 = 1200
```

невзирая на наши скобки, поскольку и * и / имеют одинаковый приоритет (хотя это "право" еще не означает, что он обязательно так поступит). Такие операторы приходится разбивать на два, т.е. вводить промежуточную переменную:

```
{ long a100 = a * 100; x = a100 / b; }
```

1.116. Составьте программу вычисления тригонометрической функции. Название функции и значение аргумента передаются в качестве параметров функции **main** (см. про *argv* и *argc* в главе "Взаимодействие с **UNIX**"):

```
$ a.out sin 0.5
sin(0.5)=0.479426
```

(здесь и далее значок **$** обозначает приглашение, выданное интерпретатором команд). Для преобразования строки в значение типа double воспользуйтесь стандартной функцией **atof**().

```
char *str1, *str2, *str3; ...
extern double atof();    double x = atof(str1);
extern long   atol();    long   y = atol(str2);
extern int    atoi();    int    i = atoi(str3);
```

либо

```
sscanf(str1, "%f",  &x);
sscanf(str2, "%ld", &y); sscanf(str3,"%d", &i);
```

К слову заметим, что обратное преобразование - числа в текст - удобнее всего делается при помощи функции **sprintf**(), которая аналогична **printf()**, но сформированная ею строка-сообщение не выдается на экран, а заносится в массив:

```
char represent[ 40 ];
int i = ... ;
sprintf( represent, "%d", i );
```

1.117. Составьте программу вычисления полинома n-ой степени:

```
          n          n-1
Y = A  * X + A    * X    + ... + A0
     n         n-1

    схема (Горнера):

Y = A0 + X * ( A1 + X * ( A2 + ... + X * An ))...)
```

Оформите алгоритм как функцию с переменным числом параметров:

```
poly( x, n, an, an-1, ... a0 );
```

О том, как это сделать - читайте раздел руководства по UNIX **man** *varargs*. Ответ:

```
#include <varargs.h>
double poly(x, n, va_alist)
       double x; int n; va_dcl
{
  va_list args;
  double sum = 0.0;
  va_start(args); /* инициализировать список арг-тов */
  while( n-- >= 0 ){
     sum *= x;
     sum += va_arg(args, double);
     /* извлечь след. аргумент типа double */
  }
  va_end(args);   /* уничтожить список аргументов */
  return sum;
}

main(){
                        /* y = 12*x*x + 3*x + 7 */
  printf( "%g\n", poly(2.0, 2, 12.0,    3.0,  7.0));
}
```

Прототип этой функции:

```
double poly(double x, int n, ... );
```

В этом примере использованы макросы **va_***нечто*. Часть аргументов, которая является списком переменной длины, обозначается в списке параметров как **va_alist**, при этом она объявляется как **va_dcl** в списке типов параметров. Заметьте, что точка-с-запятой после **va_dcl** не нужна! Описание **va_list** *args*; объявляет специальную "связную" переменную; смысл ее машинно зависим. **va_start**(*args*) инициализирует эту переменную списком фактических аргументов, соответствующих **va_alist**-у. **va_end**(*args*) деинициализирует эту переменную (это надо делать обязательно, поскольку инициализация могла быть связана с конструированием списка аргументов при помощи выделения динамической памяти; теперь мы должны уничтожить этот список и освободить память). Очередной аргумент типа *TYPE* извлекается из списка при помощи

```
TYPE x = va_arg(args, TYPE);
```

Список аргументов просматривается слева направо в одном направлении, возврат к предыдущему аргументу невозможен.

Нельзя указывать в качестве типов **char**, **short**, **float**:

```
char ch = va_arg(args, char);
```

поскольку в языке Си аргументы функции таких типов автоматически расширяются в **int**, **int**, **double** соответственно. Корректно будет так:

```
int ch = va_arg(args, int);
```

1.118. Еще об одной ловушке в языке Си на **PDP-11** (и в компиляторах бывают ошибки!):

```
unsigned x = 2;
printf( "%ld %ld",
        - (long) x,
        (long)  -x
);
```

Этот фрагмент напечатает числа -2 и 65534. Во втором случае при приведении к типу **long** был расширен знаковый бит. Встроенная операция **sizeof** выдает значение типа **unsigned**. Подумайте, каков будет эффект в следующем фрагменте программы?

```
static struct point{ int  x,  y    ;}
              p =  {     33, 13   };
FILE *fp = fopen( "00", "w" );

/* вперед на длину одной структуры */
fseek( fp, (long)  sizeof( struct point ), 0 );

/* назад на длину одной структуры */
/*!*/  fseek( fp, (long) -sizeof( struct point ), 1 );

/* записываем в начало файла одну структуру */
fwrite( &p, sizeof p, 1, fp );
```

```
        /* закрываем файл */
        fclose( fp );
```

Где должен находиться минус во втором вызове **fseek** для получения ожидаемого результата? (Данный пример может вести себя по-разному на разных машинах, вопросы касаются **PDP-11**).

1.119. Обратимся к указателям на функции:

```
void g(x){ printf("%d: here\n", x); }
main(){
  void (*f)() = g;  /* Указатель смотрит на функцию g() */
  (*f)(1); /* Старая форма вызова функции по указателю */
    f (2); /* Новая  форма вызова */
  /* В обоих случаях вызывается g(x); */
}
```

Что печатает программа?

```
typedef void (*(*FUN))(); /* Попытка изобразить
       рекурсивный тип typedef FUN (*FUN)(); */
FUN  g(FUN f){ return f; }
void main(){
     FUN y = g(g(g(g(g))));
     if(y == g) printf("OK\n");
}
```

Что печатает программа?

```
        char *f(){
                return "Hello, user!";
        }
        g(func)
            char * (*func)();
        {
                puts((*func)());
        }
        main(){
                g(f);
        }
```

Почему было бы неверно написать

```
        main(){
                g(f());
        }
```

Еще аналогичная ошибка (посмотрите про функцию **signal** в главе "Взаимодействие с **UNIX**"):

```
        #include <signal.h>
        f(){ printf( "Good bye.\n" ); exit(0); }
        main(){
             signal ( SIGINT, f() );
             ...
        }
```

Запомните, что **f**() - это ЗНАЧЕНИЕ функции **f** (т.е. она вызывается и нечто возвращает **return**-ом; это-то значение мы и используем), а **f** - это АДРЕС функции **f** (раньше это так и писалось &**f**), то есть метка начала ее машинных кодов ("точка входа").

1.120. Что напечатает программа? (Пример посвящен указателям на функции и массивам функций):

```
int f(n){ return n*2; }
int g(n){ return n+4; }
int h(n){ return n-1; }
int (*arr[3])() = { f, g, h };
main(){
 int i;
 for(i=0; i < 3; i++ )
     printf( "%d\n", (*arr[i])(i+7) );
}
```

1.121. Что напечатает программа?

```
extern double sin(), cos();
main(){ double x; /* cc -lm */
  for(x=0.0; x < 1.0; x += 0.2)
    printf("%6.4g %6.4g %6.4g\n",
        (x > 0.5 ? sin : cos)(x), sin(x), cos(x));
}
```

то же в варианте

```
extern double sin(), cos();
main(){ double x; double (*f)();
  for(x=0.0; x < 1.0; x += 0.2){
        f = (x > 0.5 ? sin : cos);
        printf("%g\n", (*f)(x));
  }
}
```

1.122. Рассмотрите четыре реализации функции *факториал*:

$$n! = 1 * 2 * \ldots * n$$

или $n! = n * (n-1)!$ где $0! = 1$

Все они иллюстрируют определенные подходы в программировании:

```
/* ЦИКЛ (ИТЕРАЦИЯ) */
int factorial1(n){ int res = 1;
    while(n > 0){ res *= n--; }
    return res;
}

/* ПРОСТАЯ РЕКУРСИЯ */
int factorial2(n){
    return (n==0 ? 1 : n * factorial2(n-1));
}
/* Рекурсия, в которой функция вызывается рекурсивно
 * единственный раз - в операторе return, называется
 * "хвостовой рекурсией" (tail recursion) и
 * легко преобразуется в цикл */

/* АВТОАППЛИКАЦИЯ */
int fi(f, n) int (*f)(), n;
{   if(n == 0) return 1;
    else       return n * (*f)(f, n-1);
}
int factorial3(n){ return fi(fi, n); }

/* РЕКУРСИЯ С НЕЛОКАЛЬНЫМ ПЕРЕХОДОМ */
#include <setjmp.h>
jmp_buf checkpoint;
void fact(n, res) register int n, res;
{   if(n) fact(n - 1, res * n);
    else  longjmp(checkpoint, res+1);
}
int factorial4(n){ int res;
    if(res = setjmp(checkpoint)) return (res - 1);
    else fact(n, 1);
}
```

1.123. Напишите функцию, печатающую целое число в системе счисления с основанием *base*. Ответ:

```
printi( n, base ){
   register int i;

   if( n < 0 ){  putchar( '-' ); n = -n;   }
   if( i = n / base )
           printi( i, base );
   i = n % base ;
   putchar( i >= 10 ? 'A' + i - 10 : '0' + i );
}
```

Попробуйте написать нерекурсивный вариант с накоплением ответа в строке. Приведем рекурсивный вариант, накапливающий ответ в строке s и пользующийся аналогом функции **printi**: функция **prints** - такая же, как **printi**, но вместо вызовов **putchar**(*нечто*); в ней написаны операторы

```
*res++ = нечто;
```

и рекурсивно вызывается конечно же **prints**. Итак:

```
static char *res;
 ... текст функции prints ...
char *itos( n, base, s )
     char *s; /* указывает на char[] массив для ответа */
{
        res = s; prints(n, base); *res = '\0';
        return s;
}
main(){ char buf[20]; printf( "%s\n", itos(19,2,buf); }
```

1.124. Напишите функцию для побитной распечатки целого числа. Имейте в виду, что число содержит 8 * sizeof(int) бит. Указание: используйте операции битового сдвига и &. Ответ:

```
printb(n){
  register i;
  for(i = 8 * sizeof(int) - 1; i >= 0; --i)
     putchar(n & (1 << i) ? '1':'0');
}
```

1.125. Напишите функцию, склоняющую существительные русского языка в зависимости от их числа. Например:

```
printf( "%d кирпич%s", n, grammar( n, "ей", "", "а" ));
```

Ответ:

```
char *grammar( i, s1, s2, s3 )
char *s1, /* прочее */
     *s2, /* один */
     *s3; /* два, три, четыре */
{
       i = i % 100;
       if( i > 10 && i <= 20 ) return s1;
       i = i % 10;
       if( i == 1 ) return s2;
       if( i == 2 || i == 3 || i == 4 )
              return s3;
       return s1;
}
```

1.126. Напишите оператор **printf**, печатающий числа из интервала 0..99 с добавлением нуля перед числом, если оно меньше 10 :

```
00 01 ... 09 10 11 ...
```

Используйте условное выражение, формат.

Ответ:

```
printf ("%s%d", n < 10 ? "0" : "", n);
     либо
printf ("%02d", n );
     либо
printf ("%c%c", '0' + n/10, '0' + n%10 );
```

1.127. Предостережем от одной ошибки, часто допускаемой начинающими.

```
putchar( "c" );   является ошибкой.
putchar( 'c' );   верно.
```

Дело в том, что **putchar** требует аргумент - символ, тогда как "c" - СТРОКА из одного символа. Большинство компиляторов (те, которые не проверяют прототипы вызова стандартных функций) НЕ обнаружит здесь никакой синтаксической ошибки (кстати, ошибка эта - семантическая).

Также ошибочны операторы

```
printf ( '\n' ); /* нужна строка */
putchar( "\n" ); /* нужен символ */
putchar( "ab" ); /* нужен символ */
putchar( 'ab' ); /* ошибка в буквенной константе */

char c; if((c = getchar()) == "q" ) ... ;
/* нужно писать 'q' */
```

Отличайте строку из одного символа и символ - это разные вещи! (Подробнее об этом - в следующей главе).

1.128. Весьма частой является ошибка "*промах на единицу*", которая встречается в очень многих и разнообразных случаях. Вот одна из возможных ситуаций:

```
int m[20]; int i = 0;
while( scanf( "%d", & m[i++] ) != EOF );
printf( "Ввели %d чисел\n", i );
```

В итоге *i* окажется на 1 больше, чем ожидалось. Разберитесь в чем дело.

Ответ: аргументы функции вычисляются **до** ее вызова, поэтому когда мы достигаем конца файла и **scanf** возвращает **EOF**, *i*++ в вызове **scanf** все равно делается. Надо написать

```
while( scanf( "%d", & m[i] ) != EOF ) i++;
```

1.129. Замечание по стилистике: при выводе сообщения на экран

```
printf( "Hello    \n" );
```

пробелы перед **\n** достаточно бессмысленны, поскольку на экране никак не отобразятся. Надо писать (экономя память)

```
printf( "Hello\n" );
```

Единственный случай, когда такие пробелы значимы - это когда вы выводите текст инверсией. Тогда пробелы отображаются как светлый фон.

Еще неприятнее будет

```
printf( "Hello\n     " );
```

поскольку концевые пробелы окажутся в *начале* следующей строки.

1.130. **printf** - интерпретирующая функция, т.е. работает она довольно медленно. Поэтому вместо

```
char s[20]; int i;
  ...
printf( "%c", s[i] );   и   printf( "\n" );
```

надо всегда писать

```
putchar( s[i] );        и   putchar( '\n' );
```

поскольку **printf** в конце-концов (сделав все преобразования по формату) внутри себя вызывает **putchar**. Так сделаем же это сразу!

1.131. То, что параметр "формат" в функции **printf** может быть выражением, позволяет делать некоторые удобные вещи. Например:

```
int x; ...
printf( x ? "значение x=%d\n" : "x равен нулю\n\n", x);
```

Формат здесь - условное выражение. Если *x*!=0, то будет напечатано значение *x* по формату **%d**. Если же *x*==0, то будет напечатана строка, не содержащая ни одного %-та. В результате аргумент *x* в списке аргументов будет просто проигнорирован. Однако, например

```
int x = ... ;
printf( x > 30000 ? "%f\n" : "%d\n", x);
```

(чтобы большие *x* печатались в виде 31000.000000) незаконно, поскольку целое число нельзя печатать по формату **%f** ни в каких случаях. Единственным способом сделать это является явное приведение *x* к типу **double**:

```
printf("%f\n", (double) x);
```

Будет ли законен оператор?

```
printf( x > 30000 ? "%f\n" : "%d\n",
        x > 30000 ? (double) x : x );
```

Ответ: нет. Условное выражение для аргумента будет иметь "старший" тип - **double**. А значение типа **double** нельзя печатать по формату **%d**. Мы должны использовать здесь оператор **if**:

```
if( x > 30000 ) printf("%f\n", (double)x);
else            printf("%d\n", x);
```

1.132. Напишите функцию, печатающую размер файла в удобном виде: если файл меньше одного килобайта - печатать его размер в байтах, если же больше - в килобайтах (и мегабайтах).

```
#define KBYTE    1024L  /* килобайт */
#define THOUSAND 1024L  /* кб. в мегабайте */
void tellsize(unsigned long sz){
  if(sz < KBYTE) printf("%lu байт", sz);
  else{
    unsigned long Kb  = sz/KBYTE;
    unsigned long Mb  = Kb/THOUSAND;
    unsigned long Dec = ((sz % KBYTE) * 10) / KBYTE;
      if( Mb ){
          Kb %= THOUSAND;
printf( Dec ? "%lu.%03lu.%01lu Мб." : "%lu.%lu Мб.",
                Mb,   Kb,   Dec );
       } else
printf( Dec ? "%lu.%01lu Кб.":"%lu Кб.", Kb, Dec);
  }
  putchar('\n');
}
```

1.133. Для печати строк используйте

```
        printf("%s", string);   /* A */
но не   printf(string);          /* B */
```

Если мы используем вариант **B**, а в строке встретится символ '%'

```
char string[] = "abc%defg";
```

то **%d** будет воспринято как *формат* для вывода целого числа. Во-первых, сама строка %d не будет напечатана; во-вторых - что же будет печататься по этому формату, когда у нас есть лишь *единственный* аргумент - *string*?! Напечатается какой-то мусор!

1.134. Почему оператор

```
char s[20];
scanf("%s", s); printf("%s\n", s);
```

в ответ на ввод строки

```
  Пушкин  A.C.
```

печатает только "Пушкин"?

Ответ: потому, что концом текста при вводе по формату **%s** считается либо **\n**, либо пробел, либо табуляция, а не только **\n**; то есть формат **%s** читает *слово* из текста. Чтение всех символов до конца строки, (включая пробелы) должно выглядеть так:

```
scanf("%[^\n]\n", s);
  %[^\n] - читать любые символы, кроме \n (до \n)
  \n     - пропустить \n на конце строки
  %[abcdef] - читать слово,
              состоящее из перечисленных букв.
  %[^abcde] - читать слово из любых букв,
    кроме перечисленных (прерваться по букве из списка).
```

Пусть теперь строки входной информации имеют формат:

```
Фрейд Зигмунд 1856 1939
```

Пусть мы хотим считывать в строку *s* фамилию, в целое *y* - год рождения, а прочие поля - игнорировать. Как это сделать? Нам поможет формат "подавление присваивания" **%***:

```
scanf("%s%*s%d%*[^\n]\n",
      s,   &y );
```

%* *пропускает* поле по формату, указанному после *, не занося его значение ни в какую переменную, а просто "забывая" его. Так формат

```
"%*[^\n]\n"
```

игнорирует "хвост" строки, включая символ перевода строки.

Символы " ", "**\t**", "**\n**" в формате вызывают пропуск всех пробелов, табуляций, переводов строк во входном потоке, что можно описать как

```
int c;
while((c = getc(stdin))== ' ' || c == '\t' || c == '\n' );
```

либо как формат

```
%*[ \t\n]
```

Перед числовыми форматами (**%d**, **%o**, **%u**, **%ld**, **%x**, **%e**, **%f**), а также **%s**, пропуск пробелов делается автоматически. Поэтому

```
scanf("%d%d",  &x, &y);
        и
scanf("%d %d", &x, &y);
```

равноправны (пробел перед вторым **%d** просто не нужен). Неявный пропуск пробелов не делается перед **%c** и **%[... ,** поэтому в ответ на ввод строки "**12 5 x**" пример

```
main(){ int n, m; char c;
   scanf("%d%d%c", &n, &m, &c);
   printf("n=%d m=%d c='%c'\n", n, m, c);
}
```

напечатает "n=12 m=5 c=' '", то есть в *c* будет прочитан пробел (предшествовавший **x**), а не **x**.

Автоматический пропуск пробелов перед **%s** не позволяет считывать по **%s** строки, *лидирующие* пробелы которых должны сохраняться. Чтобы лидирующие пробелы также считывались, следует использовать формат

```
scanf("%[^\n]%*1[\n]", s);
```

в котором модификатор длины 1 заставляет игнорировать только один символ **\n**, а не ВСЕ пробелы и переводы строк, как "**\n**". К сожалению (как показал эксперимент) этот формат не в состоянии прочесть пустую строку (состоящую только из **\n**). Поэтому можно сделать глобальный вывод: строки надо считывать при помощи функций **gets**() и **fgets**()!

1.135. Еще пара слов про **scanf**: **scanf** возвращает число успешно прочитанных им данных (обработанных **%**-ов) или **EOF** в конце файла. Неудача может наступить, если данное во входном потоке не соответствует формату, например строка

```
12 quack
```

для

```
int d1; double f; scanf("%d%lf", &d1, &f);
```

В этом случае **scanf** прочтет 12 по формату **%d** в переменную *d1*, но слово *quack* не отвечает формату **%lf**, поэтому **scanf** прервет свою работу и выдаст значение 1 (успешно прочел один формат). Строка *quack* останется невостребованной - ее прочитают последующие вызовы функций чтения; а сейчас *f* останется

неизмененной.

1.136. Си имеет квалификатор **const**, указывающий, что значение является не переменной, а константой, и попытка изменить величину по этому имени является ошибкой. Во многих случаях **const** может заменить **#define**, при этом еще явно указан тип константы, что полезно для проверок компилятором.

```
const int x = 22;
x = 33; /* ошибка: константу нельзя менять */
```

Использование **const** с указателем:

Указуемый объект - константа

```
const char *pc = "abc";
pc[1]  = 'x';    /* ошибка */
pc     = "123";  /* OK */
```

Сам указатель - константа

```
char *const cp = "abc";
cp[1]  = 'x';    /* OK */
cp     = "123";  /* ошибка */
```

Указуемый объект и сам указатель - константы

```
const char *const cpc = "abc";
cpc[1] = 'x';    /* ошибка */
cpc    = "123";  /* ошибка */
```

Указатель на константу **необходимо** *объявлять как const TYPE**

```
      int a = 1;
const int b = 2;

const int *pca  = &a;   /* OK, просто рассматриваем a как константу */
const int *pcb  = &b;   /* OK */

      int *pb   = &b;   /* ошибка, так как тогда возможно было бы написать */
          *pb   = 3;    /* изменить константу b */
```

1.137. Стандартная функция быстрой сортировки **qsort** (алгоритм *quick sort*) имеет такой формат: чтобы отсортировать массив элементов типа **TYPE**

```
TYPE arr[N];
      надо вызывать
qsort(arr,/* Что сортировать? Не с начала: arr+m    */
      N,  /* Сколько первых элементов массива?      */
          /* можно сортировать только часть: n < N  */
      sizeof(TYPE),/* Или sizeof arr[0]             */
                   /* размер одного элемента массива*/
      cmp);
```

где

```
int cmp(TYPE *a1, TYPE *a2);
```

функция сравнения элементов **a1* и **a2*. Ее аргументы - АДРЕСА двух каких-то элементов сортируемого массива. Функцию *cmp* мы должны написать сами - это функция, задающая упорядочение элементов массива. Для сортировки *по возрастанию* функция **cmp**() должна возвращать целое

```
< 0, если  *a1 должно идти раньше *a2    <
= 0, если  *a1 совпадает с        *a2   ==
> 0, если  *a1 должно идти после  *a2    >
```

Для массива *строк* элементы массива имеют тип (**char ***), поэтому аргументы функции имеют тип (**char ****). Требуемому условию удовлетворяет такая функция:

```
char *arr[N]; ...
cmps(s1, s2) char **s1, **s2;
{ return strcmp(*s1, *s2); }
```

(Про **strcmp** смотри раздел "Массивы и строки"). Заметим, что в некоторых системах программирования

(например в **TurboC++** †) вы должны использовать функцию сравнения с прототипом

```
int cmp (const void *a1, const void *a2);
```

и внутри нее явно делать приведение типа:

```
cmps (const void *s1, const void *s2)
{ return strcmp(*(char **)s1, *(char **)s2); }
```

или можно поступить следующим образом:

```
int cmps(char **s1, char **s2){
    return strcmp(*s1, *s2);
}
typedef int (*CMPS)(const void *, const void *);
qsort((void *) array, ..., ..., (CMPS) cmps);
```

Наконец, возможно и просто объявить

```
int cmps(const void *A, const void *B){
    return strcmp(A, B);
}
```

Для массива *целых* годится такая функция сравнения:

```
int arr[N]; ...
cmpi(i1, i2) int *i1, *i2;
{ return *i1 - *i2; }
```

Для массива *структур*, которые мы сортируем по целому полю *key*, годится

```
struct XXX{ int key; ... } arr[N];
cmpXXX(st1, st2) struct XXX *st1, *st2;
{ return( st1->key  -  st2->key ); }
```

Пусть у нас есть массив *long*. Можно ли использовать

```
long arr[N]; ...
cmpl(L1, L2) long *L1, *L2;
{ return *L1 - *L2; }
```

Ответ: оказывается, что нет. Функция **cmpl** должна возвращать целое, а разность двух **long**-ов имеет тип **long**. Поэтому компилятор приводит эту разность к типу **int** (как правило *обрубанием* старших битов). При этом (если **long**-числа были велики) результат может изменить знак! Например:

```
main(){
  int n; long a = 1L; long b = 777777777L;
  n = a - b;  /* должно бы быть отрицательным... */
  printf( "%ld %ld %d\n", a, b, n );
}
```

печатает 1 777777777 3472. Функция сравнения должна выглядеть так:

```
cmpl(L1, L2) long *L1, *L2; {
        if( *L1 == *L2 ) return   0;
        if( *L1 <  *L2 ) return (-1);
                         return   1;
}
```

или

```
cmpl(L1, L2) long *L1, *L2; {
    return( *L1 == *L2 ?  0 :
            *L1 <  *L2 ? -1 : 1 );
}
```

поскольку важна не величина возвращенного значения, а только ее *знак*.

Учтите, что для использования функции сравнения вы должны либо определить функцию сравнения *до* ее использования в **qsort**():

```
int cmp(...){ ... } /* реализация */
        ...
qsort(..... , cmp);
```

либо предварительно объявить имя функции сравнения, чтобы компилятор понимал, что это именно

---

† **TurboC** - компилятор Си в **MS DOS**, разработанный фирмой **Borland International**.

функция:

```
int cmp();
qsort(..... , cmp);
        ...
int cmp(...){ ... } /* реализация */
```

1.138. Пусть у нас есть две программы, пользующиеся одной и той же структурой данных W:

```
 a.c                          b.c
-------------------------   ------------------------------
#include <fcntl.h>           #include <fcntl.h>
struct W{ int x,y; }a;       struct W{ int x,y; }b;
main(){  int fd;             main(){  int fd;
  a.x = 12; a.y = 77;          fd = open("f", O_RDONLY);
  fd = creat("f", 0644);       read(fd, &b, sizeof b);
  write(fd, &a, sizeof a);     close(fd);
  close(fd);                    printf("%d %d\n", b.x, b.y);
}                             }
```

Что будет, если мы изменим структуру на

```
struct W { long x,y; };
        или
struct W { char c; int x,y; };
```

в файле *a.c* и забудем сделать это в *b.c*? Будут ли правильно работать эти программы?

Из наблюдаемого можно сделать вывод, что если две или несколько программ (или частей одной программы), размещенные в разных файлах, используют общие

- типы данных (**typedef**);
- структуры и объединения;
- константы (определения **#define**);
- прототипы функций;

то их определения лучше выносить в общий *include-файл* (*header-файл*), дабы все программы придерживались одних и тех же общих соглашений. Даже если эти соглашения со временем изменятся, то они изменятся во всех файлах синхронно и как бы сами собой. В нашем случае исправлять определение структуры придется только в include-файле, а не выискивать все места, где оно написано, ведь при этом немудрено какое-нибудь место и пропустить!

```
 W.h
----------------------
struct W{ long x, y; };
```

```
 a.c                          b.c
-------------------------   ------------------
#include <fcntl.h>           #include <fcntl.h>
#include "W.h"               #include "W.h"
struct W a;                  struct W b;
main(){  ...                 main(){  ...
                                printf("%ld...
```

Кроме того, вынесение общих фрагментов текста программы (определений структур, констант, и.т.п.) в отдельный файл экономит наши силы и время - вместо того, чтобы набивать один и тот же текст много раз в разных файлах, мы теперь пишем в каждом файле *единственную* строку - директиву **#include**. Кроме того, экономится и место на диске, ведь программа стала короче! Файлы включения имеют суффикс **.h**, что означает "header-file" (файл-заголовок).

Синхронную перекомпиляцию всех программ в случае изменения include-файла можно задать в файле *Makefile* - программе для координатора **make**†:

---

† Подробное описание **make** смотри в документации по системе **UNIX**.

```
all: a b
        echo Запуск a и b
        a ; b
a: a.c W.h
        cc a.c -o a
b: b.c W.h
        cc b.c -o b
```

Правила **make** имеют вид

```
цель: список_целей_от_которых_зависит
        команда
```

команда описывает что нужно сделать, чтобы изготовить файл *цель* из файлов *список_целей_от_которых_зависит*. Команда выполняется только если файл *цель* еще не существует, либо *хоть один из* файлов справа от двоеточия является более "молодым" (свежим), чем целевой файл (смотри поле *st_mtime* и сисвызов **stat** в главе про **UNIX**).

1.139. Программа на Си может быть размещена в нескольких файлах. Каждый файл выступает в роли "модуля", в котором собраны сходные по назначению функции и переменные. Некоторые переменные и функции можно сделать невидимыми для других модулей. Для этого надо объявить их **static**:

- Объявление переменной *внутри* функции как **static** делает переменную статической (т.е. она будет сохранять свое значение при выходе из функции) и ограничивает ее видимость пределами *данной функции*.
- Переменные, описанные вне функций, и так являются статическими (по классу памяти). Однако слово **static** и в этом случае позволяет управлять видимостью этих переменных - они будут видимы только в пределах *данного файла*.
- Функции, объявленные как **static**, также видимы только в пределах *данного файла*.
- Аргументы функции и локальные (автоматические) переменные функции и так существуют только на время вызова данной функции (память для них выделяется в стеке при входе в функцию и уничтожается при выходе) и видимы только внутри ее тела. Аргументы функции *нельзя* объявлять **static**:

  ```
  f(x) static x; { x++; }
  ```

  незаконно.

Таким образом все переменные и функции в данном файле делятся на две группы:

- Видимые только внутри данного файла (локальные для модуля). Такие имена объявляются с использованием ключевого слова **static**. В частности есть еще "более локальные" переменные - автоматические локалы функций и их формальные аргументы, которые видимы только в пределах данной функции. Также видимы лишь в пределах одной функции статические локальные переменные, объявленные в теле функции со словом **static**.
- Видимые во всех файлах (глобальные имена).

Глобальные имена образуют *интерфейс* модуля и могут быть использованы в других модулях. Локальные имена извне модуля недоступны.

Если мы используем в файле-модуле функции и переменные, входящие в интерфейс *другого* файла-модуля, мы должны объявить их как **extern** ("внешние"). Для функций описатели **extern** и **int** можно опускать:

```
// файл A.c
int x, y, z;              // глобальные
char ss[200];              // глоб.
static int v, w;           // локальные
static char *s, p[20];     // лок.
int f(){ ... }             // глоб.
char *g(){ ... }           // глоб.
static int h(){ ... }      // лок.
static char *sf(){ ... }   // лок.
int fi(){ ... }            // глоб.

// файл B.c
extern int x, y;
extern z;            // int можно опустить
extern char ss[];    // размер можно опустить
extern int f();
char *g();           // extern можно опустить
```

```
extern fi();        // int можно опустить
```

Хорошим тоном является написание комментария - из какого модуля или библиотеки импортируется переменная или функция:

```
extern int x, y;  /* import from A.c     */
char *tgetstr();  /* import from termlib */
```

Следующая программа собирается из файлов *A.c* и *B.c* командой£

```
cc A.c B.c -o AB
```

Почему компилятор сообщает "*x* дважды определено"?

```
 файл A.c                 файл B.c
-----------------------------------------
int x=12;                int x=25;
main(){                  f(y) int *y;
  f(&x);                 {
  printf("%d\n", x);        *y += x;
}                        }
```

Ответ: потому, что в каждом файле описана *глобальная* переменная *x*. Надо в одном из них (или в обоих сразу) сделать *x* локальным именем (исключить его из интерфейса модуля):
static int *x*=...;
Почему в следующем примере компилятор сообщает "**_f** дважды определено"?

```
 файл A.c                 файл B.c
-----------------------------------------------------
int x;                   extern int x;
main(){ f(5); g(77); }   g(n){ f(x+n); }
f(n)  { x=n;         }   f(m){ printf("%d\n", m); }
```

Ответ: надо сделать в файле *B.c* функцию **f** локальной: **static f**(m)...

Хоть в одном файле должна быть определена функция **main**, вызываемая *системой* при запуске программы. Если такой функции нигде нет - компилятор выдает сообщение "**_main** неопределено". Функция **main** должна быть определена *один* раз! В файле она может находиться в *любом* месте - не требуется, чтобы она была самой первой (или последней) функцией файла£.

1.140. В чем ошибка?

```
 файл A.c                 файл B.c
-----------------------------------------------------
extern int x;            extern int x;
main(){ x=2;             f(){
        f();                printf("%d\n", x);
}                        }
```

Ответ: переменная *x* в обоих файлах объявлена как **extern**, в результате память для нее нигде не выделена, т.е. *x* не определена ни в одном файле. Уберите одно из слов **extern**!

1.141. В чем ошибка?

£ Можно задать *Makefile* вида

```
CFLAGS = -O
AB:     A.o     B.o
        cc A.o B.o -o AB
A.o:    A.c
        cc -c $(CFLAGS) A.c
B.o:    B.c
        cc -c $(CFLAGS) B.c
```

и собирать программу просто вызывая команду **make**.

£ Если вы пользуетесь "новым" стилем объявления функций, но не используете прототипы, то следует определять каждую функцию **до** первого места ее использования, чтобы компилятору в точке вызова был известен ее заголовок. Это приведет к тому, что **main**() окажется *последней* функцией в файле - ее не вызывает никто, зато она вызывает кого-то еще.

```
 файл A.c                файл B.c
-------------------------------------------------
 int x;                  extern double x;
 ...                     ...
```

Типы переменных не совпадают. Большинство компиляторов не ловит такую ошибку, т.к. каждый файл компилируется отдельно, независимо от остальных, а при "склейке" файлов в общую выполняемую программу компоновщик знает лишь *имена* переменных и функций, но не их типы и прототипы. В результате программа нормально скомпилируется и соберется, но результат ее выполнения будет непредсказуем! Поэтому объявления **extern** тоже полезно выносить в include-файлы:

```
файл proto.h
------------------
extern int x;

файл A.c                файл B.c
------------------      ------------------
#include "proto.h"      #include "proto.h"
int x;                  ...
```

то, что переменная *x* в *A.c* оказывается описанной и как **extern** - вполне допустимо, т.к. в момент настоящего объявления этой переменной это слово начнет просто игнорироваться (лишь бы типы в объявлении с **extern** и без него совпадали - иначе ошибка!).

1.142. Что печатает программа и почему?

```
int a = 1;  /* пример Bjarne Stroustrup-a */
void f(){
  int b = 1;
  static int c = 1;
  printf("a=%d b=%d c=%d\n", a++, b++, c++);
}
void main(){
  while(a < 4) f();
}
```

Ответ:

```
a=1 b=1 c=1
a=2 b=1 c=2
a=3 b=1 c=3
```

1.143. Автоматическая переменная видима только внутри блока, в котором она описана. Что напечатает программа?

```
/* файл A.c */
int x=666;  /*глоб.*/
main(){
  f(3);
  printf(" ::x = %d\n", x);
  g(2); g(5);
  printf(" ::x = %d\n", x);
}
g(n){
  static int x=17; /*видима только в g*/
  printf("g::x = %2d g::n = %d\n", x++, n);
  if(n) g(n-1); else x = 0;
}

/* файл B.c */
extern x;     /*глобал*/
f(n){         /*локал функции*/
  x++;        /*глобал*/
  { int x;    /*локал блока*/
    x = n+1;  /*локал*/
    n = 2*x;  /*локал*/
  }
  x = n-1;    /*глобал*/
}
```

1.144. Функция, которая

- не содержит внутри себя статических переменных, хранящих состояние процесса обработки данных (функция без "памяти");
- получает значения параметров *только* через свои аргументы (но не через глобальные статические переменные);
- возвращает значения *только* через аргументы, либо как значение функции (через **return**);

называется *реентерабельной* (повторно входимой) или *чистой* (pure). Такая функция может параллельно (или псевдопараллельно) использоваться несколькими "потоками" обработки информации в нашей программе, без какого-либо непредвиденного влияния этих "потоков обработки" друг на друга. Первый пункт требований позволяет функции не зависеть ни от какого конкретного процесса обработки данных, т.к. она не "помнит" обработанных ею ранее данных и не строит свое поведение в зависимости от них. Вторые два пункта - это требование, чтобы *все без исключения* пути передачи данных в функцию и из нее (интерфейс функции) были перечислены в ее заголовке. Это лишает функцию "побочных эффектов", не предусмотренных программистом при ее вызове (программист обычно смотрит только на заголовок функции, и не выискивает "тайные" связи функции с программой через глобальные переменные, если только это специально не оговорено). Вот пример **не** реентерабельной функции:

```
FILE *fp; ...  /* глобальный аргумент */
char delayedInput ()
{
     static char prevchar;  /* память */
     char c;
     c = prevchar;
     prevchar = getc (fp);
     return c;
}
```

А вот ее реентерабельный эквивалент:

```
char delayedInput (char *prevchar, FILE *fp)
{
     char c;
     c = *prevchar;
     *prevchar = getc (fp);
     return c;
}
/* вызов: */
FILE *fp1, *fp2; char prev1, prev2, c1, c2;
 ... x1 = delayedInput (&prev1, fp1);
     x2 = delayedInput (&prev2, fp2); ...
```

Как видим, все "запоминающие" переменные (т.е. *prevchar*) вынесены из самой функции и подаются в нее в виде аргумента.

Реентерабельные функции независимы от остальной части программы (их можно скопировать в другой программный проект без изменений), более понятны (поскольку *все* затрагиваемые ими внешние переменные перечислены как аргументы, не надо выискивать в теле функции *глобальных* переменных, передающих значение в/из функции, т.е. эта функция не имеет *побочных* влияний), более надежны (хотя бы потому, что компилятор в состоянии проверить прототип такой функции и предупредить вас, если вы забыли задать какой-то аргумент; если же аргументы передаются через глобальные переменные - вы можете забыть проинициализировать какую-то из них). Старайтесь делать функции реентерабельными!

Вот еще один пример на эту тему. Не-реентерабельный вариант:

```
int x, y, result;
int f (){
        static int z = 4;
        y = x + z; z = y - 1;
        return x/2;
}
Вызов:     x=13; result = f(); printf("%d\n", y);
```

А вот реентерабельный эквивалент:

```
int y, result, zmem = 4;
int f (/*IN*/ int x, /*OUT*/ int *ay, /*INOUT*/ int *az){
        *az = (*ay = x + *az) - 1;
        return x/2;
}
Вызов:    result = f(13, &y, &zmem); printf("%d\n", y);
```

1.145. То, что формат заголовка функции должен быть известен компилятору до момента ее использования, побуждает нас помещать определение функции до точки ее вызова. Так, если *main* вызывает *f*, а *f* вызывает *g*, то в файле функции расположатся в порядке

```
g()   {              }
f()   { ... g(); ... }
main(){ ... f(); ... }
```

Программа обычно *разрабатывается* "сверху-вниз" - от *main* к деталям. Си же вынуждает нас размещать функции в программе в обратном порядке, и в итоге программа *читается* снизу-вверх - от деталей к *main*, и читать ее следует от конца файла к началу!

Так мы вынуждены писать, чтобы удовлетворить Си-компилятор:

```
#include <stdio.h>

unsigned long g(unsigned char *s){
        const int BITS = (sizeof(long) * 8);
        unsigned long sum = 0;

        for(;*s; s++){
                sum ^= *s;
                /* cyclic rotate left */
                sum = (sum<<1)|(sum>>(BITS-1));
        }
        return sum;
}
void f(char *s){
        printf("%s %lu\n", s, g((unsigned char *)s));
}
int main(int ac, char *av[]){
        int i;

        for(i=1; i < ac; i++)
                f(av[i]);
        return 0;
}
```

А вот как мы разрабатываем программу:

```
#include <stdio.h>

int main(int ac, char *av[]){
        int i;

        for(i=1; i < ac; i++)
                f(av[i]);
        return 0;
}
void f(char *s){
        printf("%s %lu\n", s, g((unsigned char *)s));
}
unsigned long g(unsigned char *s){
        const int BITS = (sizeof(long) * 8);
        unsigned long sum = 0;

        for(;*s; s++){
                sum ^= *s;
                /* cyclic rotate left */
                sum = (sum<<1)|(sum>>(BITS-1));
        }
        return sum;
}
```

и вот какую ругань производит Си-компилятор в ответ на эту программу:

```
"0000.c", line 10: identifier redeclared: f
        current : function(pointer to char) returning void
        previous: function() returning int : "0000.c", line 7
"0000.c", line 13: identifier redeclared: g
        current : function(pointer to uchar) returning ulong
        previous: function() returning int : "0000.c", line 11
```

Решением проблемы является - задать прототипы (объявления заголовков) всех функций в *начале файла* (или даже вынести их в header-файл).

```
#include <stdio.h>

int main(int ac, char *av[]);
void f(char *s);
unsigned long g(unsigned char *s);
        ...
```

Тогда функции будет **можно** располагать в тексте в любом порядке.

1.146. Рассмотрим процесс сборки программы из нескольких файлов на языке Си. Пусть мы имеем файлы *file1.c*, *file2.c*, *file3.c* (один из них должен содержать среди других функций функцию **main**). Ключ компилятора **-o** заставляет создавать выполняемую программу с именем, указанным после этого ключа. Если этот ключ не задан - будет создан выполняемый файл *a.out*

```
cc file1.c file2.c file3.c -o file
```

Мы получили выполняемую программу *file*. Это эквивалентно 4-м командам:

```
cc -c file1.c           получится     file1.o
cc -c file2.c                         file2.o
cc -c file3.c                         file3.o
cc file1.o file2.o file3.o -o file
```

Ключ **-c** заставляет компилятор превратить файл на языке Си в "объектный" файл (содержащий машинные команды; не будем вдаваться в подробности). Четвертая команда "склеивает" объектные файлы в единое целое - выполняемую программу†. При этом, если какие-то функции, используемые в нашей программе, не были определены (т.е. спрограммированы нами) ни в одном из наших файлов - будет просмотрена библиотека стандартных функций. Если же каких-то функций не окажется и там - будет выдано сообщение об ошибке.

---

† На самом деле, для "склейки" объектных файлов в выполняемую программу, команда **/bin/cc** вызывает программу **/bin/ld** - link editor, linker, редактор связей, компоновщик.

Если у нас уже есть какие-то готовые объектные файлы, мы можем транслировать только новые Си-файлы:

```
cc -c file4.c
cc file1.o file2.o file3.o file4.o -o file
   или (что то же самое,
   но cc сам разберется, что надо делать)
cc file1.o file2.o file3.o file4.c -o file
```

Существующие у нас объектные файлы с отлаженными функциями удобно собрать в *библиотеку* - файл специальной структуры, содержащий все указанные файлы (все файлы склеены в один длинный файл, разделяясь специальными заголовками, см. include-файл <*ar.h*>):

```
ar r file.a file1.o file2.o file3.o
```

Будет создана библиотека *file*.**a**, содержащая перечисленные **.o** файлы (имена библиотек в **UNIX** имеют суффикс **.a** - от слова *archive*, архив). После этого можно использовать библиотеку:

```
cc file4.o file5.o file.a -o file
```

Механизм таков: если в файлах *file4.o* и *file5.o* не определена какая-то функция (функции), то просматривается библиотека, и в список файлов для "склейки" добавляется файл из библиотеки, содержащий определение этой функции (из библиотеки он не удаляется!). Тонкость: из библиотеки берутся не ВСЕ файлы, а лишь те, которые содержат определения недостающих функций£. Если, в свою очередь, файлы, извлекаемые из библиотеки, будут содержать неопределенные функции - библиотека (библиотеки) будут просмотрены еще раз и.т.д. (на самом деле достаточно максимум двух проходов, так как при первом просмотре библиотеки можно составить ее *каталог*: где какие функции в ней содержатся и кого вызывают). Можно указывать и несколько библиотек:

```
cc file6.c file7.o  \
   file.a mylib.a /lib/libLIBR1.a -o file
```

Таким образом, в команде **cc** можно смешивать имена файлов: исходных текстов на Си **.c**, объектных файлов **.o** и файлов-библиотек **.a**.

Просмотр библиотек, находящихся в стандартных местах (каталогах **/lib** и **/usr/lib**), можно включить и еще одним способом: указав ключ **-l**. Если библиотека называется

```
/lib/libLIBR1.a   или   /usr/lib/libLIBR2.a
```

то подключение делается ключами

```
-lLIBR1          и      -lLIBR2
```

соответственно.

```
cc file1.c file2.c file3.o mylib.a -lLIBR1 -o file
```

Список библиотек и ключей **-l** должен идти **после** имен всех исходных **.c** и объектных **.o** файлов.

Библиотека стандартных функций языка Си */lib/libc.a* (ключ **-l***c*) подключается *автоматически* ("подключить" библиотеку - значит вынудить компилятор просматривать ее при сборке, если какие-то функции, использованные вами, не были вами определены), то есть просматривается *всегда* (именно эта библиотека содержит коды, например, для **printf**, **strcat**, **read**).

Многие прикладные пакеты функций поставляются именно в виде библиотек. Такие библиотеки состоят из ряда **.o** файлов, содержащих объектные коды для различных функций (т.е. функции в скомпилированном виде). Исходные тексты от большинства библиотек не поставляются (так как являются коммерческой тайной). Тем не менее, вы можете использовать эти функции, так как вам предоставляются разработчиком:

- описание (документация).
- include-файлы, содержащие форматы данных используемые функциями библиотеки (именно эти файлы включались **#include** в исходные тексты библ. функций. Теперь уже вы должны включать их в свою программу).

Таким образом вы знаете, как надо *вызывать* библиотечные функции и какие структуры данных вы должны использовать в своей программе для обращения к ним (хотя и не имеете текстов самих библиотечных функций, т.е. не знаете, как они устроены. Например, вы часто используете **printf**(), но задумываетесь ли вы о ее внутреннем устройстве?). Некоторые библиотечные функции могут быть вообще написаны не на Си, а

£ Поэтому библиотека может быть очень большой, а к нашей программе "приклеится" лишь небольшое число файлов из нее. В связи с этим стремятся делать файлы, помещаемые в библиотеку, как можно меньше: 1 функция; либо "пачка" функций, вызывающих друг друга.

на ассемблере или другом языке программирования††. Еще раз обращаю ваше внимание, что библиотека содержит *не исходные тексты* функций, а скомпилированные *коды* (и include-файлы содержат (как правило) не тексты функций, а только описание форматов данных)! Библиотека может также содержать статические данные, вроде массивов строк-сообщений об ошибках.

Посмотреть список файлов, содержащихся в библиотеке, можно командой

ar tv *имяФайлаБиблиотеки*

а список имен функций - командой

nm *имяФайлаБиблиотеки*

Извлечь файл (файлы) из архива (скопировать его в текущий каталог), либо удалить его из библиотеки можно командами

ar x *имяФайлаБиблиотеки имяФайла1 ...*
ar d *имяФайлаБиблиотеки имяФайла1 ...*

где ... означает список имен файлов.

"Лицом" библиотек служат прилагаемые к ним include-файлы. Системные include-файлы, содержащие общие форматы данных для стандартных библиотечных функций, хранятся в каталоге */usr/include* и подключаются так:

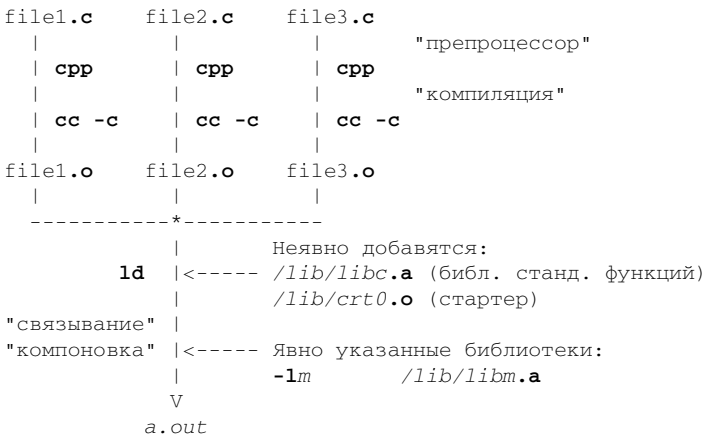
для **/usr/include/***файл.h* надо #include <*файл.h*>
для **/usr/include/sys/***файл.h* #include <**sys/***файл.h*>

(**sys** - это каталог, где описаны форматы данных, используемых ядром ОС и системными вызовами). Ваши собственные include-файлы (посмотрите в предыдущий раздел!) ищутся в текущем каталоге и включаются при помощи

```
#include "файл.h"          /*  ./файл.h       */
#include "../h/файл.h"     /*  ../h/файл.h    */
#include "/usr/my/файл.h" /*  /usr/my/файл.h */
```

Непременно изучите содержимое стандартных include-файлов в своей системе!

В качестве резюме - схема, поясняющая "превращения" Си-программы из текста на языке программирования в выполняемый код: все файлы **.c** могут использовать общие include-файлы; их подстановку в текст, а также обработку **#define** произведет препроцессор **cpp**



1.147. Напоследок - простой, но жизненно важный совет. Если вы пишете программу, которую вставите в систему для частого использования, поместите в исходный текст этой программы идентификационную строку наподобие

```
static char id[] = "This is /usr/abs/mybin/xprogram";
```

---

†† Обратите внимание, что библиотечные функции не являются частью ЯЗЫКА Си как такового. То, что в других языках (**PL/1**, **Algol-68**, **Pascal**) является частью языка (встроено в язык)- в Си вынесено на уровень библиотек. Например, в Си нет *оператора* вывода; функция вывода **printf** - это библиотечная *функция* (хотя и общепринятая). Таким образом мощь языка Си состоит именно в том, что он позволяет использовать функции, написанные другими программистами и даже на других языках, т.е. является функционально расширяемым.

Тогда в случае аварии в файловой системе, если вдруг ваш файл "потеряется" (то есть у него пропадет имя - например из-за порчи каталога), то он будет найден программой проверки файловой системы - **fsck** - и помещен в каталог **/lost+found** под специальным кодовым именем, ничего общего не имеющим со старым. Чтобы понять, что это был за файл и во что его следует переименовать (чтобы восстановить правильное имя), мы применим команду

```
strings имя_файла
```

Эта команда покажет все длинные строки из печатных символов, содержащиеся в данном файле, в частности и нашу строку *id*[]. Увидев ее, мы сразу поймем, что файл надо переименовать так:

```
mv имя_файла /usr/abs/mybin/xprogram
```

1.148. Где размещать include-файлы и как программа узнает, где же они лежат? Стандартные системные include-файлы размещены в */usr/include* и подкаталогах. Если мы пишем некую свою программу (проект) и используем директивы

```
#include "имяФайла.h"
```

то обычно include-файлы *имяФайла.h* лежат в текущем каталоге (там же, где и файлы с программой на Си). Однако мы можем помещать ВСЕ наши include-файлы в одно место (скажем, известное группе программистов, работающих над одним и тем же проектом). Хорошее место для всех ваших личных include-файлов - каталог (вами созданный)

```
$HOME/include
```

где **$HOME** - ваш домашний каталог. Хорошее место для общих include-файлов - каталог

```
/usr/local/include
```

Как сказать компилятору, что **#include ""** файлы надо брать из определенного места, а не из текущего каталога? Это делает ключ компилятора

```
cc -Iимя_каталога ...
```

Например:

```
/* Файл x.c */
#include "x.h"

int main(int ac, char *av[]){
        ....
        return 0;
}
```

И файл *x.h* находится в каталоге */home/abs/include/x.h* (*/home/abs* - мой домашний каталог). Запуск программы на компиляцию выглядит так:

```
cc -I/home/abs/include -O x.c -o x
        или
cc -I$HOME/include -O x.c -o x
```

Или, если моя программа *x.c* находится в */home/abs/progs*

```
cc -I../include -O x.c -o x
```

Ключ **-O** задает вызов компилятора с оптимизацией.

Ключ **-I** оказывает влияние и на **#include** <> директивы тоже. Для ОС **Solaris** на машинах **Sun** программы для оконной системы **X Window System** содержат строки вроде

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
```

На **Sun** эти файлы находятся не в */usr/include/X11*, а в */usr/openwin/include/X11*. Поэтому запуск на компиляцию оконных программ на **Sun** выглядит так:

```
cc -O -I/usr/openwin/include xprogram.c \
      -o xprogram -L/usr/openwin/lib -lX11
```

где **-l***X11* задает подключение графической оконной библиотеки **Xlib**.

Если include-файлы находятся во многих каталогах, то можно задать поиск в нескольких каталогах, к примеру:

```
cc -I/usr/openwin/include -I/usr/local/include -I$HOME/include ...
```

2. Массивы, строки, указатели.

Массив представляет собой агрегат из нескольких переменных одного и того же типа. Массив с именем *a* из *LENGTH* элементов типа **TYPE** объявляется так:

```
TYPE a[LENGTH];
```

Это соответствует тому, что объявляются переменные типа **TYPE** со специальными именами *a*[0], *a*[1], ..., *a*[*LENGTH-1*]. Каждый элемент массива имеет свой номер - *индекс*. Доступ к *x*-ому элементу массива осуществляется при помощи операции индексации:

```
int x = ... ;       /* целочисленный индекс   */
TYPE value = a[x]; /* чтение x-ого элемента  */
     a[x] = value; /* запись в x-тый элемент */
```

В качестве индекса может использоваться любое выражение, выдающее значение *целого* типа: **char**, **short**, **int**, **long**. Индексы элементов массива в Си начинаются с **0** (а не с 1), и индекс последнего элемента массива из *LENGTH* элементов - это *LENGTH*-1 (а не *LENGTH*). Поэтому цикл по всем элементам массива - это

```
TYPE a[LENGTH]; int indx;
for(indx=0; indx < LENGTH; indx++)
   ...a[indx]...;
```

indx < *LENGTH* равнозначно *indx* <= *LENGTH*-1. Выход за границы массива (попытка чтения/записи несуществующего элемента) может привести к непредсказуемым результатам и поведению программы. Отметим, что это одна из самых распространенных ошибок.

Статические массивы можно объявлять с *инициализацией*, перечисляя значения их элементов в {} через запятую. Если задано меньше элементов, чем длина массива - остальные элементы считаются нулями:

```
int a10[10] = { 1, 2, 3, 4 }; /* и 6 нулей */
```

Если при описании массива с инициализацией не указать его размер, он будет подсчитан компилятором:

```
int a3[] = { 1, 2, 3 }; /* как бы a3[3] */
```

В большинстве современных компьютеров (с фон-Неймановской архитектурой) память 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 (адрес может иметь и более сложную структуру, чем просто целое число - например состоять из номера *сегмента* памяти и номера байта в этом сегменте). В Си адрес переменной можно получить с помощью операции взятия адреса **&**. Пусть у нас есть переменная *var*, тогда **&***var* - ее адрес. Адрес *нельзя* присваивать целой переменной; для хранения адресов используются указатели (смотри ниже).

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 - при помощи **sizeof**(*var*). Всегда выполняется **sizeof**(**char**)==1. В некоторых машинах адреса переменных (а также агрегатов данных - массивов и структур) кратны **sizeof**(*int*) или **sizeof**(*double*) - это так называемое "выравнивание (*align*ment) данных на границу типа **int**". Это позволяет делать доступ к данным более быстрым (аппаратура работает эффективнее).

Язык Си предоставляет нам средство для работы с адресами данных - *указатели* (*pointer*)†. Указатель физически - это адрес некоторой переменной ("указуемой" переменной). Отличие указателей от машинных адресов состоит в том, что указатель может содержать адреса данных только *определенного типа*. Указатель *ptr*, который может указывать на данные типа **TYPE**, описывается так:

```
TYPE  var;      /* переменная       */
TYPE *ptr;      /* объявление ук-ля */
      ptr = & var;
```

В данном случае мы занесли в указательную переменную *ptr* адрес переменной *var*. Будем говорить, что указатель *ptr* указывает на переменную *var* (или, что *ptr* установлен на *var*). Пусть **TYPE** равно **int**, и у нас есть массив и указатели:

```
int  array[LENGTH], value;
int *ptr, *ptr1;
```

Установим указатель на *x*-ый элемент массива

---

† В данной книге слова "указатель" и "ссылка" употребляются в одном и том же смысле. Если вы обратитесь к языку **Си++**, то обнаружите, что там эти два термина (*pointer* и *reference*) означают *разные* понятия (хотя и сходные).

```
ptr = & array[x];
```

Указателю можно присвоить значение другого указателя *на такой же тип*. В результате оба указателя будут указывать на одно и то же место в памяти: *ptr1 = ptr*;

Мы можем изменять указуемую переменную при помощи операции *****

```
*ptr = 128;   /* занести 128 в указуемую перем. */
value = *ptr; /* прочесть указуемую переменную  */
```

В данном случае мы заносим и затем читаем значение переменной *array*[*x*], на которую поставлен указатель, то есть

```
*ptr  означает сейчас  array[x]
```

Таким образом, операция ***** (значение по адресу) оказывается обратной к операции **&** (взятие адреса):

```
& (*ptr) == ptr    и    * (&value) == value
```

Операция ***** объясняет смысл описания **TYPE ****ptr*; оно означает, что значение выражения ******ptr* будет иметь тип **TYPE**. Название же типа самого указателя - это **(TYPE *)**. В частности, **TYPE** может сам быть указательным типом - можно объявить указатель на указатель, вроде **char** ***ptrptr*;

Имя массива - это константа, представляющая собой указатель на 0-ой элемент массива. Этот указатель отличается от обычных тем, что его нельзя изменить (установить на другую переменную), поскольку он сам хранится не в переменной, а является просто некоторым постоянным адресом.

```
    массив          указатель
      ____________      _____
array: | array[0] |   ptr:| * |
       | array[1] |        |
       | array[2] |<--------- сейчас равен &array[2]
       |  ...     |
```

Следствием такой интерпретации имен массивов является то, что для того чтобы поставить указатель на начало массива, надо писать

```
ptr = array;  или  ptr = &array[0];
       но не
ptr = &array;
```

Операция **&** перед одиноким именем массива не нужна и недопустима!

Такое родство указателей и массивов позволяет нам применять операцию ***** к имени массива: *value* = ******array*; означает то же самое, что и *value* = *array*[0];

Указатели - не целые числа! Хотя физически это и *номера* байтов, адресная арифметика отличается от обычной. Так, если дан указатель **TYPE** **ptr*; и номер байта (адрес), на который указывает *ptr*, равен *byteaddr*, то

```
ptr = ptr + n; /* n - целое, может быть и < 0 */
```

заставит *ptr* указывать не на байт номер *byteaddr* + *n*, а на байт номер

```
byteaddr + (n * sizeof(TYPE))
```

то есть прибавление единицы к указателю продвигает адрес не на 1 байт, а на размер указываемого указателем типа данных! Пусть указатель *ptr* указывает на *x*-ый элемент массива *array*. Тогда после

```
TYPE *ptr2 = array + L;  /* L - целое */
TYPE *ptr1 = ptr   + N;  /* N - целое */
      ptr += M;          /* M - целое */
```

указатели указывают на

```
ptr1 == &array[x+N]   и   ptr  == &array[x+M]
ptr2 == &array[L]
```

Если мы теперь рассмотрим цепочку равенств

```
*ptr2 = *(array + L) = *(&array[L]) =
          array[L]
```

то получим

ОСНОВНОЕ ПРАВИЛО: пусть *ptr* - указатель или имя массива. Тогда операции индексации, взятия значения по адресу, взятия адреса и прибавления целого к указателю связаны соотношениями:

```
 ptr[x]  тождественно *(ptr+x)
&ptr[x]  тождественно  ptr+x
```

(тождества верны в обе стороны), в том числе при *x*==0 и *x* < 0. Так что, например,

```
ptr[-1] означает  *(ptr-1)
ptr[0]  означает  *ptr
```

Указатели можно индексировать подобно массивам. Рассмотрим пример:

```
        /* индекс:     0    1    2    3    4   */
double  numbers[5] = { 0.0, 1.0, 2.0, 3.0, 4.0 };
double *dptr   = &numbers[2];
double  number =  dptr[2];  /* равно 4.0 */

numbers: [0]   [1]   [2]   [3]   [4]
                      |
        [-2]  [-1]   [0]   [1]   [2]
                     dptr
```

поскольку

```
если dptr    = &numbers[x] = numbers + x
то   dptr[i] = *(dptr + i) =
             = *(numbers + x + i) = numbers[x + i]
```

Указатель на один тип можно преобразовать в указатель на другой тип: такое преобразование не вызывает генерации каких-либо машинных команд, но заставляет компилятор изменить параметры адресной арифметики, а также операции выборки данного по указателю (собственно, разница в указателях на данные разных типов состоит только в *размерах* указуемых типов; а также в генерации команд `->' для выборки полей *структур*, если указатель - на структурный тип).

Целые (**int** или **long**) числа иногда можно преобразовывать в указатели. Этим пользуются при написании драйверов устройств для доступа к регистрам по физическим адресам, например:

```
unsigned short *KISA5 = (unsigned short *) 0172352;
```

Здесь возникают два тонких момента:

1. Как уже было сказано, адреса данных часто выравниваются на границу некоторого типа. Мы же можем задать невыровненное целое значение. Такой адрес будет некорректен.

2. Структура адреса, поддерживаемая процессором, может не соответствовать формату целых (или длинных целых) чисел. Так обстоит дело с **IBM PC 8086/80286**, где адрес состоит из пары **short int** чисел, хранящихся в памяти подряд. Однако *весь* адрес (если рассматривать эти два числа как одно длинное целое) не является обычным **long**-числом, а вычисляется более сложным способом: адресная пара *SEGMENT*:*OFFSET* преобразуется так

   ```
   unsigned short SEGMENT, OFFSET; /*16 бит: [0..65535]*/
   unsigned long  ADDRESS = (SEGMENT << 4) + OFFSET;
     получается 20-и битный физический адрес ADDRESS
   ```

   Более того, на машинах с *диспетчером памяти*, адрес, хранимый в указателе, является "виртуальным" (т.е. воображаемым, ненастоящим) и может не совпадать с физическим адресом, по которому данные хранятся в памяти компьютера. В памяти может одновременно находиться несколько программ, в каждой из них будет *своя* система адресации ("*адресное пространство*"), отсчитывающая виртуальные адреса с нуля от начала области памяти, выделенной данной программе. Преобразование виртуальных адресов в физические выполняется аппаратно.

В Си принято соглашение, что указатель (*TYPE* *)**0** означает "указатель ни на что". Он является просто признаком, используемым для обозначения несуществующего адреса или конца цепочки указателей, и имеет специальное обозначение **NULL**. Обращение (выборка или запись данных) по этому указателю считается некорректным (кроме случая, когда вы пишете машинно-зависимую программу и работаете с *физическими* адресами).

Отметим, что указатель можно направить в неправильное место - на участок памяти, содержащий данные не того типа, который задан в описании указателя; либо вообще содержащий неизвестно что:

```
int i = 2, *iptr = &i;
double x = 12.76;
  iptr += 7;  /* куда же он указал ?! */
  iptr = (int *) &x;  i = *iptr;
```

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

При передаче имени массива в качестве параметра функции, как аргумент передается не копия САМОГО МАССИВА (это заняло бы слишком много места), а копия АДРЕСА 0-ого элемента этого массива

(т.е. указатель на начало массива).

```
f(int x   ){ x++;     }
g(int xa[]){ xa[0]++; }
int a[2] = { 1, 1 }; /* объявление с инициализацией */
main(){
 f(a[0]); printf("%d\n",a[0]); /* a[0] осталось равно 1*/
 g(a   ); printf("%d\n",a[0]); /* a[0] стало равно 2   */
}
```

В **f**() в качестве аргумента передается копия элемента *a*[0] (и изменение этой копии не приводит к изменению самого массива - аргумент *x* является локальной переменной в **f**()), а в **g**() таким локалом является АДРЕС массива *a* - но не сам массив, поэтому *xa*[0]++ изменяет сам массив *a* (зато, например, *xa*++ внутри **g**() изменило бы лишь локальную указательную переменную *xa*, но не адрес массива *a*).

Заметьте, что 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:

```
вместо    Fun(int xa[5]) { ... }
можно     Fun(int xa[] ) { ... }
или даже  Fun(int *xa  ) { ... }
```

Если функция должна знать длину массива - передавайте ее как дополнительный аргумент:

```
 int sum( int a[], int len ){
   int s=0, i;
   for(i=0; i < len; i++) s += a[i];
   return( s );
 }
 ... int arr[10] = { ... };
 ... int sum10 = sum(arr, 10); ...
```

Количество элементов в массиве **TYPE** *arr*[*N*]; можно вычислить специальным образом, как

```
#define LENGTH (sizeof(arr) / sizeof(arr[0]))
        или
#define LENGTH (sizeof(arr) / sizeof(TYPE))
```

Оба способа выдадут число, равное *N*. Эти конструкции обычно употребляются для вычисления длины массивов, задаваемых в виде

```
TYPE arr[] = { ....... };
```

без явного указания размера. **sizeof**(*arr*) выдает размер всего массива в байтах. **sizeof**(*arr*[0]) выдает размер одного элемента. И все это не зависит от типа элемента (просто потому, что все элементы массивов имеют *одинаковый* размер).

Строка в Си - это последовательность байт (букв, символов, литер, *character*), завершающаяся в конце специальным признаком - байтом '**0**'. Этот признак добавляется компилятором автоматически, когда мы задаем строку в виде "*строка*". Длина строки (т.е. число литер, 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*. При передаче строки в качестве аргумента в функцию, функции не требуется знать длину строки, т.к. передается указатель на *начало* массива, а наличие ограничителя '\0' позволяет обнаружить *конец* строки при ее просмотре.

С массивами байт можно использовать следующую конструкцию, задающую массивы (строки) одинакового размера:

```
char stringA [ITSSIZE];
char stringB [sizeof stringA];
```

В данном разделе мы в основном будем рассматривать строки и указатели на символы.

2.1. Операции взятия адреса объекта и разыменования указателя - взаимно обратны.

```
TYPE  objx;
TYPE *ptrx = &objx;  /* инициализируем адресом objx */

*(&objx) = objx;
&(*ptrx) = ptrx;
```

Вот пример того, как можно заменить условный оператор условным выражением (это удастся не всегда):

```
if(c) a = 1;
else  b = 1;
```

Предупреждение: такой стиль **не** способствует понятности программы и даже компактности ее кода.

```
#include <stdio.h>
int main(int ac, char *av[]){
        int a, b, c;

        a = b = c = 0;
        if(av[1])  c = atoi(av[1]);

        *(c ? &a : &b) = 1;     /* !!! */

        printf("cond=%d a=%d b=%d\n", c, a, b);
        return 0;
}
```

2.2. Каким образом инициализируются по умолчанию внешние и статические массивы? Инициализируются ли по умолчанию автоматические массивы? Каким образом можно присваивать значения элементам массива, относящегося к любому классу памяти?

2.3. Пусть задан массив int *arr*[10]; что тогда означают выражения:

```
 arr[0]        *arr            *arr + 2
 arr[2]        *(arr + 2)       arr
&arr[2]         arr+2
```

2.4. Правильно ли написано увеличение величины, на которую указывает указатель *a*, на единицу?

```
*a++;
```

Ответ: нет, надо:

```
(*a)++;   или    *a += 1;
```

2.5. Дан фрагмент текста:

```
char a[] = "xyz";
char *b  = a + 1;
```

Чему равны

```
b[-1]       b[2]      "abcd"[3]
```

(Ответ: 'x', '\0', 'd')

Можно ли написать *a*++ ? То же про *b*++ ? Можно ли написать *b*=*a* ? *a*=*b* ? (нет, да, да, нет)

2.6. Ниже приведена программа, вычисляющая среднее значение элементов массива

```
int arr [] = {1, 7, 4, 45, 31, 20, 57, 11};
main () {
   int i; long sum;

   for ( i = 0, sum = 0L;
         i < (sizeof(arr)/sizeof(int)); i++ )
              sum += arr[i];
   printf ("Среднее значение = %ld\n", sum/8)
}
```

Перепишите указанную программу с применением указателей.

2.7. Что напечатается в результате работы программы?

```
char arr[] = {'С', 'Л', 'А', 'В', 'А'};
main () {
   char *pt; int i;

   pt = arr + sizeof(arr) - 1;
   for( i = 0; i < 5; i++, pt--  )
```

```
                printf("%c %c\n", arr[i], *pt);
        }
```

Почему массив *arr*[] описан вне функции **main**()? Как внести его в функцию **main**() ? Ответ: написать внутри **main**

static char *arr*[]=...

2.8. Можно ли писать на Си так:

```
f( n, m ){
        int x[n]; int y[n*2];
        int z[n * m];
        ...
}
```

Ответ: к сожалению нельзя (**Си** - это не **Algol**). При отведении памяти для массива в качестве размера должна быть указана *константа* или выражение, которое может быть еще во время компиляции вычислено до целочисленной константы, т.е. массивы имеют *фиксированную* длину.

2.9. Предположим, что у нас есть описание массива

```
static int mas[30][100];
```

a) выразите адрес mas[22][56] иначе

b) выразите адрес mas[22][0] двумя способами

c) выразите адрес mas[0][0] тремя способами

2.10. Составьте программу инициализации двумерного массива a[10][10], выборки элементов с a[5][5] до a[9][9] и их распечатки. Используйте доступ к элементам по указателю.

2.11. Составьте функцию вычисления скалярного произведения двух векторов. Длина векторов задается в качестве одного из аргументов.

2.12. Составьте функцию умножения двумерных матриц **a**[][] * **b**[][].

2.13. Составьте функцию умножения трехмерных матриц **a**[][][] * **b**[][][].

2.14. Для тех, кто программировал на языке **Pascal**: какая допущена ошибка?

```
char a[10][20];
char c;
int x,y;
  ...
c = a[x,y];
```

Ответ: многомерные массивы в Си надо индексировать так:

```
c = a[x][y];
```

В написанном же примере мы имеем в качестве индекса выражение *x*,*y* (оператор "запятая") со значением *y*, т.е.

```
c = a[y];
```

Синтаксической ошибки нет, но смысл совершенно изменился!

2.15. Двумерные массивы в памяти представляются как одномерные. Например, если

```
int a[N][M];
```

то конструкция *a*[*y*][*x*] превращается при компиляции в одномерную конструкцию, подобную такой:

```
int a[N * M]; /* массив развернут построчно */
#define a_yx(y, x)    a[(x) + (y) * M]
```

то есть

```
a[y][x] есть *(&a[0][0] + y * M + x)
```

Следствием этого является то, что компилятор для генерации индексации двумерных (и более) массовов должен знать M - размер массива по 2-ому измерению (а также 3-ему, 4-ому, и.т.д.). В частности, при передаче многомерного массива в функцию

```
f(arr) int arr[N][M]; { ... }   /* годится     */
f(arr) int arr[] [M]; { ... }   /* годится     */
f(arr) int arr[] [];  { ... }   /* не годится */

f(arr) int (*arr)[M]; { ... }   /* годится     */
f(arr) int  *arr [M]; { ... }   /* не годится:
             это уже не двумерный массив,
             а одномерный массив указателей  */
```

А также при описании внешних массивов:

```
extern int a[N][M];     /* годится */
extern int a[ ][M];     /* годится */
extern int a[ ][ ];     /* не годится: компилятор
          не сможет сгенерить операцию индексации */
```

Вот как, к примеру, должна выглядеть работа с двумерным массивом arr[ROWS][COLS], отведенным при помощи **malloc**();

```
void f(int array[][COLS]){
        int x, y;
        for(y=0; y < ROWS; y++)
            for(x=0; x < COLS; x++)
                array[y][x] = 1;
}
void main(){
        int *ptr = (int *) malloc(sizeof(int) * ROWS * COLS);
        f( (int (*) [COLS]) ptr);
}
```

2.16. Как описывать ссылки (указатели) на двумерные массивы? Рассмотрим такую программу:

```
#include <stdio.h>
#define First  3
#define Second 5

char arr[First][Second] = {
        "ABC.",
        { 'D', 'E', 'F', '?', '\0' },
        { 'G', 'H', 'Z', '!', '\0' }
};

char (*ptr)[Second];

main(){
        int i;

        ptr = arr;      /* arr и ptr теперь взаимозаменимы */
        for(i=0; i < First; i++)
                printf("%s\t%s\t%c\n", arr[i], ptr[i], ptr[i][2]);
}
```

Указателем здесь является *ptr*. Отметим, что у него задана размерность по второму измерению: *Second*, именно для того, чтобы компилятор мог правильно вычислить двумерные индексы.

Попробуйте сами объявить

```
char (*ptr)[4];
char (*ptr)[6];
char **ptr;
```

и увидеть, к каким невеселым эффектам это приведет (компилятор, кстати, будет ругаться; но есть вероятность, что он все же транслирует это для вас. Но работать оно будет плачевно). Попробуйте также использовать ptr[x][y].

Обратите также внимание на инициализацию строк в нашем примере. Строка "ABC." равносильна объявлению

```
        { 'A', 'B', 'C', '.', '\0' },
```

2.17. Массив *s* моделирует двумерный массив char *s*[*H*][*W*]; Перепишите пример при помощи указателей, избавьтесь от операции умножения. Прямоугольник (*x0*,*y0*,*width*,*height*) лежит целиком внутри (*0*,*0*,*W*,*H*).

```
char s[W*H]; int x,y; int x0,y0,width,height;
for(x=0; x < W*H; x++) s[x] = '.';
     ...
for(y=y0; y < y0+height; y++)
  for(x=x0; x < x0+width; x++)
      s[x + W*y] = '*';
```

Ответ:

```
char s[W*H]; int i,j; int x0,y0,width,height;
char *curs;
     ...
for(curs = s + x0 + W*y0, i=0;
    i < height; i++, curs += W-width)
  for(j=0; j < width; j++)
       *curs++ = '*';
```

Такая оптимизация возможна в некоторых функциях из главы "Работа с видеопамятью".

2.18. Что означают описания?

```
int i;            // целое.
int *pi;          // указатель на целое.
int *api[3];      // массив из 3х ук-лей на целые.
int (*pai)[3];    // указатель на массив из 3х целых.
                  // можно описать как    int **pai;
int fi();         // функция, возвращающая целое.
int *fpi();       // ф-ция, возвр. ук-ль на целое.
int (*pfi)();     // ук-ль на ф-цию, возвращающую целое.
int *(*pfpi)();   // ук-ль на ф-цию, возвр. ук-ль на int.
int (*pfpfi())(); // ф-ция, возвращающая указатель на
                  // "функцию, возвращающую целое".
int (*fai())[3];  // ф-ция, возвр. ук-ль на массив
                  // из 3х целых. иначе ее
                  // можно описать как    int **fai();
int (*apfi[3])(); // массив из 3х ук-лей на функции,
                  // возвращающие целые.
```

Переменные в Си описываются в формате их использования. Так описание

```
int (*f)();
```

означает, что f можно использовать в виде

```
int value;
value = (*f)(1, 2, 3 /* список аргументов */);
```

Однако из такого способа описания тип самой описываемой переменной и его смысл довольно неочевидны. Приведем прием (позаимствованный из журнала "*Communications of the ACM*"), позволяющий прояснить смысл описания. Описание на Си переводится в описание в стиле языка **Algol-68**. Далее

```
ref      ТИП    означает  "указатель на ТИП"
proc()   ТИП              "функция, возвращающая ТИП"
array of ТИП              "массив из элементов ТИПа"
x:       ТИП              "x имеет тип ТИП"
```

Приведем несколько примеров, из которых ясен и способ преобразования:

```
int (*f())();     означает
        (*f())()  :                    int
         *f()     :              proc() int
          f()     :         ref proc() int
          f       :  proc() ref proc() int
```

то есть **f** - функция, возвращающая указатель на функцию, возвращающую целое.

```
int (*f[3])();    означает
        (*f[])()  :                      int
         *f[]     :               proc() int
          f[]     :          ref proc() int
          f       :  array of ref proc() int
```

f - массив указателей на функции, возвращающие целые. Обратно: опишем **g** как указатель на функцию, возвращающую указатель на массив из 5и указателей на функции, возвращающие указатели на целые.

```
       g          : ref p() ref array of ref p() ref int
      *g          :     p() ref array of ref p() ref int
     (*g)()       :         ref array of ref p() ref int
    *(*g)()       :             array of ref p() ref int
   (*(*g)())[5]   :                      ref p() ref int
  *(*(*g)())[5]   :                          p() ref int
 (*(*(*g)())[5])():                              ref int
*(*(*(*g)())[5])():                                  int
                     int *(*(*(*g)())[5])();
```

В Си невозможны функции, возвращающие массив:

```
proc() array of ...
        а только
proc() ref array of ...
```

Само название типа (например, для использования в операции приведения типа) получается вычеркиванием имени переменной (а также можно опустить размер массива):

```
g = ( int *(*(*(*)())[])() ) 0;
```

2.19. Напишите функцию **strcat**(*d*,*s*), приписывающую строку *s* к концу строки *d*.

Ответ:

```
char *strcat(d,s) register char *d, *s;
{  while( *d ) d++;      /* ищем конец строки d */
   while( *d++ = *s++ ); /* strcpy(d, s)        */
   return (d-1);         /* конец строки        */
}
```

Цикл, помеченный "**strcpy**" - это наиболее краткая запись операторов

```
   do{ char c;
       c = (*d = *s); s++; d++;
   } while(c != '\0');
```

На самом деле **strcat** должен по стандарту возвращать свой первый аргумент, как и функция **strcpy**:

```
char *strcat(d,s) register char *d, *s;
{  char *p = d;
   while( *d ) d++;
   strcpy(d, s); return p;
}
```

Эти два варианта демонстрируют, что функция может быть реализована разными способами. Кроме того видно, что вместо стандартной библиотечной функции мы можем определить *свою* одноименную функцию, несколько отличающуюся поведением от стандартной (как возвращаемое значение в 1-ом варианте).

2.20. Напишите программу, которая объединяет и распечатывает две строки, введенные с терминала. Для ввода строк используйте функцию **gets**(), а для их объединения - **strcat**(). В другом варианте используйте **sprintf**(*result*,"**%s%s**",*s1*,*s2*);

2.21. Модифицируйте предыдущую программу таким образом, чтобы она выдавала длину (число символов) объединенной строки. Используйте функцию **strlen**(). Приведем несколько версий реализации **strlen**:

```
/* При помощи индексации массива */
int strlen(s) char s[];
{   int length = 0;
    for(; s[length] != '\0'; length++);
    return (length);
}
/* При помощи продвижения указателя */
int strlen(s) char *s;
```

```
{   int length;
    for(length=0; *s; length++, s++);
    return length;
}
/* При помощи разности указателей */
int strlen(register char *s)
{   register char *p = s;
    while(*p) p++;   /* ищет конец строки */
    return (p - s);
}
```

Разность двух указателей на один и тот же тип - целое число:

```
если TYPE *p1, *p2;
то  p2 - p1 = целое число штук TYPE
              лежащих между p2 и p1
если p2 = p1 + n
то   p2 - p1 = n
```

Эта разность может быть и отрицательной если *p2* < *p1*, то есть *p2* указывает на более левый элемент массива.

2.22. Напишите оператор Си, который обрубает строку *s* до длины *n* букв. Ответ:

```
if( strlen(s) > n )
    s[n] = '\0';
```

Первое сравнение вообще говоря излишне. Оно написано лишь на тот случай, если строка *s* короче, чем *n* букв и хранится в массиве, который также короче *n*, т.е. не имеет *n*-ого элемента (поэтому в него нельзя производить запись признака конца).

2.23. Напишите функции преобразования строки, содержащей изображение целого числа, в само это число. В двух разных вариантах аргумент-адрес должен указывать на первый байт строки; на последний байт. Ответ:

```
#define isdigit(c) ('0' <= (c) && (c) <= '9')

int atoi(s) register char *s;
{   register int res=0, neg=0;
    for(;;s++){
        switch(*s){
        case ' ': case '\t': continue;
        case '-':            neg++;
        case '+':            s++;
        } break;
    }
    while(isdigit(*s))
        res = res * 10  +  *s++ - '0';
    return( neg ? -res : res );
}

int backatoi(s) register char *s;
{   int res=0, pow=1;
    while(isdigit(*s)){
        res += (*s-- - '0') * pow;
        pow *= 10;
    }
    if(*s == '-') res = -res;
    return res;
}
```

2.24. Можно ли для занесения в массив *s* строки "hello" написать

```
char s[6]; s = "hello";
     или
char s[6], d[] = "hello"; s = d;
```

Ответ: нет. Массивы в Си нельзя присваивать целиком. Для пересылки массива байт надо использовать функцию **strcpy**(*s*,*d*). Здесь же мы пытаемся изменить **адрес** *s* (имя массива - это адрес начала памяти, выделенной для хранения массива), сделав его равным адресу безымянной строки "hello" (или массива *d* во

втором случае). Этот адрес является *константой* и не может быть изменен!

Заметим однако, что *описание* массива с инициализацией вполне допустимо:

```
char s[6] = "hello";
        или
char s[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
        или
char s[] = "hello";
        или
char s[] = { "hello" };
```

В этом случае компилятор резервирует память для хранения массива и расписывает ее байтами начального значения. Обратите внимание, что строка в двойных кавычках (если ее рассматривать как *массив* букв) имеет длину на единицу больше, чем написано букв в строке, поскольку в конце массива находится символ '**\0**' - признак конца, добавленный компилятором. Если бы мы написали

```
char s[5] = "hello";
```

то компилятор сообщил бы об ошибке, поскольку длины массива (5) недостаточно, чтобы разместить 6 байт. В третьей строке примера написано *s*[], чтобы компилятор *сам* посчитал необходимую длину массива.

Наконец, возможна ситуация, когда массив больше, чем хранящаяся в нем строка. Тогда "лишнее" место содержит какой-то мусор (в static-памяти изначально - байты \0).

```
char s[12] = "hello";
содержит:     h e l l o \0 ? ? ? ? ? ?
```

В программах текстовой обработки под "длиной строки" обычно понимают количество букв в строке НЕ считая закрывающий байт '**\0**'. Именно такую длину считает стандартная функция **strlen**(*s*). Поэтому следует различать такие понятия как "*(текущая) длина строки*" и "*длина массива*, *в котором хранится строка*": **sizeof**(*s*). Для написанного выше примера эти значения равны соответственно 5 и 12.

Следует также отличать массивы от указателей:

```
char *sp = "bye bye";
sp = "hello";
```

будет вполне законно, поскольку в данном случае *sp* - не имя массива (т.е. константа, равная адресу начала массива), а *указатель* (переменная, хранящая адрес некоторой области памяти). Поскольку указатель - это переменная, то ее значение изменять можно: в данном случае *sp* сначала содержала адрес безымянного массива, в котором находится "bye bye"; затем мы занесли в *sp* адрес безымянного массива, хранящего строку "hello". Здесь не происходит копирования массива, а происходит просто присваивание переменной *sp* нового значения адреса.

Предостережем от возможной неприятности:

```
char d[5]; char s[] = "abcdefgh";
strcpy(d, s);
```

Длины массива *d* просто не хватит для хранения такой длинной строки. Поскольку это ничем не контролируется (ни компилятором, ни самой **strcpy**, ни вами явным образом), то при копировании строки "избыточные" байты запишутся *после* массива *d* поверх других данных, которые будут испорчены. Это приведет к непредсказуемым эффектам.

Некоторые возможности для контроля за длиной строк-аргументов вам дают функции **strncpy**(*d*,*s*,*len*); **strncat**(*d*,*s*,*len*); **strncmp**(*s1*,*s2*,*len*). Они пересылают (сравнивают) не более, чем *len* первых символов строки *s* (строк *s1*, *s2*). Посмотрите в документацию! Напишите функцию **strncmp** (сравнение строк по первым *len* символам), посмотрев на функцию **strncpy**:

```
char *strncpy(dst, src, n)
     register char *dst, *src;
     register int n;
{    char *save;
     for(save=dst; --n >= 0; )
         if( !(*dst++ = *src++)){
             while(--n >= 0)
                *dst++ = '\0';
             return save;
         }
     return save;
}
```

Отметьте, что **strncpy** обладает одним неприятным свойством: если *n* <= **strlen**(*src*), то строка *dst* не будет

иметь на конце символа '\0', то есть будет находиться в некорректном (не каноническом) состоянии.

Ответ:

```
int strncmp(register char *s1, register char *s2, register int n)
{
        if(s1 == s2)
                return(0);
        while(--n >= 0 && *s1 == *s2++)
                if(*s1++ == '\0')
                        return(0);
        return((n < 0)? 0: (*s1 - *--s2));
}
```

2.25. В чем ошибка?

```
#include <stdio.h>  /* для putchar */
char s[] = "We don't need no education";
main(){ while(*s) putchar(*s++); }
```

Ответ: здесь *s* - константа, к ней неприменима операция ++. Надо написать

```
char *s = "We don't need no education";
```

сделав *s* указателем на безымянный массив. Указатель уже можно изменять.

2.26. Какие из приведенных конструкций обозначают одно и то же?

```
char a[]  = "";          /* пустая строка */
char b[]  = "\0";
char c    = '\0';
char z[]  = "ab";
char aa[] = { '\0' };
char bb[] = { '\0', '\0' };
char xx[] = { 'a', 'b' };
char zz[] = { 'a', 'b', '\0' };
char *ptr = "ab";
```

2.27. Найдите ошибки в описании символьной строки:

```
main() {
   char mas[] = {'s', 'o', 'r', 't'};  /* "sort" ? */
   printf("%s\n", mas);
}
```

Ответ: строка должна кончаться '\0' (в нашем случае **printf** не обнаружив символа конца строки будет выдавать и байты, находящиеся в памяти *после* массива *mas*, т.е. мусор); инициализированный массив не может быть автоматическим - требуется **static**:

```
main() {
   static char mas[] = {'s', 'o', 'r', 't', '\0'};
}
```

Заметим, что

```
main(){    char *mas = "sort";   }
```

законно, т.к. сама строка здесь хранится в статической памяти, а инициализируется лишь указатель на этот массив байт.

2.28. В чем ошибка? Программа собирается из двух файлов: *a.c* и *b.c* командой

```
        cc a.c b.c -o ab
a.c                          b.c
-------------------------------------------------
int n = 2;                   extern int n;
char s[] = "012345678";      extern char *s;
main(){                      f(){
  f();                           s[n] = '+';
  printf("%s\n", s );        }
}
```

Ответ: дело в том, что типы (**char ***) - указатель, и **char**[**]** - массив, означают одно и то же только при

объявлении формального параметра функции:

```
f(char *arg){...}     f(char arg[]){...}
```

это будет *локальная переменная*, содержащая указатель на **char** (т.е. адрес некоторого байта в памяти). Внутри функции мы можем *изменять* эту переменную, например *arg*++. Далее, и (**char ***) и **char**[] одинаково *используются*, например, оба эти типа можно индексировать: *arg*[*i*]. Но вне функций они *объявляют* разные объекты! Так **char** **p*; это скалярная переменная, хранящая адрес (указатель):

```
  --------      -------
p:|   *--|----->| '0' | char
  --------      | '1' | char
                ...
```

тогда как **char** *a*[20]; это адрес начала массива (а вовсе не переменная):

```
                -------
              a:| '0' | char
                | '1' | char
                ...
```

В нашем примере в файле *b.c* мы объявили внешний массив *s* как переменную. В результате компилятор будет интерпретировать начало массива *s* как *переменную*, содержащую указатель на **char**.

```
                -------
              s:| '0' |   \  это будет воспринято как
                | '1' |   /  адрес других данных.
                | '2' |
                ...
```

И индексироваться будет уже ЭТОТ адрес! Результат - обращение по несуществующему адресу. То, что написано у нас, эквивалентно

```
char s[]  = "012345678";
char **ss = s;      /* s - как бы "массив указателей"  */
     /* первые байты s интерпретируются как указатель: */
char  *p  = ss[0];
     p[2] = '+';
```

Мы же должны были объявить в *b.c*

```
extern char s[];  /* размер указывать не требуется */
```

Вот еще один аналогичный пример, который пояснит вам, что происходит (а заодно покажет порядок байтов в long). Пример выполнялся на **IBM PC 80386**, на которой

```
       sizeof(char *) = sizeof(long) = 4
```

```
a.c                        b.c
-------------------------------------------------
char s[20] = {1,2,3,4};   extern char *s;
main(){                     f(){
                              /*печать указателя как long */
  f();                         printf( "%08lX\n", s );
}                           }
```

печатается 04030201.

2.29. Что напечатает программа?

```
  static char str1[ ]  = "abc";
  static char str2[4];

  strcpy( str2, str1 );
  /* можно ли написать str2 = str1; ? */

  printf( str1 == str2 ? "равно":"не равно" );
```

Как надо правильно сравнивать строки? Что на самом деле сравнивается в данном примере?

Ответ: сравниваются *адреса массивов*, хранящих строки. Так

```
char str1[2];
char str2[2];
main(){
  printf( str1 < str2 ? "<":">");
}
```

печатает <, а если написать

```
char str2[2];
char str1[2];
```

то напечатается >.

2.30. Напишите программу, спрашивающую ваше имя до тех пор, пока вы его правильно не введете. Для сравнения строк используйте функцию **strcmp**() (ее реализация есть в главе "Мобильность").

2.31. Какие значения возвращает функция **strcmp**() в следующей программе?

```
#include <stdio.h>
main() {
  printf("%d\n", strcmp("abc", "abc")); /*   0 */
  printf("%d\n", strcmp("ab" , "abc")); /* -99 */
  printf("%d\n", strcmp("abd", "abc")); /*   1 */
  printf("%d\n", strcmp("abc", "abd")); /*  -1 */
  printf("%d\n", strcmp("abc", "abe")); /*  -2 */
}
```

2.32. В качестве итога предыдущих задач: помните, что в Си строки (а не адреса) надо сравнивать как

```
if( strcmp("abc", "bcd") <  0) ... ;
if( strcmp("abc", "bcd") == 0) ... ;
          вместо
if( "abc" <  "bcd" ) ... ;
if( "abc" == "bcd" ) ... ;
```

и присваивать как

```
char d[80], s[80];
strcpy( d, s );      вместо    d = s;
```

2.33. Напишите программу, которая сортирует по алфавиту и печатает следующие ключевые слова языка Си:

```
int char double long
for while if
```

2.34. Вопрос не совсем про строки, скорее про цикл: чем плоха конструкция?

```
char s[]  = "You're a smart boy, now shut up.";
int i, len;
for(i=0; i < strlen(s); i++)
         putchar(s[i]);
```

Ответ: в соответствии с семантикой Си цикл развернется примерно в

```
        i=0;
LOOP:   if( !(i < strlen(s))) goto ENDLOOP;
          putchar(s[i]);
        i++;
        goto LOOP;
ENDLOOP:         ;
```

Заметьте, что хотя длина строки *s* не меняется, **strlen**(*s*) вычисляется на КАЖДОЙ итерации цикла, совершая лишнюю работу! Борьба с этим такова:

```
for(i=0, len=strlen(s); i < len; i++ )
         putchar(s[i]);
    или
for(i=0, len=strlen(s); len > 0; i++, --len )
         putchar(s[i]);
```

Аналогично, в цикле

```
while( i < strlen(s))...;
```

функция тоже будет вычисляться при каждой проверке условия! Это, конечно, относится к любой функции, используемой в условии, а не только к **strlen**. (Но, разумеется, случай когда функция возвращает признак "надо ли продолжать цикл" - совсем другое дело: такая функция *обязана* вычисляться каждый раз).

2.35. Что напечатает следующая программа?

```
#include <stdio.h>
main(){
    static char str[] = "До встречи в буфете";
    char *pt;

    pt = str; puts(pt); puts(++pt);
    str[7] = '\0'; puts(str); puts(pt);
    puts(++pt);
}
```

2.36. Что напечатает следующая программа?

```
main() {
    static char name[] = "Константин";
    char *pt;
    pt = name + strlen(name);
    while(--pt >= name)
         puts(pt);
}
```

2.37. Что напечатает следующая программа?

```
    char str1[] = "abcdef";
    char str2[] = "xyz";
    main(){
        register char *a, *b;
        a = str1; b = str2;
        while( *b )
               *a++ = *b++;
        printf( "str=%s a=%s\n", str1, a );

        a = str1; b = str2;
        while( *b )
               *++a = *b++;
        printf( "str=%s a=%s\n", str1, a );
    }
```

Ответ:

```
str=xyzdef a=def
str=xxyzef a=zef
```

2.38. Что печатает программа?

```
char *s;
for(s = "Ситроен"; *s; s+= 2){
    putchar(s[0]); if(!s[1]) break;
}
putchar('\n');
```

2.39. Что напечатает программа? Рассмотрите продвижение указателя s, указателей - элементов массива *strs*[]. Разберитесь с порядком выполнения операций. В каких случаях ++ изменяет указатель, а в каких - букву в строке? Нарисуйте себе картинку, изображающую состояние указателей - она поможет вам распутать эти спагетти. Уделите разбору этого примера достаточное время!

```
#include <stdio.h>      /* определение NULL */
/* Латинский алфавит: abcdefghijklmnopqrstuvwxyz */
char *strs[] = {
  "abcd","ABCD","0fpx","159",
  "hello","-gop","A1479",NULL
```

```
};
main(){
  char c,      **s = strs,       *p;
  c = *++*s;   printf("#1 %d %c %s\n", s-strs, c, *s);
  c = **++s;   printf("#2 %d %c %s\n", s-strs, c, *s);
  c = **s++;   printf("#3 %d %c %s\n", s-strs, c, *s);
  c = ++**s;   printf("#4 %d %c %s\n", s-strs, c, *s);
  c = (**s)++; printf("#5 %d %c %s\n", s-strs, c, *s);
  c = ++*++*s; printf("#6 %d %c %s\n", s-strs, c, *s);
  c = *++*s++; printf("#7 %d %c %s %s\n",
                          s-strs, c, *s, strs[2]);
  c = ++*++*s++; printf("#8 %d %c %s %s\n",
                          s-strs, c, *s, strs[3]);
  c = ++*++*++s; printf("#9 %d %c %s\n", s-strs,c,*s);
  c = ++**s++;   printf("#10 %d %c %s\n",s-strs,c,*s);
  p = *s; c = ++*(*s)++;
  printf("#11 %d %c %s %s %s\n",s-strs,c,*s,strs[6],p);
  c = ++*((*s)++); printf("#12 %d %c %s %s\n",
                            s-strs, c, *s, strs[6]);
  c = (*++(*s))++; printf("#13 %d %c %s %s\n",
                            s-strs, c, *s, strs[6]);
  for(s=strs; *s; s++)
      printf("strs[%d]=\"%s\"\n", s-strs, *s);
  putchar('\n');
}
```

Печатается:

```
#1 0 b bcd               strs[0]="bcd"
#2 1 A ABCD              strs[1]="ABCD"
#3 2 A 0fpx              strs[2]="px"
#4 2 1 1fpx              strs[3]="69"
#5 2 1 2fpx              strs[4]="hello"
#6 2 g gpx               strs[5]="iop"
#7 3 p 159 px            strs[6]="89"
#8 4 6 hello 69
#9 5 h hop
#10 6 i A1479
#11 6 B 1479 1479 B1479
#12 6 2 479 479
#13 6 7 89 89
```

Учтите, что конструкция

```
char *strs[1] = { "hello" };
```

означает, что в *strs*[0] содержится *указатель* на начальный байт *безымянного массива*, содержащего строку "hello". Этот указатель можно изменять! Попробуйте составить еще подобные примеры из *, ++, ().

2.40. Что печатает программа?

```
char str[25] = "Hi, ";
char *f(char **s){ int cnt;
  for(cnt=0; **s != '\0'; (*s)++, ++cnt);
  return("ny" + (cnt && (*s)[-1] == ' ') + (!cnt));
}
void main(void){ char *s = str;
  if( *f(&s) == 'y') strcat(s,  "dude");
  else               strcat(s, " dude");
  printf("%s\n", str);
}
```

Что она напечатает, если задать

```
char str[25]="Hi,";   или    char str[25]="";
```

2.41. В чем состоит ошибка? (Любимая ошибка начинающих)

```
main(){
   char *buf;      /* или char buf[]; */
   gets( buf );
   printf( "%s\n", buf );
}
```

Ответ: память под строку *buf* не выделена, указатель *buf* не проинициализирован и смотрит неизвестно куда. Надо было писать например так:

```
char buf[80];
        или
char mem[80], *buf = mem;
```

Обратите на этот пример **особое** внимание, поскольку, описав указатель (но никуда его не направив), новички успокаиваются, не заботясь о выделении памяти для хранения данных. Указатель должен указывать на ЧТО-ТО, в чем можно хранить данные, а не "висеть", указывая "пальцем в небо"! Запись информации по "висячему" указателю разрушает память программы и приводит к скорому (но часто не немедленному и потому таинственному) краху.

Вот программа, которая также использует неинициализированный указатель. На машине **SPARCstation 20** эта программа убивается операционной системой с диагностикой "Segmentation fault" (SIGSEGV). Это как раз и значит обращение по указателю, указывающему "пальцем в небо".

```
main(){
        int *iptr;
        int  ival  = *iptr;

        printf("%d\n", ival);
}
```

2.42. Для получения строки "Life is life" написана программа:

```
main(){
    char buf[ 60 ];
    strcat( buf, "Life " );
    strcat( buf, "is "   );
    strcat( buf, "life"  );
    printf( "%s\n", buf );
}
```

Что окажется в массиве *buf*?

Ответ: в начале массива окажется мусор, поскольку автоматический массив не инициализируется байтами '\0', а функция **strcat**() приписывает строки к концу строки. Для исправления можно написать

```
*buf = '\0';
```

перед первым **strcat**()-ом, либо вместо первого **strcat**()-а написать **strcpy**(*buf*, "Life ");

2.43. Составьте макроопределение **copystr**(*s1*, *s2*) для копирования строки *s2* в строку *s1*.

2.44. Составьте макроопределение **lenstr**(*s*) для вычисления длины строки.

Многие современные компиляторы сами обращаются с подобными короткими (1-3 оператора) стандартными функциями как с макросами, то есть при обращении к ним генерят не вызов функции, а подставляют текст ее тела в место обращения. Это делает объектный код несколько "толще", но зато быстрее. В расширенных диалектах Си и в Си++ компилятору можно предложить обращаться так и с вашей функцией - для этого функцию следует объявить как **inline** (такие функции называются еще "intrinsic").

2.45. Составьте рекурсивную и нерекурсивную версии программы инвертирования (зеркального отображения) строки:

```
abcdef --> fedcba.
```

2.46. Составьте функцию **index**(*s*, *t*), возвращающую номер первого вхождения символа *t* в строку *s*; если символ *t* в строку не входит, функция возвращает -1.

Перепишите эту функцию с указателями, чтобы она возвращала указатель на первое вхождение символа. Если символ в строке отсутствует - выдавать NULL. В **UNIX System-V** такая функция называется **strchr**. Вот возможный ответ:

```
char *strchr(s, c) register char *s, c;
{    while(*s && *s != c) s++;
     return *s == c ? s : NULL;
}
```

Заметьте, что p=**strchr**(s,'\0'); выдает указатель на конец строки. Вот пример использования:

```
extern char *strchr();
char *s = "abcd/efgh/ijklm";
char *p = strchr(s, '/');
printf("%s\n", p==NULL ? "буквы / нет" : p);
if(p) printf("Индекс вхождения = s[%d]\n", p - s );
```

2.47. Напишите функцию str**r**chr(), указывающую на последнее вхождение символа. Ответ:

```
char *strrchr(s, c) register char *s, c;
{     char *last = NULL;
      do if(*s == c) last = s; while(*s++);
      return last;
}
```

Вот пример ее использования:

```
extern char *strrchr();
char p[] = "wsh";          /* эталон */
main(argc, argv) char *argv[];{
    char *s = argv[1];     /* проверяемое имя */
    /* попробуйте вызывать
     * a.out csh
     * a.out /bin/csh
     * a.out wsh
     * a.out /usr/local/bin/wsh
     */
    char *base =
         (base = strrchr(s, '/')) ? base+1 : s;
    if( !strcmp(p, base))
         printf("Да, это %s\n" , p);
    else printf("Нет, это %s\n", base);

    /* еще более изощренный вариант: */
    if( !strcmp(p,(base=strrchr(s,'/')) ? ++base :
                                          (base=s))
      )  printf("Yes %s\n", p);
    else printf("No  %s\n", base);
}
```

2.48. Напишите макрос **substr**(*to*,*from*,*n*,*len*) который записывает в *to* кусок строки *from* начиная с *n*-ой позиции и длиной *len*. Используйте стандартную функцию **strncpy**.

Ответ:

```
#define substr(to, from, n, len) strncpy(to, from+n, len)
```

или более корректная функция:

```
char *substr(to, from, n, len) char *to, *from;
{
   int lfrom = strlen(from);
   if(n < 0 ){ len += n; n = 0; }
   if(n >= lfrom || len <= 0)
        *to = '\0';  /* пустая строка */
   else{
        /* длина остатка строки: */
        if(len > lfrom-n) len = lfrom - n;
        strncpy(to, from+n, len);
        to[len] = '\0';
   }
   return to;
}
```

2.49. Напишите функцию, проверяющую, оканчивается ли строка на "*.abc*", и если нет - приписывающую "*.abc*" к концу. Если же строка уже имеет такое окончание - ничего не делать. Эта функция полезна для генерации имен файлов с заданным расширением. Сделайте расширение аргументом функции.

Для сравнения конца строки *s* со строкой *p* следует использовать:

```
int ls = strlen(s), lp = strlen(p);
if(ls >= lp && !strcmp(s+ls-lp, p)) ...совпали...;
```

2.50. Напишите функции вставки символа *c* в указанную позицию строки (с раздвижкой строки) и удаления символа в заданной позиции (со сдвижкой строки). Строка должна изменяться "на месте", т.е. никуда не копируясь. Ответ:

```
/* удаление */
char delete(s, at) register char *s;
{
        char c;
        s += at; if((c = *s) == '\0') return c;
        while( s[0] = s[1] ) s++;
        return c;
}
/* либо просто strcpy(s+at, s+at+1); */

/* вставка */
insert(s, at, c) char s[], c;
{
        register char *p;
        s += at; p = s;
        while(*p) p++;  /* на конец строки */
        p[1] = '\0';    /* закрыть строку  */
        for( ; p != s; p-- )
                p[0] = p[-1];
        *s = c;
}
```

2.51. Составьте программу удаления символа *c* из строки *s* в каждом случае, когда он встречается.

Ответ:

```
delc(s, c) register char *s; char c;
{
   register char *p = s;
   while( *s )
     if( *s != c ) *p++ = *s++;
     else          s++;
   *p = '\0'; /* не забывайте закрывать строку ! */
}
```

2.52. Составьте программу удаления из строки *S1* каждого символа, совпадающего с каким-либо символом строки *S2*.

2.53. Составьте функцию **scopy**(*s*,*t*), которая копирует строку *s* в *t*, при этом символы табуляции и перевода строки должны заменяться на специальные двухсимвольные последовательности "\n" и "\t". Используйте **switch**.

2.54. Составьте функцию, которая "укорачивает" строку, заменяя изображения спецсимволов (вроде "\n") на сами эти символы ('\n'). Ответ:

```
extern char *strchr();
void unquote(s) char *s;
{       static char from[] = "nrtfbae",
                    to  [] = "\n\r\t\f\b\7\33";
        char c, *p, *d;

        for(d=s; c = *s; s++)
                if( c == '\\'){
                        if( !(c = *++s)) break;
```

```
                        p = strchr(from, c);
                        *d++ = p ? to[p - from] : c;
                    }else  *d++ = c;
            *d = '\0';
    }
```

2.55. Напишите программу, заменяющую в строке *S* все вхождения подстроки *P* на строку *Q*, например:

```
    P = "ура"; Q = "ой";
    S = "ура-ура-ура!";
       Результат: "ой-ой-ой!"
```

2.56. Кроме функций работы со строками (где предполагается, что массив байт завершается признаком конца '**\0**'), в Си предусмотрены также функции для работы с массивами байт без ограничителя. Для таких функций необходимо явно указывать *длину* обрабатываемого массива. Напишите функции: пересылки массива длиной *n* байт **memcpy**(*dst*,*src*,*n*); заполнения массива символом *c* **memset**(*s*,*c*,*n*); поиска вхождения символа в массив **memchr**(*s*,*c*,*n*); сравнения двух массивов **memcmp**(*s1*,*s2*,*n*); Ответ:

```
#define REG register
char *memset(s, c, n) REG char *s, c;
{    REG char *p = s;
     while( --n >= 0 ) *p++ = c;
     return s;
}
char *memcpy(dst, src, n)
      REG char *dst, *src;
      REG int n;
{     REG char *d = dst;
      while( n-- > 0 ) *d++ = *src++;
      return dst;
}
char *memchr(s, c, n) REG char *s, c;
{
      while(n-- && *s++ != c);
      return( n < 0 ? NULL : s-1 );
}
int memcmp(s1, s2, n)
      REG char *s1, *s2; REG n;
{
      while(n-- > 0 && *s1 == *s2)
        s1++, s2++;
      return( n < 0 ? 0 : *s1 - *s2 );
}
```

Есть такие *стандартные* функции.

2.57. Почему лучше пользоваться *стандартными* функциями работы со строками и памятью (**strcpy**, **strlen**, **strchr**, **memcpy**, ...)?

Ответ: потому, что они обычно реализованы поставщиками системы ЭФФЕКТИВНО, то есть написаны не на Си, а на ассемблере с использованием специализированных машинных команд и регистров. Это делает их более быстрыми. Написанный Вами эквивалент на Си может использоваться для повышения мобильности программы, либо для внесения поправок в стандартные функции.

2.58. Рассмотрим программу, копирующую строку саму в себя:

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

char string[] = "abcdefghijklmn";
void main(void){
        memcpy(string+2, string, 5);
        printf("%s\n", string);
        exit(0);
```

Она печатает abababahijklmn. Мы могли бы ожидать, что кусок длины 5 символов "abcde" будет скопирован как есть: ab[abcde]hijklmn, а получили ab[ababa]hijklmn - циклическое повторение первых двух символов строки... В чем дело? Дело в том, что когда области источника (src) и получателя (dst) перекрываются, то в

некий момент *src берется из УЖЕ перезаписанной ранее области, то есть испорченной! Вот программа, иллюстрирующая эту проблему:

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

char string[] = "abcdefghijklmn";
char *src = &string[0];
char *dst = &string[2];
int n     = 5;

void show(int niter, char *msg){
        register length, i;

        printf("#%02d %s\n", niter, msg);
        length = src-string;
        putchar('\t');
        for(i=0; i < length+3; i++) putchar(' ');
        putchar('S');  putchar('\n');

        printf("\t...%s...\n", string);

        length = dst-string;
        putchar('\t');
        for(i=0; i < length+3; i++) putchar(' ');
        putchar('D');  putchar('\n');
}

void main(void){
        int iter = 0;

        while(n-- > 0){
                show(iter,   "перед");
                  *dst++ = toupper(*src++);
                show(iter++, "после");
        }
        exit(0);
}
```

Она печатает:

```
#00 перед
          S
        ...abcdefghijklmn...
             D
#00 после
           S
        ...abAdefghijklmn...
              D
#01 перед
           S
        ...abAdefghijklmn...
              D
#01 после
            S
        ...abABefghijklmn...
              D
#02 перед
            S
        ...abABefghijklmn...
              D
#02 после
             S
        ...abABAfghijklmn...
               D
#03 перед
             S
        ...abABAfghijklmn...
               D
#03 после
              S
        ...abABABghijklmn...
                D
#04 перед
              S
        ...abABABghijklmn...
                D
#04 после
               S
        ...abABABAhijklmn...
                 D
```

Отрезки НЕ перекрываются, если один из них лежит либо целиком левее, либо целиком правее другого (n - длина обоих отрезков).

```
dst        src                  src        dst
########   @@@@@@@@             @@@@@@@@   ########

   dst+n <= src         или          src+n <= dst
   dst <= src-n         или          dst >= src+n
```

Отрезки перекрываются в случае

```
! (dst <= src - n || dst >= src + n) =
  (dst >  src - n && dst <  src + n)
```

При этом опасен только случай dst > src. Таким образом опасная ситуация описывается условием

```
src < dst && dst < src + n
```

(если dst==src, то вообще ничего не надо делать). Решением является копирование "от хвоста к голове":

```
void bcopy(register char *src, register char *dst,
           register int n){

        if(dst >= src){
                dst += n-1;
                src += n-1;
                while(--n >= 0)
                        *dst-- = *src--;
        }else{
                while(n-- > 0)
                        *dst++ = *src++;
        }
}
```

Или, ограничиваясь только опасным случаем:

```
void bcopy(register char *src, register char *dst,
           register int n){

        if(dst==src || n <= 0) return;
        if(src < dst && dst < src + n) {
                dst += n-1;
                src += n-1;
                while(--n >= 0)
                        *dst-- = *src--;
        }else   memcpy(dst, src, n);
}
```

Программа

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

char string[] = "abcdefghijklmn";
char *src = &string[0];
char *dst = &string[2];
int n     = 5;

void show(int niter, char *msg){
        register length, i;

        printf("#%02d %s\n", niter, msg);
        length = src-string;
        putchar('\t');
        for(i=0; i < length+3; i++) putchar(' ');
        putchar('S');  putchar('\n');

        printf("\t...%s...\n", string);

        length = dst-string;
        putchar('\t');
        for(i=0; i < length+3; i++) putchar(' ');
        putchar('D');  putchar('\n');
}
```

```
void main(void){
        int iter = 0;

        if(dst==src || n <= 0){
                printf("Ничего не надо делать\n");
                return;
        }

        if(src < dst && dst < src + n) {
                dst += n-1;
                src += n-1;
                while(--n >= 0){
                        show(iter,   "перед");
                          *dst-- = toupper(*src--);
                        show(iter++, "после");
                }
        }else
                while(n-- > 0){
                        show(iter,   "перед");
                          *dst++ = toupper(*src++);
                        show(iter++, "после");
                }
        exit(0);
}
```

Печатает

```
#00 перед
              S
        ...abcdefghijklmn...
                 D
#00 после
             S
        ...abcdefEhijklmn...
              D
#01 перед
             S
        ...abcdefEhijklmn...
              D
#01 после
            S
        ...abcdeDEhijklmn...
             D
#02 перед
            S
        ...abcdeDEhijklmn...
             D
#02 после
           S
        ...abcdCDEhijklmn...
            D
#03 перед
           S
        ...abcdCDEhijklmn...
            D
#03 после
          S
        ...abcBCDEhijklmn...
           D
#04 перед
          S
        ...abcBCDEhijklmn...
           D
#04 после
         S
        ...abABCDEhijklmn...
           D
```

Теперь **bcopy**() - удобная функция для копирования и сдвига массивов, в частности массивов указателей. Пусть у нас есть массив строк (выделенных **malloc**-ом):

```
char *lines[NLINES];
```

Тогда циклическая перестановка строк выглядит так:

```
void scrollUp(){
        char *save = lines[0];
        bcopy((char *) lines+1, /* from */
              (char *) lines,   /* to */
              sizeof(char *) * (NLINES-1));
        lines[NLINES-1] = save;
}
void scrollDown(){
        char *save = lines[NLINES-1];
        bcopy((char *) &lines[0], /* from */
              (char *) &lines[1], /* to */
              sizeof(char *) * (NLINES-1));
        lines[0] = save;
}
```

Возможно, что написание по аналогии функции для копирования массивов элементов типа (void *) - обобщенных указателей - может оказаться еще понятнее и эффективнее. Такая функция - **memmove** - стандартно существует в **UNIX SVR4**. Заметьте, что порядок аргументов в ней обратный по отношению к **bcopy**. Следует отметить, что в SVR4 *все* функции **mem...** имеют указатели типа (void *) и счетчик типа **size_t** - тип для количества байт (вместо **unsigned long**); в частности длина файла имеет именно этот тип (смотри системные вызовы **lseek** и **stat**).

```
#include <sys/types.h>

void memmove(void *Dst, const void *Src,
             register size_t n){

            register caddr_t src = (caddr_t) Src,
                             dst = (caddr_t) Dst;

            if(dst==src || n <= 0) return;
            if(src < dst && dst < src + n) {
                    dst += n-1;
                    src += n-1;
                    while(--n >= 0)
                            *dst-- = *src--;
            }else   memcpy(dst, src, n);
}
```

caddr_t - это тип для указателей на БАЙТ, фактически это (unsigned char *). Зачем вообще понадобилось использовать **caddr_t**? Затем, что для

```
void *pointer;
int n;
```

значение

```
pointer + n
```

не определено и невычислимо, ибо **sizeof(void)** не имеет смысла - это не 0, а просто ошибка, диагностируемая компилятором!

2.59. Еще об опечатках: вот что бывает, когда вместо знака `=' печатается `-' (на клавиатуре они находятся рядом...).

```
#include <stdio.h>
#include <strings.h>
char *strdup(const char *s){
        extern void *malloc();
        return strcpy((char *)malloc(strlen(s)+1), s);
}
char *ptr;
void main(int ac, char *av[]){
        ptr - strdup("hello"); /* подразумевалось ptr = ... */
        *ptr = 'H';
        printf("%s\n", ptr);
        free(ptr);
        exit(0);
}
```

Дело в том, что запись (а часто и чтение) по ***pointer**, где *pointer*==**NULL**, приводит к аварийному прекращению программы. В нашей программе *ptr* осталось равным **NULL** - указателем в никуда. В операционной системе **UNIX** на машинах с аппаратной защитой памяти, страница памяти, содержащая адрес **NULL** (0) бывает закрыта на запись, поэтому любое обращение по записи в эту страницу вызывает прерывание от диспетчера памяти и аварийное прекращение процесса. Система сама помогает ловить ваши ошибки (но уже во время выполнения программы). Это ОЧЕНЬ частая ошибка - запись по адресу **NULL**. MS DOS в таких случаях предпочитает просто зависнуть, и вы бываете вынуждены играть аккорд из трех клавиш - Ctrl/Alt/Del, так и не поняв в чем дело.

2.60. Раз уж речь зашла о функции **strdup** (кстати, это *стандартная* функция), приведем еще одну функцию для сохранения строк.

```
char *savefromto(register char *from, char *upto)
{
        char *ptr, *s;

        if((ptr = (char *) malloc(upto - from + 1)) == NULL)
                return NULL;

        for(s = ptr; from < upto; from++)
                *s++ = *from;

        *s = '\0';
        return ptr;
}
```

Сам символ (**upto*) не сохраняется, а заменяется на '\0'.

2.61. Упрощенный аналог функции **printf**.

```
/*
 * Машинно - независимый printf() (упрощенный вариант).
 * printf - Форматный Вывод.
 */
#include <stdio.h>
#include <ctype.h>
#include <varargs.h>
#include <errno.h>
#include <string.h>

extern int errno;       /* код системной ошибки, формат %m */

/* чтение значения числа */
#define GETN(n,fmt)                     \
        n = 0;                          \
        while(isdigit(*fmt)){           \
                n = n*10 + (*fmt - '0'); \
                fmt++;                  \
        }
```

```
void myprintf(fmt, va_alist)
       register char *fmt; va_dcl
{
  va_list ap;
  char c, *s; int i;
  int width, /* минимальная ширина поля */
      prec,  /* макс. длина данного */
      sign,  /* выравнивание: 1 - вправо, -1 - влево */
      zero,  /* ширина поля начинается с 0 */
      glong; /* требуется длинное целое */

  va_start(ap);
  for(;;){
     while((c = *fmt++) != '%'){
       if( c == '\0' ) goto out;
       putchar(c);
     }
     sign = 1; zero = 0; glong = 0;
     if(*fmt == '-'){ sign = (-1); fmt++; }
     if(*fmt == '0'){ zero = 1; fmt++; }
     if(*fmt == '*'){
         width = va_arg(ap, int);
         if(width < 0){ width = -width; sign = -sign; }
         fmt++;
     }else{
         GETN(width, fmt);
     }
     width *= sign;

     if(*fmt == '.'){
        if(*++fmt == '*'){
           prec = va_arg(ap, int); fmt++;
        }else{
           GETN(prec, fmt);
        }
     }else prec = (-1); /* произвольно */

     if( *fmt == 'l' ){
        glong = 1; fmt++;
     }
```

```
        switch(c = *fmt++){
        case 'c':
           putchar(va_arg(ap, int)); break;
        case 's':
           prStr(width, prec, va_arg(ap, char *)); break;
        case 'm':
           prStr(width, prec, strerror(errno));    break;
           /* strerror преобразует код ошибки в строку-расшифровку */
        case 'u':
           prUnsigned(width,
                     glong ? va_arg(ap, unsigned long) :
                             (unsigned long) va_arg(ap, unsigned int),
                     10 /* base */, zero); break;
        case 'd':
           prInteger(width,
                     glong ? va_arg(ap, long) : (long) va_arg(ap, int),
                     10 /* base */, zero);  break;
        case 'o':
           prUnsigned(width,
                     glong ? va_arg(ap, unsigned long) :
                             (unsigned long) va_arg(ap, unsigned int),
                     8 /* base */, zero);   break;
        case 'x':
           prUnsigned(width,
                     glong ? va_arg(ap, unsigned long) :
                             (unsigned long) va_arg(ap, unsigned int),
                     16 /* base */, zero);  break;
        case 'X':
           prUnsigned(width,
                     glong ? va_arg(ap, unsigned long) :
                             (unsigned long) va_arg(ap, unsigned int),
                     -16 /* base */, zero); break;
        case 'b':
           prUnsigned(width,
                     glong ? va_arg(ap, unsigned long) :
                             (unsigned long) va_arg(ap, unsigned int),
                     2 /* base */, zero);   break;
        case 'a':  /* address */
           prUnsigned(width,
                     (long) (char *) va_arg(ap, char *),
                     16 /* base */, zero);  break;
        case 'A':  /* address */
           prUnsigned(width,
                     (long) (char *) va_arg(ap, char *),
                     -16 /* base */, zero); break;
        case 'r':
           prRoman(width, prec, va_arg(ap, int)); break;
        case '%':
           putchar('%'); break;
        default:
           putchar(c);   break;
        }
     }
out:
     va_end(ap);
}

/* ------------------------------------------------------- */
int strnlen(s, maxlen) char *s;
{
        register n;
        for( n=0; *s && n < maxlen; n++, s++ );
        return n;
}
```

```
/* Печать строки */
static prStr(width, prec, s) char *s;
{
  int ln;         /* сколько символов выводить */
  int toLeft = 0; /* к какому краю прижимать   */

  if(s == NULL){ pr( "(NULL)", 6); return; }

  /* Измерить длину и обрубить длинную строку.
   * Дело в том, что строка может не иметь \0 на конце, тогда
   * strlen(s) может привести к обращению в запрещенные адреса */
  ln = (prec > 0 ? strnlen(s, prec) : strlen(s));

  /* ширина поля */
  if( ! width ) width = (prec > 0 ? prec : ln);
  if( width < 0){ width = -width; toLeft = 1; }
  if( width > ln){
        /* дополнить поле пробелами */
        if(toLeft){ pr(s, ln); prSpace(width - ln, ' ');  }
        else      { prSpace(width - ln, ' '); pr(s, ln);  }
  }     else      { pr(s, ln);                             }
}

/* Печать строки длиной l */
static pr(s, ln) register char *s; register ln;
{
  for( ; ln > 0 ; ln-- )
    putchar( *s++ );
}

/* Печать n символов c */
static prSpace(n, c) register n; char c;{
  for( ; n > 0 ; n-- )
    putchar( c );
}

/* ------------------------------------------------------------ */
static char *ds;

/* Римские цифры */
static prRoman(w,p,n){
        char bd[60];
        ds = bd;
        if( n < 0 ){ n = -n; *ds++ = '-'; }
        prRdig(n,6);
        *ds = '\0';
        prStr(w, p, bd);
}

static prRdig(n, d){
        if( !n ) return;
        if( d ) prRdig( n/10, d - 2);
        tack(n%10, d);
}
```

```
static tack(n, d){
        static char im[] = "  MDCLXVI";
                /* ..1000 500 100 50 10 5 1 */
        if( !n ) return;
        if( 1 <= n && n <= 3 ){
                repeat(n, im[d+2]); return;
        }
        if( n == 4 )
                *ds++ = im[d+2];
        if( n == 4 || n == 5 ){
                *ds++ = im[d+1]; return;
        }
        if( 6 <= n && n <= 8 ){
                *ds++ = im[d+1];
                repeat(n - 5, im[d+2] );
                return;
        }
        /* n == 9 */
        *ds++ = im[d+2]; *ds++ = im[d];
}

static repeat(n, c) char c;
{       while( n-- > 0 ) *ds++ = c;      }

/* ---------------------------------------------------------- */
static char aChar = 'A';

static prInteger(w, n, base, zero) long n;
{
        /* преобразуем число в строку */
        char bd[128];
        int neg = 0;    /* < 0 */

        if( n < 0 ){ neg = 1; n = -n; }

        if( base < 0 ){ base = -base; aChar = 'A'; }
        else          {               aChar = 'a'; }

        ds = bd; prUDig( n, base ); *ds = '\0';
        /* Теперь печатаем строку */
        prIntStr( bd, w, zero, neg );
}

static prUnsigned(w, n, base, zero) unsigned long n;
{
        char bd[128];

        if( base < 0 ){ base = -base; aChar = 'A'; }
        else          {               aChar = 'a'; }

        ds = bd; prUDig( n, base ); *ds = '\0';
        /* Теперь печатаем строку */
        prIntStr( bd, w, zero, 0 );
}

static prUDig( n, base ) unsigned long n;
{
        unsigned long aSign;

        if((aSign = n/base ) > 0 )
                prUDig( aSign, base );
        aSign = n % base;
        *ds++ = (aSign < 10 ? '0' + aSign : aChar + (aSign - 10));
}
```

```
static prIntStr( s, width, zero, neg ) char *s;
{
  int ln;         /* сколько символов выводить */
  int toLeft = 0; /* к какому краю прижимать   */

  ln = strlen(s);            /* длина строки s */

  /* Ширина поля: вычислить, если не указано явно */
  if( ! width ){
        width = ln;  /* ширина поля     */
        if( neg )   width++;         /* 1 символ для минуса */
  }
  if( width < 0 ){ width = -width; toLeft = 1; }

  if( ! neg ){  /* Положительное число */
    if(width > ln){
        if(toLeft){ pr(s, ln);              prSpace(width - ln, ' ');  }
        else      { prSpace(width - ln, zero ? '0' : ' '); pr(s, ln);  }
    }   else      { pr(s, ln);                                          }

  }else{        /* Отрицательное число */
    if(width > ln){
        /* Надо заполнять оставшуюся часть поля */

        width -- ; /* width содержит одну позицию для минуса */
        if(toLeft){ putchar('-'); pr(s, ln); prSpace(width - ln, ' ');  }
        else{
                if( ! zero ){
                        prSpace(width - ln, ' '); putchar('-'); pr(s,ln);
                } else {
                        putchar('-'); prSpace(width - ln, '0'); pr(s, ln);
                }
        }
    }   else    {       putchar('-'); pr(s, ln);   }
  }
}
```

```
/* ---------------------------------------------------------- */
main(){
        int i, n;
        static char s[] = "Hello, world!\n";
        static char p[] = "Hello, world";
        long t = 7654321L;

        myprintf( "%%abc%Y\n");
        myprintf( "%s\n",        "abs" );
        myprintf( "%5s|\n",       "abs" );
        myprintf( "%-5s|\n",      "abs" );
        myprintf( "%5s|\n",       "xyzXYZ" );
        myprintf( "%-5s|\n",      "xyzXYZ" );
        myprintf( "%5.5s|\n",     "xyzXYZ" );
        myprintf( "%-5.5s|\n",    "xyzXYZ" );
        myprintf( "%r\n",       444 );
        myprintf( "%r\n",       999 );
        myprintf( "%r\n",       16 );
        myprintf( "%r\n",       18 );
        myprintf( "%r\n",       479 );
        myprintf( "%d\n",  1234 );
        myprintf( "%d\n",  -1234 );
        myprintf( "%ld\n",  97487483 );
        myprintf( "%2d|%2d|\n",   1, -3 );
        myprintf( "%-2d|%-2d|\n", 1, -3 );
        myprintf( "%02d|%2d|\n",   1, -3 );
        myprintf( "%-02d|%-2d|\n", 1, -3 );
        myprintf( "%5d|\n",   -12 );
        myprintf( "%05d|\n",  -12 );
        myprintf( "%-5d|\n",  -12 );
        myprintf( "%-05d|\n", -12 );

        for( i = -6; i < 6; i++ )
             myprintf( "width=%2d|%0*d|%0*d|%*d|%*d|\n", i,
               i,  123,  i, -123, i,  123, i, -123);
        myprintf( "%s at location %a\n", s, s );
        myprintf( "%ld\n", t );
        n = 1; t = 1L;
        for( i=0; i < 34; i++ ){
     myprintf( "for %2d   |%016b|%d|%u|\n\t |%032lb|%ld|%lu|\n",
                     i,       n, n, n,           t,  t,  t );
                n *= 2;
                t *= 2;
        }
        myprintf( "%8x %8X\n", 7777, 7777 );
        myprintf( "|%s|\n", p );
        myprintf( "|%10s|\n", p );
        myprintf( "|%-10s|\n", p );
        myprintf( "|%20s|\n", p );
        myprintf( "|%-20s|\n", p );
        myprintf( "|%20.10s|\n", p );
        myprintf( "|%-20.10s|\n", p );
        myprintf( "|%.10s|\n", p );
}
```

Выдача этой программы:

```
%abcY
abs
  abs|
abs  |
xyzXYZ|
xyzXYZ|
xyzXY|
xyzXY|
CDXLIV
CMXCIX
XVI
XVIII
CDLXXIX
1234
-1234
97487483
 1|-3|
1 |-3|
01|-3|
1 |-3|
  -12|
-0012|
-12  |
-12  |
width=-6|123   |-123  |123   |-123  |
width=-5|123  |-123 |123  |-123 |
width=-4|123 |-123|123 |-123|
width=-3|123|-123|123|-123|
width=-2|123|-123|123|-123|
width=-1|123|-123|123|-123|
width= 0|123|-123|123|-123|
width= 1|123|-123|123|-123|
width= 2|123|-123|123|-123|
width= 3|123|-123|123|-123|
width= 4|0123|-123| 123|-123|
width= 5|00123|-0123|  123| -123|
Hello, world!
 at location 400980
7654321
for  0   |0000000000000001|1|1|
         |00000000000000000000000000000001|1|1|
for  1   |0000000000000010|2|2|
         |00000000000000000000000000000010|2|2|
for  2   |0000000000000100|4|4|
         |00000000000000000000000000000100|4|4|
for  3   |0000000000001000|8|8|
         |00000000000000000000000000001000|8|8|
for  4   |0000000000010000|16|16|
         |00000000000000000000000000010000|16|16|
for  5   |0000000000100000|32|32|
         |00000000000000000000000000100000|32|32|
for  6   |0000000001000000|64|64|
         |00000000000000000000000001000000|64|64|
for  7   |0000000010000000|128|128|
         |00000000000000000000000010000000|128|128|
for  8   |0000000100000000|256|256|
         |00000000000000000000000100000000|256|256|
for  9   |0000001000000000|512|512|
         |00000000000000000000001000000000|512|512|
for 10   |0000010000000000|1024|1024|
         |00000000000000000000010000000000|1024|1024|
for 11   |0000100000000000|2048|2048|
         |00000000000000000000100000000000|2048|2048|
for 12   |0001000000000000|4096|4096|
         |00000000000000000001000000000000|4096|4096|
```

```
for 13   |0010000000000000|8192|8192|
         |00000000000000000010000000000000|8192|8192|
for 14   |0100000000000000|16384|16384|
         |00000000000000000100000000000000|16384|16384|
for 15   |1000000000000000|32768|32768|
         |00000000000000001000000000000000|32768|32768|
for 16   |1000000000000000|65536|65536|
         |00000000000000010000000000000000|65536|65536|
for 17   |10000000000000000|131072|131072|
         |00000000000000100000000000000000|131072|131072|
for 18   |10000000000000000|262144|262144|
         |00000000000001000000000000000000|262144|262144|
for 19   |10000000000000000|524288|524288|
         |00000000000010000000000000000000|524288|524288|
for 20   |10000000000000000|1048576|1048576|
         |00000000001000000000000000000000|1048576|1048576|
for 21   |10000000000000000|2097152|2097152|
         |00000000010000000000000000000000|2097152|2097152|
for 22   |10000000000000000|4194304|4194304|
         |00000000100000000000000000000000|4194304|4194304|
for 23   |10000000000000000|8388608|8388608|
         |00000001000000000000000000000000|8388608|8388608|
for 24   |10000000000000000|16777216|16777216|
         |00000010000000000000000000000000|16777216|16777216|
for 25   |10000000000000000|33554432|33554432|
         |00000100000000000000000000000000|33554432|33554432|
for 26   |10000000000000000|67108864|67108864|
         |00001000000000000000000000000000|67108864|67108864|
for 27   |10000000000000000|134217728|134217728|
         |00010000000000000000000000000000|134217728|134217728|
for 28   |10000000000000000|268435456|268435456|
         |00100000000000000000000000000000|268435456|268435456|
for 29   |10000000000000000|536870912|536870912|
         |01000000000000000000000000000000|536870912|536870912|
for 30   |10000000000000000|1073741824|1073741824|
         |10000000000000000000000000000000|-2147483648|2147483648|
for 31   |10000000000000000|-2147483648|2147483648|
         |00000000000000000000000000000000|0|0|
for 32   |0000000000000000|0|0|
         |00000000000000000000000000000000|0|0|
for 33   |0000000000000000|0|0|
         |00000000000000000000000000000000|0|0|
    1e61     1E61
|Hello, world|
|Hello, world|
|Hello, world|
|        Hello, world|
|Hello, world        |
|          Hello, wor|
|Hello, wor          |
|Hello, wor|
```

2.62. Рассмотрим программу суммирования векторов:

```
int A[1024], B[1024], C[1024];
        ...
for(i=0; i < 1024; i++) C[i] = A[i] + B[i];
```

А почему бы не

```
for(i=1024-1; i >=0 ; --i) ...;
```

А почему бы не в произвольном порядке?

```
foreach i in (0..1023) ...;
```

Данный пример показывает, что некоторые операции обладают врожденным параллеллизмом, ведь все 1024 сложений можно было бы выполнять параллельно! Однако тупой компилятор будет складывать их именно в том порядке, в котором вы ему велели. Только самые современные компиляторы на многопроцессорных

системах умеют автоматически распараллеливать такие циклы. Сам язык **Си** не содержит средств указания параллельности (разве что снова - библиотеки и системные вызовы для этого).

3. Мобильность и машинная зависимость программ. Проблемы с русскими буквами.

Программа считается мобильной, если она без каких-либо изменений ее исходного текста (либо после настройки некоторых констант при помощи **#define** и **#ifdef**) транслируется и работает на разных типах машин (с разной разрядностью, системой команд, архитектурой, периферией) под управлением операционных систем одного семейства. Заметим, что мобильными могут быть только исходные тексты программ, объектные модули для разных процессоров, естественно, несовместимы!

3.1. Напишите программу, печатающую размер типов данных **char**, **short**, **int**, **long**, **float**, **double**, (**char** *) в байтах. Используйте для этого встроенную операцию *sizeof*.

3.2. Составьте мобильную программу, выясняющую значения следующих величин для любой машины, на которой работает программа:

1) Наибольшее допустимое знаковое целое.

2) Наибольшее беззнаковое целое.

3) Наибольшее по абсолютной величине отрицательное целое.

4) Точность значения $|x|$, отличающегося от 0, где x - вещественное число.

5) Наименьшее значение e, такое что машина различает числа 1 и $1+e$ (для вещественных чисел).

3.3. Составьте мобильную программу, выясняющую длину машинного слова ЭВМ (число битов в переменной типа **int**). Указание: для этого можно использовать битовые сдвиги.

3.4. Надо ли писать в своих программах определения

```
#define EOF  (-1)
#define NULL ((char *) 0)   /* или ((void *)0) */
```

Ответ: НЕТ. Во-первых, эти константы уже определены в include-файле, подключаемом по директиве

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

поэтому правильнее написать именно эту директиву. Во-вторых, это было бы просто неправильно: конкретные значения этих констант на данной машине (в данной реализации системы) могут быть другими! Чтобы придерживаться тех соглашений, которых придерживаются все стандартные функции данной реализации, вы ДОЛЖНЫ брать эти константы из <*stdio.h*>.

По той же причине следует писать

```
#include <fcntl.h>
int fd = open( имяФайла, O_RDONLY); /* O_WRONLY, O_RDWR */
       вместо
int fd = open( имяФайла, 0);         /* 1,        2       */
```

3.5. Почему может завершаться по защите памяти следующая программа?

```
#include <sys/types.h>
#include <stdio.h>
time_t t;
extern time_t time();
      ...
t = time(0);
/* узнать текущее время в секундах с 1 Янв. 1970 г.*/
```

Ответ: дело в том, что прототип системного вызова **time**() это:

```
time_t time( time_t *t );
```

то есть аргумент должен быть указателем. Мы же вместо указателя написали в качестве аргумента 0 (типа int). На машине **IBM PC AT 286** указатель - это 2 слова, а целое - одно. Недостающее слово будет взято из стека произвольно. В результате **time**() получает в качестве аргумента не нулевой указатель, а мусор. Правильно будет написать:

```
       t = time(NULL);
либо (по определению time())
          time( &t );
```

а еще более корректно так:

```
t = time((time_t *)NULL);
```

Мораль: везде, где требуется нулевой указатель, следует писать **NULL** (или явное приведение нуля к типу указателя), а не просто 0.

3.6. Найдите ошибку:

```
void f(x, s) long x; char *s;
{
    printf( "%ld %s\n", x, s );
}
void main(){
    f( 12, "hello" );
}
```

Эта программа работает на **IBM PC 386**, но не работает на **IBM PC 286**.

Ответ. Здесь возникает та же проблема, что и в примере про **sin**(12). Дело в том, что **f** требует первый аргумент типа **long** (4 байта на **IBM PC 286**), мы же передаем ей **int** (2 байта). В итоге в *x* попадает неверное значение; но более того, недостающие байты отбираются у следующего аргумента - *s*. В итоге и адрес строки становится неправильным, программа обращается по несуществующему адресу и падает. На **IBM PC 386** и **int** и **long** имеют длину 4 байта, поэтому там эта ошибка не проявляется!

Опять-таки, это повод для использования прототипов функций (когда вы прочитаете про них - вернитесь к этому примеру!). Напишите прототип

```
void f(long x, char *s);
```

и ошибки не будет.

В данном примере мы использовали тип **void**, которого не сушествовало в ранних версиях языка Си. Этот тип означает, что функция не возвращает значения (то есть является "процедурой" в смысле языков **Pascal** или **Algol**). Если мы не напишем слово **void** перед **f**, то компилятор будет считать функцию **f** возвращающей целое (**int**), хотя эта функция ничего не возвращает (в ней нет оператора **return**). В большинстве случаев это не принесет вреда и программа будет работать. Но зато если мы напишем

```
int x = f((long) 666, "good bye" );
```

то *x* получит непредсказуемое значение. Если же **f** описана как **void**, то написанный оператор заставит компилятор сообщить об ошибке.

Тип (**void ***) означает указатель на что угодно (понятно, что к такому указателю операции [], *, -> неприменимы: сначала следует явно привести указатель к содержательному типу "указатель на *тип*"). В частности, сейчас стало принято считать, что функция динамического выделения памяти (memory allocation) **malloc**() (которая отводит в куче£ область памяти заказанного размера и выдает указатель на нее) имеет прототип:

```
void *malloc(unsigned size); /* size байт */
char      *s = (char *)      malloc( strlen(buf)+1 );
struct ST *p = (struct ST *) malloc( sizeof(struct ST));
                              /* или sizeof(*p) */
```

хотя раньше принято было char *malloc();

3.7. Поговорим про оператор **sizeof**. Отметим распространенную ошибку, когда **sizeof** принимают за функцию. Это не так! **sizeof** вычисляется компилятором при *трансляции программы*, а не программой во время выполнения. Пусть

```
char a[] = "abcdefg";
char *b  = "hijklmn";
```

Тогда

```
sizeof(a)    есть 8  (байт \0 на конце - считается)
sizeof(b)    есть 2  на PDP-11 (размер указателя)
strlen(a)    есть 7
strlen(b)    есть 7
```

Если мы сделаем

£ "Куча" (*heap*, *pool*) - область статической памяти, увеличивающаяся по мере надобности, и предназначенная как раз для хранения динамически отведенных данных.

```
b = "This ia a new line";
strcpy(a, "abc");
```

то все равно

```
sizeof(b) останется равно 2
sizeof(a)                 8
```

Таким образом **sizeof** выдает количество зарезервированной для переменной памяти (в байтах), независимо от текущего ее содержимого.

Операция **sizeof** применима даже к выражениям. В этом случае она сообщает нам, каков будет размер у *результата* этого выражения. Само выражение при этом *не вычисляется*, так в

```
double f(){ printf( "Hi!\n"); return 12.34; }
main(){
        int x = 2; long y = 4;
        printf( "%u\n", sizeof(x + y + f()));
}
```

будет напечатано значение, совпадающее с **sizeof**(double), а фраза "Hi!" не будет напечатана.

Когда оператор **sizeof** применяется к переменной (а не к имени типа), можно не писать круглые скобки:

```
sizeof(char *);   но   sizeof x;
```

3.8. Напишите объединение, в котором может храниться либо указатель, либо целое, либо действительное число. Ответ:

```
union all{
      char *s; int i; double f;
} x;
x.i = 12  ; printf("%d\n", x.i);
x.f = 3.14; printf("%f\n", x.f);
x.s = "Hi, there"; printf("%s\n", x.s);
printf("int=%d double=%d (char *)=%d all=%d\n",
   sizeof(int), sizeof(double), sizeof(char *),
   sizeof x);
```

В данном примере вы обнаружите, что размер переменной *x* равен максимальному из размеров типов **int**, **double**, **char ***.

Если вы хотите использовать одну и ту же переменную для хранения данных разных типов, то для получения *мобильной* программы вы должны пользоваться только объединениями и никогда не привязываться к длине слова и представлению этих типов данных на конкретной ЭВМ! Раньше, когда программисты не думали о мобильности, они писали программы, где в одной переменой типа **int** хранили в зависимости от нужды то целые значения, то указатели (это было на машинах **PDP** и **VAX**). Увы, такие программы оказались непереносимы на машины, на которых *sizeof*(**int**) != *sizeof*(**char ***), более того, они оказались весьма туманны для понимания их другими людьми. Не следуйте этому стилю (такой стиль американцы называют "*poor style*"), более того, всеми силами *избегайте* его!

Сравните два примера, использующие два стиля программирования. Первый стиль не так плох, как только что описанный, но все же мы рекомендуем использовать только второй:

```
/* СТИЛЬ ПЕРВЫЙ: ЯВНЫЕ ПРЕОБРАЗОВАНИЯ ТИПОВ */
typedef void *PTR; /* универсальный указатель */
struct a { int x, y;    PTR pa; } A;
struct b { double u, v; PTR pb; } B;
#define Aptr(p) ((struct a *)(p))
#define Bptr(p) ((struct b *)(p))
PTR ptr1, ptr2;
main(){
    ptr1 = &A; ptr2 = &B;
    Bptr(ptr2)->u = Aptr(ptr1)->x = 77;
    printf("%f %d\n", B.u, A.x);
}

/* СТИЛЬ ВТОРОЙ: ОБ'ЕДИНЕНИЕ   */
/* предварительное объявление: */
extern struct a; extern struct b;
/* универсальный тип данных:   */
typedef union everything {
```

```
    int i; double d; char *s;
    struct a *ap; struct b *bp;
} ALL;
struct a { int x, y;    ALL pa; } A;
struct b { double u, v; ALL pb; } B;
ALL ptr1, ptr2, zz;
main(){
    ptr1.ap = &A; ptr2.bp = &B; zz.i = 77;
    ptr2.bp->u = ptr1.ap->x = zz.i;
    printf("%f %d\n", B.u, A.x);
}
```

3.9. Для выделения классов символов (например цифр), следует пользоваться макросами из include-файла <*ctype.h*> Так вместо

```
if( '0' <= c   &&    c <= '9' ) ...
```

следует использовать

```
#include <ctype.h>
     .....
if(isdigit(c)) ...
```

и вместо

```
if((c >='a' &&  c <= 'z') || (c >= 'A' && c <= 'Z')) ...
```

надо

```
if(isalpha(c)) ...
```

Дело в том, что сравнения < и > зависят от расположения букв в используемой кодировке. Но например, в кодировке **КОИ-8** русские буквы расположены НЕ в алфавитном порядке. Вследствие этого, если для

```
char c1, c2;
   c1 < c2
```

то это еще не значит, что буква *c1* предшествует букве *c2* в алфавите! Лексикографическое сравнение требует специальной перекодировки букв к "упорядоченной" кодировке.

Аналогично, сравнение

```
if( c >= 'a' && c <= 'я' )
```

скорее всего не даст ожидаемого результата. Макроопределения же в <*ctype.h*> используют массив флагов для каждой буквы кодировки, и потому не зависят от порядка букв (и работают быстрее). Идея реализации такова:

```
extern unsigned char _ctype[];  /*массив флагов*/
#define US(c)  (sizeof(c)==sizeof(char)?((c)&0xFF):(c))
/* подавление расширения знакового бита */
        /* Ф Л А Г И */
#define _U  01  /* uppercase: большая буква    */
#define _L  02  /* lowercase: малая буква      */
#define _N  04  /* number:    цифра            */
#define _S 010  /* space:     пробел           */
                /* ... есть и другие флаги ... */
#define isalpha(c) ((_ctype+1)[US(c)] & (_U|_L)   )
#define isupper(c) ((_ctype+1)[US(c)] &  _U       )
#define islower(c) ((_ctype+1)[US(c)] &     _L    )
#define isdigit(c) ((_ctype+1)[US(c)] &        _N )
#define isalnum(c) ((_ctype+1)[US(c)] & (_U|_L|_N))
#define tolower(c) ((c) + 'a' - 'A' )
#define toupper(c) ((c) + 'A' - 'a' )
```

где массив **_ctype**[] заполнен заранее (это проинициализированные статические данные) и хранится в стандартной библиотеке Си. Вот его фрагмент:

```
unsigned char _ctype[256 /* размер алфавита */ + 1] = {
 /* EOF   код (-1)        */       0,
              ...
 /* '1'   код  061 0x31   */      _N,
              ...
 /* 'A'   код 0101 0x41   */      _U,
              ...
 /* 'a'   код 0141 0x61   */      _L,
              ...
};
```

Выигрыш в скорости получается вот почему: если мы определим†

```
#define isalpha(c) (((c) >= 'a' && (c) <= 'z') || \
                    ((c) >= 'A' && (c) <= 'Z'))
```

то этот оператор состоит из 7 операций. Если же мы используем **isalpha** из <*ctype.h*> (как определено выше) - мы используем только две операции: индексацию и проверку битовой маски **&**. Операции **_ctype**+1 и **_U**|**_L** вычисляются до констант еще при компиляции, и поэтому не вызывают генерации машинных команд.

Определенные выше **toupper** и **tolower** работают верно лишь в кодировке **ASCII**£, в которой все латинские буквы расположены подряд и по алфавиту. Обратите внимание, что **tolower** имеет смысл применять только к большим буквам, а **toupper** - только к маленьким:

```
if( isupper(c) )  c = tolower(c);
```

Существует еще черезвычайно полезный макрос **isspace**(c), который можно было бы определить как

```
#define isspace(c) (c==' ' ||c=='\t'||c=='\f'|| \
                    c=='\n'||c=='\r')
        или
#define isspace(c) (strchr(" \t\f\n\r",(c)) != NULL)
```

На самом деле он, конечно, реализован через флаги в **_ctype**[]. Он используется для определения символов-пробелов, служащих заполнителями промежутков между *словами* текста.

Есть еще два нередко используемых макроса: **isprint**(*c*), проверяющий, является ли *c* ПЕЧАТНЫМ символом, т.е. имеющим изображение на экране; и **iscntrl**(*c*), означающий, что символ *c* является управляющим, т.е. при его выводе на терминал ничего не изобразится, но терминал произведет некоторое действие, вроде очистки экрана или перемещения курсора в каком-то направлении. Они нужны, как правило, для отображения управляющих ("контроловских") символов в специальном печатном виде, вроде **^A** для кода '\01'.

Задание: исследуйте кодировку и <*ctype.h*> на вашей машине. Напишите функцию лексикографического сравнения букв и строк.

Указание: пусть буквы имеют такие коды (это не соответствует реальности!):

```
буква:      а   б   в   г   д   е
код:        1   4   2   5   3   0

нужно:      0   1   2   3   4   5
```

Тогда идея функции **Ctou** перекодировки к упорядоченному алфавиту такова:

```
unsigned char UU[] = { 5, 0, 2, 4, 1, 3 };
/* в действительности - 256 элементов: UU[256] */

Ctou(c) unsigned char c; { return UU[c]; }

int strcmp(s1, s2) char *s1, *s2; {
    /* Проигнорировать совпадающие начала строк */
    while(*s1 && *s1 == *s2) s1++, s2++;
    /* Вернуть разность [не]совпавших символов  */
    return Ctou(*s1) - Ctou(*s2);
}
```

---

† Обратите внимание, что символ \ в конце строки макроопределения позволяет продолжить макрос на следующей строке, поэтому макрос может состоять из многих строк.

£ **ASCII** - American Standard Code for Information Interchange - наиболее распространенная в мире кодировка (Американский стандарт).

Разберитесь с принципом формирования массива *UU*.

3.10. В современных UNIX-ах с поддержкой различных языков таблица **ctype** загружается из некоторых системных файлов - для каждого языка своя. Для какого языка - выбирается по содержимому переменной окружения **LANG**. Если переменная не задана - используется значение "**C**", английский язык. Загрузка таблиц должна происходить явно, вызовом

```
...
#include <locale.h>
...
main(){
        setlocale(LC_ALL, "");
        ...
        все остальное
        ...
}
```

3.11. Вернемся к нашей любимой проблеме со знаковым битом у типа **char**.

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

int main(int ac, char *av[]){
        char c;
        char *string = "абвгдежзиклмноп";

        setlocale(LC_ALL, "");

        for(;c = *string;string++){
#ifdef DEBUG
                printf("%c %d %d\n", *string, *string, c);
#endif
                if(isprint(c)) printf("%c - печатный символ\n", c);
        }
        return 0;
}
```

Эта программа неожиданно печатает

```
% a.out
в - печатный символ
з - печатный символ
```

И все. В чем дело???

Рассмотрим к примеру символ 'г'. Его код '\307'. В операторе

```
c = *string;
```

Символ *c* получает значение -57 (десятичное), которое ОТРИЦАТЕЛЬНО. В системном файле */usr/include/ctype.h* макрос **isprint** определен так:

```
#define isprint(c)      ((_ctype + 1)[c] & (_P|_U|_L|_N|_B))
```

И значение *c* используется в нашем случае как *отрицательный индекс в массиве*, ибо индекс приводится к типу **int** (**signed**). Откуда теперь извлекается значение флагов - нам неизвестно; можно только с уверенностью сказать, что НЕ из массива **_ctype**.

Проблему решает либо использование

```
isprint(c & 0xFF)
```

либо

```
isprint((unsigned char) c)
```

либо объявление в нашем примере

```
unsigned char c;
```

В первом случае мы явно приводим **signed** к **unsigned** битовой операцией, обнуляя лишние биты. Во втором и третьем - **unsigned char** расширяется в **unsigned int**, который останется положительным. Вероятно, второй путь предпочтительнее.

3.12. Итак, снова напомним, что русские буквы **char**, а не **unsigned char** дают *отрицательные* индексы в массиве.

```
char c = 'г';
int x[256];

        ...x[c]...              /* индекс < 0 */
        ...x['г']...
```

Поэтому байтовые индексы должны быть либо **unsigned char**, либо **& 0xFF**. Как в следующем примере:

```
/* Программа преобразования символов в файле: транслитерация
                  tr abcd prst  заменяет строки
                  xxxxdbcaxxxx -> xxxxtrspxxxx
   По мотивам книги М.Дансмура и Г.Дейвиса.
*/
#include <stdio.h>

#define ASCII 256 /* число букв в алфавите ASCII */
/* BUFSIZ определено в stdio.h */
char mt[ ASCII ];       /* таблица перекодировки */

/* начальная разметка таблицы */
void mtinit(){
        register int i;
        for( i=0; i < ASCII; i++ )
                mt[i] = (char) i;
}

int main(int argc, char *argv[])
{
        register char *tin, *tout; /* unsigned char */
        char buffer[ BUFSIZ ];

        if( argc != 3 ){
                fprintf( stderr, "Вызов: %s что наЧто\n", argv[0] );
                return(1);
        }
        tin  = argv[1]; tout = argv[2];

        if( strlen(tin) != strlen(tout)){
                fprintf( stderr, "строки разной длины\n" );
                return(2);
        }

        mtinit();
        do{
                mt[ (*tin++) & 0xFF ]  = *tout++;
                /*   *tin - имеет тип char.
                 *   & 0xFF подавляет расширение знака
                 */
        } while( *tin );

        tout = mt;
        while( fgets( buffer, BUFSIZ, stdin ) != NULL ){
                for( tin = buffer; *tin; tin++ )
                        *tin = tout[ *tin & 0xFF ];
                fputs( buffer, stdout );
        }
        return(0);
}
```

3.13.

```
int main(int ac, char *av[]){
        char c = 'r';
        if('a' <= c && c < 256)
                printf("Это одна буква.\n");
        return 0;
}
```

Увы, эта программа не печатает НИЧЕГО. Просто потому, что **signed char** в сравнении (в операторе **if**) приводится к типу **int**. А как целое число - русская буква *отрицательна*. Снова решением является либо использование везде (*c* & 0xFF), либо объявление **unsigned char** *c*. В частности, этот пример показывает, что НЕЛЬЗЯ просто так сравнивать две переменные типа **char**. Нужно принимать предохранительные меры по подавлению расширения знака:

```
if((ch1 & 0xFF) < (ch2 & 0xFF))...;
```

Для **unsigned char** такой проблемы не будет.

3.14. Почему неверно:

```
#include <stdio.h>
main(){
        char c;

        while((c = getchar()) != EOF)
                putchar(c);
}
```

Потому что *c* описано как **char**, в то время как **EOF** - значение типа **int** равное (-1).

Русская буква "Большой твердый знак" в кодировке КОИ-8 имеет код '\377' (0xFF). Если мы подадим на вход этой программе эту букву, то в сравнении **signed char** со значением знакового целого **EOF**, *c* будет приведено тоже к знаковому целому - расширением знака. 0xFF превратится в (-1), что означает, что поступил символ **EOF**. Сюрприз!!! Посему данная программа будет делать вид, что в любом файле с большим русским твердым знаком после этого знака (и включая его) дальше ничего нет. Что есть досадное заблуждение.

Решением служит ПРАВИЛЬНОЕ объявление **int** *c*.

3.15. Изучите поведение программы

```
#define TYPE char

void f(TYPE c){
        if(c == 'й') printf("Это буква й\n");
        printf("c=%c c=\\%03o c=%03d c=0x%0X\n", c, c, c, c);
}

int main(){
        f('r'); f('й');
        f('z'); f('Z');
        return 0;
}
```

когда TYPE определено как **char**, **unsigned char**, **int**. Объясните поведение. Выдачи в этих трех случаях таковы (int == 32 бита):

```
c=r c=\37777777707 c=-57 c=0xFFFFFFC7
Это буква й
c=й c=\37777777712 c=-54 c=0xFFFFFFCA
c=z c=\172 c=122 c=0x7A
c=Z c=\132 c=090 c=0x5A

c=r c=\307 c=199 c=0xC7
c=й c=\312 c=202 c=0xCA
c=z c=\172 c=122 c=0x7A
c=Z c=\132 c=090 c=0x5A

и снова как 1 случай.
```

Рассмотрите альтернативу

```
        if(c == (unsigned char) 'й') printf("Это буква й\n");
```

где предполагается, что знак у русских букв и у *c* НЕ расширяется. В данном случае фраза 'Это буква й' не печатается ни с типом **char**, ни с типом **int**, поскольку в сравнении *c* приводится к типу **signed int** расширением знакового бита (который равен 1). Слева получается отрицательное число!

В таких случаях вновь следует писать

```
if((unsigned char)c == (unsigned char)'й') printf("Это буква й\n");
```

3.16. Обычно возникают проблемы при написании функций с переменным числом аргументов. В языке Си эта проблема решается использованием макросов **va_args**, не зависящих от соглашений о вызовах функций на данной машине, и использующих эти макросы специальных функций. Есть два стиля оформления таких программ: с использованием <*varargs.h*> и <*stdarg.h*>. Первый был продемонстрирован в первой главе на примере функции **poly**(). Для иллюстрации второго приведем пример функции трассировки, записывающей собщение в файл:

```
#include <stdio.h>
#include <stdarg.h>
void trace(char *fmt, ...) {
    va_list args;
    static FILE *fp = NULL;

    if(fp == NULL){
       if((fp = fopen("TRACE", "w")) == NULL) return;
    }
    va_start(args, fmt);
    /* второй аргумент: арг-т после которого
     * в заголовке функции идет ... */
    vfprintf(fp, fmt, args); /* библиотечная ф-ция */
    fflush(fp);     /* вытолкнуть сообщение в файл */
    va_end(args);
}

main(){ trace( "%s\n", "Go home.");
        trace( "%d %d\n", 12, 34);
}
```

Символ `...' (троеточие) в заголовке функции обозначает переменный (возможно пустой) список аргументов. Он должен быть самым *последним*, следуя за всеми обязательными аргументами функции.

Макрос **va_arg**(*args*,*type*), извлекающий из переменного списка аргументов `**...**' очередное значение типа *type*, одинаков в обоех моделях. Функция **vfprintf** может быть написана через функцию **vsprintf** (в действительности обе функции - стандартные):

```
int vfprintf(FILE *fp, const char *fmt, va_list args){
    /*static*/ char buffer[1024]; int res;
    res = vsprintf(buffer, fmt, args);
    fputs(buffer, fp); return res;
}
```

Функция **vsprintf**(*str*,*fmt*,*args*); аналогична функции **sprintf**(*str*,*fmt*,...) - записывает преобразованную по формату строку в байтовый массив *str*, но используется в контексте, подобном приведенному. В конец сформированной строки **sprintf** записывает '\0'.

3.17. Напишите функцию **printf**, понимающую форматы **%c** (буква), **%d** (целое), **%o** (восьмеричное), **%x** (шестнадцатеричное), **%b** (двоичное), **%r** (римское), **%s** (строка), **%ld** (длинное целое). Ответ смотри в приложении.

3.18. Для того, чтобы один и тот же исходный текст программы транслировался на разных машинах (в разных системах), приходится выделять в программе системно-зависимые части. Такие части должны поразному выглядеть на разных машинах, поэтому их оформляют в виде так называемых "*условно компилируемых*" частей:

```
#ifdef XX
        ... вариант1
#else
        ... вариант2
#endif
```

Эта директива препроцессора ведет себя следующим образом: если макрос с именем *XX* был определен
#define *XX*

то в программу подставляется *вариант1*, если же нет - *вариант2*. Оператор **#else** не обязателен - при его отсутствии *вариант2* пуст. Существует также оператор **#ifndef**, который подставляет *вариант1* если макрос *XX* **не** определен. Есть еще и оператор **#elif** - else if:

```
#ifdef макро1
  ...
#elif  макро2
  ...
#else
  ...
#endif
```

Определить макрос можно не только при помощи **#define**, но и при помощи ключа компилятора, так

```
cc -DXX file.c ...
```

соответствует включению в начало файла *file.c* директивы

```
#define XX
```

А для программы

```
main(){
#ifdef XX
        printf( "XX = %d\n", XX);
#else
        printf( "XX undefined\n");
#endif
}
```

ключ

```
cc -D"XX=2" file.c ...
```

эквивалентен заданию директивы

```
#define XX 2
```

Что будет, если совсем не задать ключ **-D** в данном примере?

Этот прием используется в частности в тех случаях, когда какие-то стандартные типы или функции в данной системе носят другие названия:

```
cc -Dvoid=int ...
cc -Dstrchr=index ...
```

В некоторых системах компилятор *автоматически* определяет специальные макросы: так компиляторы в **UNIX** неявно подставляют один из ключей (или несколько сразу):

```
-DM_UNIX
-DM_XENIX
-Dunix
-DM_SYSV
-D__SVR4
-DUSG
... бывают и другие
```

Это позволяет программе "узнать", что ее компилируют для системы **UNIX**. Более подробно про это написано в документации по команде **cc**.

3.19. Оператор **#ifdef** применяется в include-файлах, чтобы исключить повторное включение одного и того же файла. Пусть файлы *aa.h* и *bb.h* содержат

```
       aa.h                          bb.h
#include "cc.h"                  #include "cc.h"
typedef unsigned long ulong;     typedef int cnt_t;
```

А файлы *cc.h* и *00.c* содержат

```
       cc.h                          00.c
       ...                       #include "aa.h"
struct II { int x, y; };         #include "bb.h"
       ...                       main(){ ... }
```

В этом случае текст файла *cc.h* будет вставлен в *00.c* **дважды**: из *aa.h* и из *bb.h*. При компиляции *00.c* компилятор сообщит "Переопределение структуры II". Чтобы include-файл не подставлялся еще раз, если он уже однажды был включен, придуман следующий прием - следует оформлять файлы включений так:

```
/* файл   cc.h */
#ifndef  _CC_H
# define _CC_H  /* определяется при первом включении */
        ...
        struct II { int x, y; };
        ...
#endif /* _CC_H */
```

Второе и последующие включения такого файла будут подставлять *пустое место*, что и требуется. Для файла <*sys/types.h*> было бы использовано макроопределение **_SYS_TYPES_H**.

3.20. Любой макрос можно отменить, написав директиву

#undef *имяМакро*

Пример:

```
#include <stdio.h>
#undef M_UNIX
#undef M_SYSV
main() {
        putchar('!');
#undef  putchar
#define putchar(c) printf( "Буква '%c'\n", c);
        putchar('?');

#if defined(M_UNIX) || defined(M_SYSV)
/* или просто #if M_UNIX */
        printf("Это UNIX\n");
#else
        printf("Это не UNIX\n");
#endif /* UNIX */
}
```

Обычно **#undef** используется именно для *пере*определения макроса, как **putchar** в этом примере (дело в том, что **putchar** - это макрос из <*stdio.h*>).

Директива **#if**, использованная нами, является расширением оператора **#ifdef** и подставляет текст если выполнено указанное условие:

```
#if  defined(MACRO)  /* равно #ifdef(MACRO)  */
#if !defined(MACRO)  /* равно #ifndef(MACRO) */
#if VALUE > 15       /* если целая константа
                        #define VALUE 25
                        больше 15 (==, !=, <=, ...) */
#if COND1 || COND2   /* если верно любое из условий */
#if COND1 && COND2   /* если верны оба условия      */
```

Директива **#if** допускает использование в качестве аргумента довольно сложных выражений, вроде

```
#if !defined(M1) && (defined(M2) || defined(M3))
```

3.21. Условная компиляция может использоваться для трассировки программ:

```
#ifdef DEBUG
# define DEBUGF(body)   \
{                       \
        body;           \
}
#else
# define DEBUGF(body)
#endif

int f(int x){   return x*x; }
int main(int ac, char *av[]){
        int x = 21;
        DEBUGF(x = f(x); printf("%s equals to %d\n", "x", x));
        printf("x=%d\n", x);
}
```

При компиляции

```
cc -DDEBUG file.c
```

в выходном потоке программы будет присутствовать отладочная выдача. При компиляции без **-D***DEBUG* этой выдачи не будет.

3.22. В языке **C++** (развитие языка **Си**) слова **class**, **delete**, **friend**, **new**, **operator**, **overload**, **template**, **public**, **private**, **protected**, **this**, **virtual** являются зарезервированными (ключевыми). Это может вызвать небольшую проблему при переносе текста программы на **Си** в систему программирования **C++**, например:

```
#include <termio.h>
  ...
int fd_tty = 2;   /* stderr */
struct termio old, new;
ioctl (fd_tty, TCGETA, &old);
new = old;
new.c_lflag |= ECHO | ICANON;
ioctl (fd_tty, TCSETAW, &new);
  ...
```

Строки, содержащие имя переменной (или функции) **new**, окажутся неправильными в **C++**. Проще всего эта проблема решается переименованием переменной (или функции). Чтобы не производить правки во всем тексте, достаточно переопределить имя при помощи директивы **define**:

```
#define new    new_modes
  ... старый текст ...
#undef new
```

При переносе программы на **Си** в **C++** следует также учесть, что в **C++** для *каждой* функции должен быть задан *прототип*, прежде чем эта функция будет использована (**Си** позволяет опускать прототипы для многих функций, особенно возвращающих значения типов **int** или **void**).

4. Работа с файлами.

Файлы представляют собой области памяти на внешнем носителе (как правило магнитном диске), предназначенные для:

- хранения данных, превосходящих по объему память компьютера (меньше, разумеется, тоже можно);
- долговременного хранения информации (она сохраняется при выключении машины).

В **UNIX** и в **MS DOS** файлы не имеют предопределенной структуры и представляют собой просто линейные *массивы байт*. Если вы хотите задать некоторую структуру хранимой информации - вы должны позаботиться об этом в своей программе *сами*. Файлы отличаются от обычных массивов тем, что

- они могут изменять свой размер;
- обращение к элементам этих массивов производится не при помощи операции индексации [], а при помощи специальных системных вызовов и функций;
- доступ к элементам файла происходит в так называемой "позиции чтения/записи", которая автоматически продвигается при операциях чтения/записи, т.е. файл просматривается последовательно. Есть, правда, функции для произвольного изменения этой позиции.

Файлы имеют *имена* и организованы в иерархическую древовидную структуру из *каталогов* и простых файлов. Об этом и о системе именования файлов прочитайте в документации по **UNIX**.

4.1. Для работы с каким-либо файлом наша программа должна *открыть* этот файл - установить связь между именем файла и некоторой переменной в программе. При открытии файла в ядре операционной системы выделяется "связующая" структура **file** "*открытый файл*", содержащая:

f_offset:
: указатель позиции чтения/записи, который в дальнейшем мы будем обозначать как *RWptr*. Это **long**-число, равное расстоянию в байтах от начала файла до позиции чтения/записи;

f_flag: режимы открытия файла: чтение, запись, чтение и запись, некоторые дополнительные флаги;

f_inode:
: расположение файла на диске (в **UNIX** - в виде ссылки на **I-узел** файла†);

и кое-что еще.

У каждого процесса имеется таблица открытых им файлов - это массив ссылок на упомянутые "связующие" структуры£. При открытии файла в этой таблице ищется свободная ячейка, в нее заносится ссылка на структуру "открытый файл" в ядре, и ИНДЕКС этой ячейки выдается в вашу программу в виде целого числа - так называемого "*дескриптора файла*".

При *закрытии* файла связная структура в ядре уничтожается, ячейка в таблице считается свободной, т.е. связь программы и файла разрывается.

---

† **I-узел** (I-node, индексный узел) - своеобразный "паспорт", который есть у каждого файла (в том числе и каталога). В нем содержатся:

```
- длина файла                    long   di_size;
- номер владельца файла          int    di_uid;
- коды доступа и тип файла       ushort di_mode;
- время создания и последней модификации
                time_t di_ctime, di_mtime;
- начало таблицы блоков файла    char   di_addr[...];
- количество имен файла          short  di_nlink;
и.т.п.
```

Содержимое некоторых полей этого паспорта можно узнать вызовом **stat**(). Все I-узлы собраны в единую область в начале файловой системы - так называемый **I-файл**. Все I-узлы пронумерованы, начиная с номера 1. Корневой каталог (файл с именем "/") как правило имеет I-узел номер 2.

£ У каждого процесса в **UNIX** также есть свой "паспорт". Часть этого паспорта находится в таблице процессов в ядре ОС, а часть - "приклеена" к самому процессу, однако не доступна из программы непосредственно. Эта вторая часть паспорта носит название "**u-area**" или структура **user**. В нее, в частности, входят таблица открытых процессом файлов

struct **file** **u_ofile*[NOFILE];

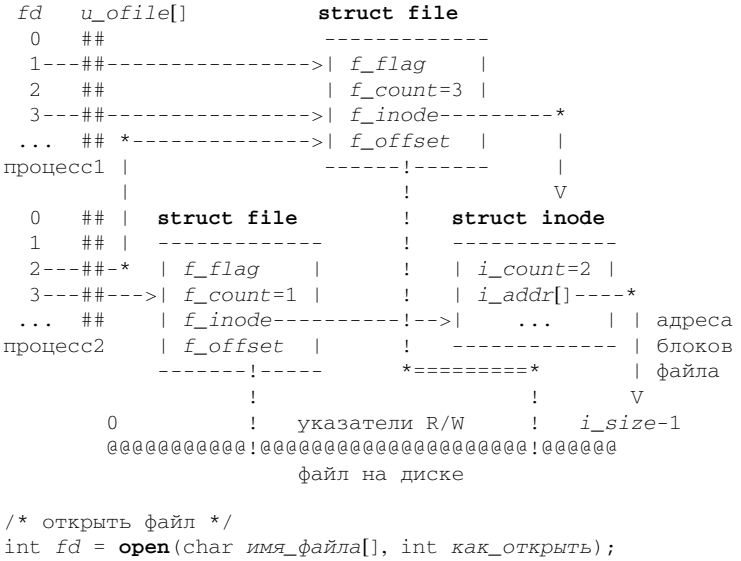
ссылка на I-узел текущего каталога

struct **inode** **u_cdir*;

а также ссылка на часть паспорта в таблице процессов

struct **proc** **u_procp*;

Дескрипторы являются *локальными* для каждой программы. Т.е. если две программы открыли один и тот же файл - дескрипторы этого файла в каждой из них не обязательно совпадут (хотя и могут). Обратно: одинаковые дескрипторы (номера) в разных программах не обязательно обозначают один и тот же файл. Следует учесть и еще одну вещь: несколько или один процессов могут открыть один и тот же файл одновременно *несколько* раз. При этом будет создано несколько "связующих" структур (по одной для каждого открытия); каждая из них будет иметь СВОЙ указатель чтения/записи. Возможна и ситуация, когда несколько дескрипторов ссылаются к одной структуре - смотри ниже описание вызова **dup2**.




```
/* открыть файл */
int fd = open(char имя_файла[], int как_открыть);
        ...  /* какие-то операции с файлом */
close(fd);  /* закрыть */
```

Параметр *как_открыть*:

```
#include <fcntl.h>
O_RDONLY  - только для чтения.
O_WRONLY  - только для записи.
O_RDWR    - для чтения и записи.
O_APPEND  - иногда используется вместе с
открытием для записи, "добавление" в файл:
    O_WRONLY|O_APPEND, O_RDWR|O_APPEND
```

Если файл еще не существовал, то его нельзя открыть: **open** вернет значение (-1), сигнализирующее об ошибке. В этом случае файл надо создать:

```
int fd = creat(char имя_файла[], int коды_доступа);
```

Дескриптор *fd* будет открыт для записи в этот новый пустой файл. Если же файл уже существовал, **creat** опустошает его, т.е. уничтожает его прежнее содержимое и делает его длину равной 0L байт. *Коды_доступа* задают права пользователей на доступ к файлу. Это число задает битовую шкалу из 9и бит, соответствующих строке

```
биты:   876 543 210
        rwx rwx rwx
r - можно читать файл
w - можно записывать в файл
x - можно выполнять программу из этого файла
```

Первая группа - эта права владельца файла, вторая - членов его группы, третяя - всех прочих. Эти коды для владельца файла имеют еще и мнемонические имена (используемые в вызове **stat**):

```
#include <sys/stat.h>  /* Там определено: */
#define S_IREAD          0400
#define S_IWRITE         0200
#define S_IEXEC          0100
```

Подробности - в руководствах по системе **UNIX**. Отметим в частности, что **open**() может вернуть код ошибки *fd* < 0 не только в случае, когда файл не существует (*errno*==**ENOENT**), но и в случае, когда вам не разрешен соответствующий доступ к этому файлу (*errno*==**EACCES**; про переменную кода ошибки *errno* см. в главе "Взаимодействие с **UNIX**").

Вызов **creat** - это просто разновидность вызова **open** в форме

```
fd = open( имя_файла,
           O_WRONLY|O_TRUNC|O_CREAT, коды_доступа);
```

O_TRUNC
означает, что если файл уже существует, то он должен быть опустошен при открытии. Коды доступа и владелец не изменяются.

O_CREAT
означает, что файл должен быть создан, если его не было (без этого флага файл не создастся, а **open** вернет *fd* < 0). Этот флаг требует задания третьего аргумента *коды_доступа*†. Если файл уже существует - этот флаг не имеет никакого эффекта, но зато вступает в действие **O_TRUNC**.

Существует также флаг

O_EXCL
который может использоваться совместно с **O_CREAT**. Он делает следующее: если файл уже существует, **open** вернет код ошибки (*errno*==**EEXIST**). Если файл не существовал - срабатывает **O_CREAT** и файл создается. Это позволяет предохранить уже существующие файлы от уничтожения.

Файл удаляется при помощи

```
int unlink(char имя_файла[]);
```

У каждой программы по умолчанию открыты три первых дескриптора, обычно связанные

```
0 - с клавиатурой (для чтения)
1 - с дисплеем    (выдача результатов)
2 - с дисплеем    (выдача сообщений об ошибках)
```

Если при вызове **close**(*fd*) дескриптор *fd* не соответствует открытому файлу (не был открыт) - ничего не происходит.

Часто используется такая метафора: если представлять себе файлы как книжки (только чтение) и блокноты (чтение и запись), стоящие на полке, то *открытие* файла - это выбор блокнота по заглавию на его обложке и открытие обложки (на первой странице). Теперь можно читать записи, дописывать, вычеркивать и править записи в середине, листать книжку! Страницы можно сопоставить *блокам* файла (см. ниже), а "полку" с книжками - каталогу.

4.2. Напишите программу, которая копирует содержимое одного файла в другой (новый) файл. При этом используйте системные вызовы чтения и записи **read** и **write**. Эти сисвызовы пересылают массивы байт из памяти в файл и наоборот. Но любую переменную можно рассматривать как массив байт, если забыть о структуре данных в переменной!

Читайте и записывайте файлы большими кусками, кратными 512 байтам. Это уменьшит число обращений к диску. Схема:

```
char buffer[512]; int n; int fd_inp, fd_outp;
        ...
while((n = read (fd_inp,  buffer, sizeof buffer)) > 0)
           write(fd_outp, buffer, n);
```

Приведем несколько примеров использования **write**:

```
char c = 'a';
int  i = 13, j = 15;
char s[20] = "foobar";
```

---

† Заметим, что на самом деле коды доступа у нового файла будут равны
di_mode = (*коды_доступа* & ~*u_cmask*) | **IFREG**;
(для каталога вместо **IFREG** будет **IFDIR**), где маска *u_cmask* задается системным вызовом
umask(*u_cmask*);
(вызов выдает прежнее значение маски) и в дальнейшем наследуется всеми потомками данного процесса (она хранится в **u-area** процесса). Эта маска позволяет запретить доступ к определенным операциям для *всех* создаваемых нами файлов, несмотря на явно заданные коды доступа, например
umask(0077); /* ???------ */
делает значащими только первые 3 бита кодов доступа (для владельца файла). Остальные биты будут равны нулю.

Все это относится и к созданию каталогов вызовом **mkdir**.

```
char p[]   = "FOOBAR";
struct { int x, y; } a = { 666, 999 };
/* создаем файл с доступом    rw-r--r-- */
int fd = creat("aFile", 0644);
write(fd, &c, 1);
write(fd, &i, sizeof i);  write(fd, &j, sizeof(int));
write(fd, s,  strlen(s)); write(fd, &a, sizeof a);
write(fd, p,  sizeof(p) - 1);
close(fd);
```

Обратите внимание на такие моменты:

- При использовании **write**() и **read**() надо передавать АДРЕС данного, которое мы хотим записать в файл (места, куда мы хотим прочитать данные из файла).
- Операции **read** и **write** возвращают число действительно прочитанных/записанных байт (при записи оно может быть меньше указанного нами, если на диске не хватает места; при чтении - если от позиции чтения до конца файла содержится меньше информации, чем мы затребовали).
- Операции **read**/**write** продвигают указатель чтения/записи

  ```
  RWptr += прочитанное_или_записанное_число_байт;
  ```

  При открытии файла указатель стоит на начале файла: *RWptr*=0. При записи файл если надо автоматически увеличивает свой размер. При чтении - если мы достигнем конца файла, то **read** будет возвращать "прочитано 0 байт" (т.е. при чтении указатель чтения не может стать больше размера файла).
- Аргумент *сколькоБайт* имеет тип **unsigned**, а не просто **int**:

```
int n = read (int fd, char *адрес, unsigned сколькоБайт);
int n = write(int fd, char *адрес, unsigned сколькоБайт);
```

Приведем *упрощенные* схемы логики этих сисвызовов, когда они работают с обычным дисковым файлом (в **UNIX** *устройства* тоже выглядят для программ как файлы, но иногда с особыми свойствами):

4.2.1. m = write(fd, addr, n);

```
если( ФАЙЛ[fd] не открыт на запись) то вернуть (-1);
если(n == 0) то вернуть 0;
если( ФАЙЛ[fd] открыт на запись с флагом O_APPEND ) то
  RWptr = длина_файла; /* т.е. встать на конец файла */
если( RWptr > длина_файла ) то
  заполнить нулями байты файла в интервале
  ФАЙЛ[fd][ длина_файла..RWptr-1 ] = '\0';
скопировать байты из памяти процесса в файл
  ФАЙЛ[fd][ RWptr..RWptr+n-1 ] = addr[ 0..n-1 ];
  отводя на диске новые блоки, если надо
RWptr += n;
если( RWptr > длина_файла ) то
      длина_файла = RWptr;
вернуть n;
```

4.2.2. m = read(fd, addr, n);

```
если( ФАЙЛ[fd] не открыт на чтение) то вернуть (-1);
если( RWptr >= длина_файла ) то вернуть 0;
m = MIN( n, длина_файла - RWptr );
скопировать байты из файла в память процесса
  addr[ 0..m-1 ] = ФАЙЛ[fd][ RWptr..RWptr+m-1 ];
RWptr += m;
вернуть m;
```

4.3. Найдите ошибки в фрагменте программы:

```
#define STDOUT 1  /* дескриптор стандартного вывода */
int i;
static char s[20] = "hi\n";
char c = '\n';
struct a{ int x,y; char ss[5]; } po;

scanf( "%d%d%d%s%s", i, po.x, po.y, s, po.ss);
write( STDOUT, s, strlen(s));
write( STDOUT, c, 1 );       /* записать 1 байт */
```

Ответ: в функции **scanf** перед аргументом *i* должна стоять операция "адрес", то есть **&***i*. Аналогично про **&***po.x* и **&***po.y*. Заметим, что *s* - это *массив*, т.е. *s* и так есть адрес, поэтому перед *s* операция **&** не нужна; аналогично про *po.ss* - здесь **&** не требуется.

В системном вызове **write** второй аргумент должен быть *адресом* данного, которое мы хотим записать в файл. Поэтому мы должны были написать **&***c* (во втором вызове **write**).

Ошибка в **scanf** - указание *значения* переменной вместо ее *адреса* - является довольно распространенной и *не может* быть обнаружена компилятором (даже при использовании прототипа функции **scanf**(char **fmt*, **...**), так как **scanf** - функция с переменным числом аргументов заранее *не определенных* типов). Приходится полагаться исключительно на собственную внимательность!

4.4. Как по дескриптору файла узнать, открыт он на чтение, запись, чтение и запись одновременно? Вот два варианта решения:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/param.h> /* там определено NOFILE */
#include <errno.h>

char *typeOfOpen(fd){
  int flags;
  if((flags=fcntl (fd, F_GETFL, NULL)) < 0 )
    return NULL;  /* fd вероятно не открыт */
  flags &= O_RDONLY | O_WRONLY | O_RDWR;
  switch(flags){
  case O_RDONLY:  return "r";
  case O_WRONLY:  return "w";
  case O_RDWR:    return "r+w";
  default:        return NULL;
  }
}

char *type2OfOpen(fd){
  extern errno; /* см. главу "системные вызовы" */
  int r=1, w=1;
  errno = 0; read(fd, NULL, 0);
  if( errno == EBADF ) r = 0;
  errno = 0; write(fd, NULL, 0);
  if( errno == EBADF ) w = 0;
  return (w && r) ? "r+w" :
          w       ? "w"   :
          r       ? "r"   :
                    "closed";
}

main(){
  int i; char *s, *p;
  for(i=0; i < NOFILE; i++ ){
    s = typeOfOpen(i); p = type2OfOpen(i);
    printf("%d:%s %s\n", i, s? s: "closed", p);
  }
}
```

Константа **NOFILE** означает максимальное число одновременно открытых файлов для одного процесса (это размер таблицы открытых процессом файлов, таблицы дескрипторов). Изучите описание системного вызова **fcntl** (file control).

4.5. Напишите функцию *rename*() для переименования файла. Указание: используйте системные вызовы **link**() и **unlink**(). Ответ:

```
rename( from, to )
  char *from,     /* старое имя */
       *to;       /* новое имя  */
{
    unlink( to );   /* удалить файл to    */
    if( link( from, to ) < 0 ) /* связать */
        return (-1);
    unlink( from ); /* стереть старое имя */
    return 0;       /* OK */
}
```

Вызов

link(*существующее_имя*, *новое_имя*);

создает файлу альтернативное имя - в **UNIX** файл может иметь несколько имен: так каждый каталог имеет какое-то имя в родительском каталоге, а также имя "." в себе самом. Каталог же, содержащий подкаталоги, имеет некоторое имя в своем родительском каталоге, имя "." в себе самом, и по одному имени ".." в каждом из своих подкаталогов.

Этот вызов будет неудачен, если файл *новое_имя* уже существует; а также если мы попытаемся создать альтернативное имя в *другой* файловой системе. Вызов

unlink(*имя_файла*)

удаляет имя файла. Если файл больше не имеет имен - он уничтожается. Здесь есть одна тонкость: рассмотрим фрагмент

```
int fd;
close(creat("/tmp/xyz", 0644)); /*Создать пустой файл*/
fd = open("/tmp/xyz", O_RDWR);
unlink("/tmp/xyz");
        ...
close(fd);
```

Первый оператор создает пустой файл. Затем мы открываем файл и уничтожаем его единственное имя. Но поскольку есть программа, открывшая этот файл, он не удаляется немедленно! Программа далее работает с *безымянным* файлом при помощи дескриптора *fd*. Как только файл закрывается - он будет уничтожен системой (как не имеющий имен). Такой трюк используется для создания временных рабочих файлов.

Файл можно удалить из каталога только в том случае, если данный *каталог* имеет для вас код доступа "запись". Коды доступа самого *файла* при удалении *не играют роли*.

В современных версиях **UNIX** есть системный вызов **rename**, который делает то же самое, что и написанная нами одноименная функция.

4.6. Существование альтернативных имен у файла позволяет нам решить некоторые проблемы, которые могут возникнуть при использовании чужой программы, от которой нет исходного текста (которую нельзя поправить). Пусть программа выдает некоторую информацию в файл *zz.out* (и это имя жестко зафиксировано в ней, и не задается через аргументы программы):

```
/* Эта программа компилируется в a.out */
main(){
    int fd = creat("zz.out", 0644);
    write(fd, "It's me\n", 8);
}
```

Мы же хотим получить вывод на терминал, а не в файл. Очевидно, мы должны сделать файл *zz.out* синонимом устройства **/dev/tty** (см. конец этой главы). Это можно сделать командой **ln**:

```
$ rm zz.out ; ln /dev/tty zz.out
$ a.out
$ rm zz.out
```

или программно:

```
/* Эта программа компилируется в start */
/* и вызывается  вместо a.out          */
#include <stdio.h>
main(){
   unlink("zz.out");
   link("/dev/tty", "zz.out");
     if( !fork()){ execl("a.out", NULL); }
     else wait(NULL);
   unlink("zz.out");
}
```

(про **fork**, **exec**, **wait** смотри в главе про **UNIX**).

Еще один пример: программа **a.out** желает запустить программу */usr/bin/vi* (смотри про функцию **system**() сноску через несколько страниц):

```
main(){
   ... system("/usr/bin/vi xx.c"); ...
}
```

На вашей же машине редактор **vi** помещен в */usr/local/bin/vi*. Тогда вы просто создаете альтернативное имя этому редактору:

```
$ ln /usr/local/bin/vi /usr/bin/vi
```

Помните, что альтернативное имя файлу можно создать лишь в *той же* файловой системе, где содержится исходное имя. В семействе **BSD** † это ограничение можно обойти, создав "символьную ссылку" вызовом

```
symlink(link_to_filename,link_file_name_to_be_created);
```

Символьная ссылка - это файл, содержащий имя другого файла (или каталога). Система не производит автоматический подсчет числа таких ссылок, поэтому возможны "висячие" ссылки - указывающие на уже удаленный файл. Прочесть содержимое файла-ссылки можно системным вызовом

```
char linkbuf[ MAXPATHLEN + 1]; /* куда поместить ответ */
int len = readlink(pathname, linkbuf, sizeof linkbuf);
linkbuf[len] = '\0';
```

Системный вызов **stat** автоматически разыменовывает символьные ссылки и выдает информацию про указуемый файл. Системный вызов **lstat** (аналог **stat** за исключением названия) выдает информацию про саму ссылку (тип файла **S_IFLNK**). Коды доступа к ссылке не имеют никакого значения для системы, существенны только коды доступа самого указуемого файла.

Еще раз: символьные ссылки удобны для указания файлов и каталогов на другом диске. Пусть у вас не помещается на диск каталог /opt/wawa. Вы можете разместить каталог wawa на диске USR: /usr/wawa. После чего создать символьную ссылку из /opt:

```
ln -s /usr/wawa /opt/wawa
```

чтобы программы видели этот каталог под его прежним именем /opt/wawa.

Еще раз:

hard link
- то, что создается системным вызовом **link**, имеет тот же I-node (индексный узел, паспорт), что и исходный файл. Это просто альтернативное имя файла, учитываемое в поле *di_nlink* в I-node.

symbolic link
- создается вызовом **symlink**. Это отдельный самостоятельный файл, с собственным I-node. Правда, коды доступа к этому файлу не играют никакой роли; значимы только коды доступа указуемого файла.

4.7. Напишите программу, которая находит в файле символ **@** и выдает файл с этого места дважды. Указание: для запоминания позиции в файле используйте вызов **lseek**() - позиционирование указателя чтения/записи:

---

† **BSD** - семейство **UNIX**-ов из University of California, Berkley. **Berkley Software Distribution**.

```
long offset, lseek();
   ...
/* Узнать текущую позицию чтения/записи:
 * сдвиг на 0 от текущей позиции. lseek вернет новую
 * позицию указателя (в байтах от начала файла). */
offset = lseek(fd, 0L, 1);  /* ftell(fp) */
```

А для возврата в эту точку:

```
   lseek(fd, offset, 0);    /* fseek(fp, offset, 0) */
```

По поводу **lseek** надо помнить такие вещи:

- **lseek**(*fd*, *offset*, *whence*) устанавливает указатель чтения/записи на расстояние *offset* байт

```
при whence:
  0    от начала файла     RWptr  = offset;
  1    от текущей позиции  RWptr += offset;
  2    от конца файла      RWptr  = длина_файла + offset;

Эти значения whence можно обозначать именами:
#include <stdio.h>
  0    это   SEEK_SET
  1    это   SEEK_CUR
  2    это   SEEK_END
```

- Установка указателя чтения/записи - это *виртуальная* операция, т.е. реального подвода магнитных головок и вообще обращения к диску она не вызывает. Реальное движение головок к нужному месту диска произойдет только при операциях чтения/записи **read**()/**write**(). Поэтому **lseek**() - *дешевая* операция.
- **lseek**() возвращает новую позицию указателя чтения/записи *RWptr* относительно *начала файла* (long смещение в байтах). Помните, что если вы используете это значение, то вы должны предварительно описать **lseek** как функцию, возвращающую длинное целое: **long lseek**();
- Аргумент *offset* должен иметь тип **long** (не ошибитесь!).
- Если поставить указатель за конец файла (это допустимо!), то операция записи **write**() сначала заполнит байтом '\0' все пространство от конца файла до позиции указателя; операция **read**() при попытке чтения из-за конца файла вернет "прочитано 0 байт". Попытка поставить указатель перед началом файла вызовет ошибку.
- Вызов **lseek**() неприменим к pipe и FIFO-файлам, поэтому попытка сдвинуться на 0 байт выдаст ошибку:

```
/* это стандартная функция */
int isapipe(int fd){
     extern errno;
     return (lseek(fd, 0L, SEEK_CUR) < 0 && errno == ESPIPE);
}
```

выдает "истину", если *fd* - дескриптор "трубы"(pipe).

4.8. Каков будет эффект следующей программы?

```
int fd = creat("aFile", 0644); /* creat создает файл
    открытый на запись, с доступом rw-r--r-- */
write(fd, "begin", 5 );
lseek(fd, 1024L * 1000, 0);
write(fd, "end", 3 );
close(fd);
```

Напомним, что при записи в файл, его длина *автоматически* увеличивается, когда мы записываем информацию за прежним концом файла. Это вызывает отведение места на диске для хранения новых данных (порциями, называемыми **блоками** - размером от 1/2 до 8 Кб в разных версиях). Таким образом, размер файла ограничен только наличием свободных блоков на диске.

В нашем примере получится файл длиной 1024003 байта. Будет ли он занимать на диске 1001 блок (по 1 Кб)?

В системе **UNIX** - нет! Вот кое-что про механику выделения блоков:

- Блоки располагаются на диске не обязательно подряд - у каждого файла есть специальным образом организованная таблица адресов его блоков.
- Последний блок файла может быть занят не целиком (если длина файла не кратна размеру блока), тем не менее число блоков у файла всегда *целое* (кроме семейства **BSD**, где блок может делиться на фрагменты, принадлежащие разным файлам). Операционная система в каждый момент времени знает длину файла с точностью до одного байта и не позволяет нам "заглядывать" в остаток блока, пока при своем "росте" файл не займет эти байты.

• Блок на диске физически выделяется лишь *после операции записи* в этот блок.

В нашем примере: при создании файла его размер 0, и ему выделено 0 блоков. При первой записи файлу будет выделен один блок (логический блок номер 0 для файла) и в его начало запишется "begin". Длина файла станет равна 5 (остаток блока - 1019 байт - не используется и файлу логически не принадлежит!). Затем **lseek** поставит указатель записи далеко за конец файла и **write** запишет в 1000-ый блок слово "end". 1000-ый блок будет выделен на диске. В этот момент у файла "возникнут" и все промежуточные блоки 1..999. Однако они будут только "числиться за файлом", но на диске отведены *не будут* (в таблице блоков файла это обозначается адресом 0)! При чтении из них будут читаться байты '\0'. Это так называемая "*дырка*" в файле. Файл имеет размер 1024003 байта, но на диске занимает всего 2 блока (на самом деле чуть больше, т.к. часть таблицы блоков файла тоже находится в специальных блоках файла). Блок из "дырки" станет реальным, если в него что-нибудь записать.

Будьте готовы к тому, что "размер файла" (который, кстати, можно узнать системным вызовом **stat**) - это в **UNIX** не то же самое, что "место, занимаемое файлом на диске".

4.9. Найдите ошибки:

```
FILE *fp;
    ...
fp = open( "файл", "r" ); /* открыть */
close(fp);                /* закрыть */
```

Ответ: используется системный вызов **open**() вместо функции **fopen**(); а также **close** вместо **fclose**, а их форматы (и результат) различаются! Следует четко различать две существующие в Си модели обмена с файлами: через системные вызовы: **open**, **creat**, **close**, **read**, **write**, **lseek**; и через библиотеку буферизованного обмена **stdio**: **fopen**, **fclose**, **fread**, **fwrite**, **fseek**, **getchar**, **putchar**, **printf**, и.т.д. В первой из них обращение к файлу происходит по целому *fd* - дескриптору файла, а во втором - по указателю **FILE** **fp* - указателю на файл. Это параллельные механизмы (по своим возможностям), хотя второй является просто надстройкой над первым. Тем не менее, лучше их не смешивать.

4.10. Доступ к диску (чтение/запись) гораздо (на несколько порядков) медленнее, чем доступ к данным в оперативной памяти. Кроме того, если мы читаем или записываем файл при помощи системных вызовов маленькими порциями (по 1-10 символов)

```
char c;
while( read(0, &c, 1)) ... ; /* 0 - стандартный ввод */
```

то мы проигрываем еще в одном: каждый системный вызов - это обращение к ядру операционной системы. При каждом таком обращении происходит довольно *большая* дополнительная работа (смотри главу "Взаимодействие с **UNIX**"). При этом накладные расходы на такое посимвольное чтение файла могут значительно превысить полезную работу.

Еще одной проблемой является то, что системные вызовы работают с файлом как с неструктурированным массивом байт; тогда как человеку часто удобнее представлять, что файл поделен на строки, содержащие читабельный текст, состоящий лишь из обычных печатных символов (текстовый файл).

Для решения этих двух проблем была построена специальная библиотека функций, названная **stdio** - "стандартная библиотека ввода/вывода" (*standard input/output library*). Она является частью библиотеки */lib/libc.a* и представляет собой *надстройку* над системными вызовами (т.к. в конце концов все ее функции время от времени обращаются к системе, но гораздо *реже*, чем если использовать сисвызовы непосредственно). Небезызвестная директива **#include** <*stdio.h*> включает в нашу программу файл с объявлением форматов данных и констант, используемых этой библиотекой.

Библиотеку **stdio** можно назвать библиотекой буферизованного обмена, а также библиотекой работы с текстовыми файлами (т.е. имеющими разделение на строки), поскольку для оптимизации обменов с диском (для уменьшения числа обращений к нему и тем самым сокращения числа системных вызовов) эта библиотека вводит *буферизацию*, а также предоставляет несколько функций для работы со строчно-организованными файлами.

Связь с файлом в этой модели обмена осуществляется уже не при помощи целого числа - *дескриптора файла* (*file descriptor*), а при помощи адреса "связной" структуры **FILE**. Указатель на такую структуру условно называют *указателем на файл* (*file pointer*)†. Структура **FILE** содержит в себе:

• дескриптор *fd* файла для обращения к системным вызовам;

---

† Это не та "связующая" структура **file** в ядре, про которую шла речь выше, а ЕЩЕ одна - в памяти самой программы.

- указатель на буфер, размещенный в памяти программы;
- указатель на текущее место в буфере, откуда надо выдать или куда записать очередной символ; этот указатель продвигается при каждом вызове **getc** или **putc**;
- счетчик оставшихся в буфере символов (при чтении) или свободного места (при записи);
- режимы открытия файла (чтение/запись/чтение+запись) и текущее состояние файла. Одно из состояний - при чтении файла был достигнут его конец£;
- способ буферизации;

Предусмотрено несколько стандартных структур **FILE**, указатели на которые называются *stdin*, *stdout* и *stderr* и связаны с дескрипторами 0, 1, 2 соответственно (стандартный ввод, стандартный вывод, стандартный вывод ошибок). Напомним, что эти каналы открыты неявно (автоматически) и, если не перенаправлены, связаны с вводом с клавиатуры и выводом на терминал.

Буфер в оперативной памяти нашей программы создается (функцией **malloc**) при открытии файла при помощи функции **fopen**(). После открытия файла все операции обмена с файлом происходят не по 1 байту, а большими порциями размером с буфер - обычно по 512 байт (константа **BUFSIZ**).

При чтении символа

```
int c; FILE *fp = ... ;
c = getc(fp);
```

в буфер считывается **read**-ом из файла порция информации, и **getc** выдает ее первый байт. При последующих вызовах **getc** выдаются следующие байты из буфера, а обращений к диску уже не происходит! Лишь когда буфер будет исчерпан - произойдет очередное чтение с диска. Таким образом, информация читается из файла с опережением, заранее наполняя буфер; а по требованию выдается уже из буфера. Если мы читаем 1024 байта из файла при помощи **getc**(), то мы 1024 раза вызываем эту функцию, но всего 2 раза системный вызов **read** - для чтения двух порций информации из файла, каждая - по 512 байт.

При записи

```
char c; FILE *fp = ... ;
putc(c, fp);
```

выводимые символы накапливаются в буфере. Только когда в нем окажется большая порция информации, она за одно обращение **write** записывается на диск. Буфер записи "выталкивается" в файл в таких случаях:

- буфер заполнен (содержит **BUFSIZ** символов).
- при закрытии файла (**fclose** или **exit** ††).
- при вызове функции **fflush** (см. ниже).
- в специальном режиме - после помещения в буфер символа '**n**' (см. ниже).
- в некоторых версиях - перед любой операцией чтения из канала *stdin* (например, при вызове **gets**), при условии, что *stdout* буферизован построчно (режим **_IOLBF**, смотри ниже), что по-умолчанию так и есть.

Приведем упрощенную схему, поясняющую взаимоотношения основных функций и макросов из **stdio** (кто кого вызывает). Далее *s* означает строку, *c* - символ, *fp* - указатель на структуру **FILE** ££. Функции, работающие со строками, в цикле вызывают посимвольные операции. Обратите внимание, что в конце концов все функции обращаются к *системным вызовам* **read** и **write**, осуществляющим ввод/вывод низкого уровня.

Системные вызовы далее обозначены **жирно**, макросы - *курсивом*.

£ Проверить это состояние позволяет макрос **feof**(*fp*); он истинен, если конец был достигнут, ложен - если еще нет.

†† При выполнении вызова завершения программы **exit**(); все открытые файлы автоматически закрываются.

†† Обозначения *fd* для дескрипторов и *fp* для указателей на файл прижились и их следует придерживаться. Если переменная должна иметь более мнемоничное имя - следует писать так: *fp_output*, *fd_input* (а не просто *fin*, *fout*).

Открыть файл, создать буфер:

```
#include <stdio.h>
FILE *fp = fopen(char *name, char *rwmode);
               |  вызывает
               V
   int fd = open (char *name, int irwmode);
Если открываем на запись и файл не существует (fd < 0),
то создать файл вызовом:
       fd = creat(char *name, int accessmode);
       fd будет открыт для записи в файл.
```

По умолчанию fopen() использует для **creat** коды доступа accessmode равные 0666 (rw-rw-rw-).

Соответствие аргументов fopen и **open**:

rwmode	irwmode
"r"	**O_RDONLY**
"w"	**O_WRONLY**|**O_CREAT** |**O_TRUNC**
"r+"	**O_RDWR**
"w+"	**O_RDWR** |**O_CREAT** |**O_TRUNC**
"a"	**O_WRONLY**|**O_CREAT** |**O_APPEND**
"a+"	**O_RDWR** |**O_CREAT** |**O_APPEND**

Для **r**, **r+** файл уже должен существовать, в остальных случаях файл создается, если его не было.

Если fopen() не смог открыть (или создать) файл, он возвращает значение **NULL**:

```
if((fp = fopen(name, rwmode)) == NULL){ ...неудача... }
```

Итак, схема:

Закрыть файл, освободить память выделенную под буфер:

```
fclose(fp) ---> close(fd);
```

И чуть в стороне - функция позиционирования:

```
fseek(fp,long_off,whence) ---> lseek(fd,long_off,whence);
```

Функции **_flsbuf** и **_filbuf** - внутренние для **stdio**, они как раз сбрасывают буфер в файл либо читают новый буфер из файла.

По указателю *fp* можно узнать дескриптор файла:

```
int fd = fileno(fp);
```

Это макроопределение просто выдает поле из структуры *FILE*. Обратно, если мы открыли файл **open**-ом, мы можем ввести буферизацию этого канала:

```
int fd = open(name, O_RDONLY);  /* или creat() */
        ...
FILE *fp = fdopen(fd, "r");
```

(здесь надо вновь указать КАК мы открываем файл, что должно соответствовать режиму открытия **open**-ом). Теперь можно работать с файлом через *fp*, а не *fd*.

В приложении имеется текст, содержащий упрощенную реализацию главных функций из библиотеки **stdio**.

4.11. Функция **ungetc**(*c*,*fp*) "возвращает" прочитанный байт в файл. На самом деле байт возвращается в буфер, поэтому эта операция неприменима к небуферизованным каналам. Возврат соответствует сдвигу указателя чтения из буфера (который увеличивается при **getc**()) на 1 позицию *назад*. Вернуть можно только *один символ подряд* (т.е. перед следующим **ungetc**-ом должен быть хоть один **getc**), поскольку в противном случае можно сдвинуть указатель за начало буфера и, записывая туда символ *c*, разрушить память программы.

```
while((c = getchar()) != '+' );
/* Прочли '+' */   ungetc(c ,stdin);
/* А можно заменить этот символ на другой! */
c = getchar();     /* снова прочтет '+' */
```

4.12. Очень часто делают ошибку в функции **fputc**, путая порядок ее аргументов. Так ничего не стоит написать:

```
FILE *fp = ......;
fputc( fp, '\n' );
```

Запомните навсегда!

```
int fputc( int c,  FILE *fp );
```

указатель файла идет вторым! Существует также макроопределение

```
putc( c, fp );
```

Оно ведет себя как и функция **fputc**, но не может быть передано в качестве аргумента в функцию:

```
#include <stdio.h>
putNtimes(    fp,     c,     n,       f       )
        FILE *fp; int c; int n; int (*f)();
{       while( n > 0 ){ (*f)( c, fp ); n--; }}

             возможен вызов
        putNtimes( fp, 'a', 3, fputc );
             но недопустимо
        putNtimes( fp, 'a', 3, putc );
```

Тем не менее всегда, где возможно, следует пользоваться макросом - он работает быстрее. Аналогично, есть функция **fgetc**(fp) и макрос **getc**(fp).

Отметим еще, что **putchar** и **getchar** это тоже всего лишь макросы

```
#define putchar(c)      putc((c), stdout)
#define getchar()       getc(stdin)
```

4.13. Известная вам функция **printf** также является частью библиотеки *stdio*. Она входит в семейство функций:

```
FILE   *fp; char bf[256];
fprintf(fp, fmt, ... );
 printf(    fmt, ... );
sprintf(bf, fmt, ... );
```

Первая из функций форматирует свои аргументы в соответствии с форматом, заданным строкой *fmt* (она содержит форматы в виде **%**-ов) и записывает строку-результат посимвольно (вызывая **putc**) в файл *fp*. Вторая - это всего-навсего **fprintf** с каналом *fp* равным *stdout*. Третяя выдает сформатированную строку не в файл, а записывает ее в массив *bf*. В конце строки **sprintf** добавляет нулевой байт '\0' - признак конца.

Для чтения данных по формату используются функции семейства

```
fscanf(fp, fmt, /* адреса арг-тов */...);
 scanf(    fmt, ... );
sscanf(bf, fmt, ... );
```

Функции **fprintf** и **fscanf** являются наиболее мощным средством работы с текстовыми файлами (содержащими *изображение* данных в виде печатных символов).

4.14. Текстовые файлы (имеющие строчную организацию) хранятся на диске как линейные массивы байт. Для разделения строк в них используется символ '**\n**'. Так, например, текст

```
стр1
стрк2
кнц
```

хранится как массив

```
с т р 1 \n с т р к 2 \n к н ц     длина=14 байт
          !
   указатель чтения/записи (read/write pointer RWptr)
   (расстояние в байтах от начала файла)
```

При выводе на экран дисплея символ **\n** преобразуется драйвером терминалов в последовательность **\r\n**, которая возвращает курсор в начало строки ('**\r**') и опускает курсор на строку вниз ('**\n**'), то есть курсор переходит в начало следующей строки.

В **MS DOS** строки в файле на диске разделяются двумя символами **\r\n** и при выводе на экран никаких преобразований не делается†. Зато библиотечные функции языка Си преобразуют эту последовательность при чтении из файла в **\n**, а при записи в файл превращают **\n** в **\r\n**, поскольку в Си считается, что строки разделяются только **\n**. Для работы с файлом без таких преобразований, его надо открывать как "бинарный":

```
FILE *fp = fopen( имя, "rb" );  /* b - binary */
int fd   = open ( имя, O_RDONLY | O_BINARY );
```

Все нетекстовые файлы в **MS DOS** надо открывать именно так, иначе могут произойти разные неприятности. Например, если мы программой копируем нетекстовый файл в текстовом режиме, то одиночный символ **\n** будет считан в программу как **\n**, но записан в новый файл как пара **\r\n**. Поэтому новый файл будет отличаться от оригинала (что для файлов с данными и программ совершенно недопустимо!).

Задание: напишите программу подсчета строк и символов в файле. Указание: надо подсчитать число символов '**\n**' в файле и учесть, что *последняя* строка файла может не иметь этого символа на конце. Поэтому если последний символ файла (тот, который вы прочитаете самым последним) не есть '**\n**', то добавьте к счетчику строк 1.

4.15. Напишите программу подсчета количества вхождений каждого из символов алфавита в файл и печатающую результат в виде таблицы в 4 колонки. (Указание: заведите массив из 256 счетчиков. Для больших файлов счетчики должны быть типа long).

4.16. Почему вводимый при помощи функций **getchar**() и **getc**(*fp*) символ должен описываться типом **int** а не **char**?

Ответ: функция **getchar**() сообщает о конце файла тем, что возвращает значение **EOF** (*end of file*), равное целому числу (-1). Это НЕ символ кодировки **ASCII**, поскольку **getchar**() может прочесть из файла любой символ кодировки (кодировка содержит символы с кодами 0...255), а специальный признак не должен совпадать ни с одним из хранимых в файле символов. Поэтому для его хранения требуется больше одного байта (нужен хотя бы еще 1 бит). Проверка на конец файла в программе обычно выглядит так:

---

† Управляющие символы имеют следующие значения:

```
'\n' - '\012' (10)  line feed
'\r' - '\015' (13)  carriage return
'\t' - '\011'  (9)  tab
'\b' - '\010'  (8)  backspace
'\f' - '\014' (12)  form feed
'\a' - '\007'  (7)  audio bell (alert)
'\0' - 0.           null byte
```

```
        ...
while((ch = getchar()) != EOF ){
        putchar(ch);
        ...
}
```

- Пусть *ch* имеет тип **unsigned char**. Тогда *ch* всегда лежит в интервале 0...255 и НИКОГДА не будет равно (-1). Даже если **getchar**() вернет такое значение, оно будет приведено к типу **unsigned char** обрубанием и станет равным 255. При сравнении с целым (-1) оно расширится в **int** добавлением нулей слева и станет равно 255. Таким образом, наша программа никогда не завершится, т.к. вместо признака конца файла она будет читать символ с кодом 255 (255 != -1).
- Пусть *ch* имеет тип **signed char**. Тогда перед сравнением с целым числом **EOF** байт *ch* будет приведен к типу **signed int** при помощи расширения знакового бита (7-ого). Если **getchar** вернет значение (-1), то оно будет сначала в присваивании значения байту *ch* обрублено до типа **char**: 255; но в сравнении с **EOF** значение 255 будет приведено к типу **int** и получится (-1). Таким образом, истинный конец файла будет обнаружен. Но теперь, если из файла будет прочитан настоящий символ с кодом 255, он будет приведен в сравнении к целому значению (-1) и будет также воспринят как конец файла. Таким образом, если в нашем файле окажется символ с кодом 255, то программа воспримет его как фальшивый конец файла и оставит весь остаток файла необработанным (а в нетекстовых файлах такие символы - не редкость).
- Пусть *ch* имеет тип **int** или **unsigned int** (больше 8 бит). Тогда все корректно.

Отметим, что в **UNIX** признак конца файла в самом файле физически НЕ ХРАНИТСЯ. Система в любой момент времени знает длину файла с точностью до одного байта; признак **EOF** вырабатывается стандартными функциями тогда, когда обнаруживается, что указатель чтения достиг конца файла (то есть позиция чтения стала равной длине файла - последний байт уже прочитан).

В **MS DOS** же в текстовых файлах признак конца (**EOF**) хранится явно и обозначается символом **CTRL/Z**. Поэтому, если программным путем записать куда-нибудь в середину файла символ **CTRL/Z**, то некоторые программы перестанут "видеть" остаток файла после этого символа!

Наконец отметим, что разные функции при достижении конца файла выдают разные значения: **scanf**, **fscanf**, **fgetc**, **getc**, **getchar** выдают **EOF**, **read** - выдает **0**, а **gets**, **fgets** - **NULL**.

4.17. Напишите программу, которая запрашивает ваше имя и приветствует вас. Для ввода имени используйте стандартные библиотечные функции

```
 gets(s);
fgets(s,slen,fp);
```

В чем разница?

Ответ: функция **gets**() читает строку (завершающуюся '**\n**') из канала *fp==stdin*. Она не контролирует длину буфера, в которую считывается строка, поэтому если строка окажется слишком длинной - ваша программа повредит свою память (и аварийно завершится). Единственный возможный совет - делайте буфер *достаточно большим* (очень туманное понятие!), чтобы вместить максимально возможную (длинную) строку.

Функция **fgets**() контролирует длину строки: если строка на входе окажется длиннее, чем *slen* символов, то остаток строки не будет прочитан в буфер *s*, а будет оставлен "на потом". Следующий вызов **fgets** прочитает этот сохраненный остаток. Кроме того **fgets**, в отличие от **gets**, не обрубает символ '**\n**' на конце строки, что доставляет нам дополнительные хлопоты по его уничтожению, поскольку в Си "нормальные" строки завершаются просто '**\0**', а не "**\n\0**".

```
char buffer[512]; FILE *fp = ... ; int len;
        ...
while(fgets(buffer, sizeof buffer, fp)){
  if((len = strlen(buffer)) && buffer[len-1] == '\n')
  /* @ */                     buffer[--len] =  '\0';
  printf("%s\n", buffer);
}
```

Здесь *len* - длина строки. Если бы мы выбросили оператор, помеченный '@', то **printf** печатал бы текст через строку, поскольку выдавал бы код '**\n**' дважды - из строки *buffer* и из формата "%s\n".

Если в файле больше нет строк (файл дочитан до конца), то функции **gets** и **fgets** возвращают значение **NULL**. Обратите внимание, что **NULL**, а не **EOF**. Пока файл не дочитан, эти функции возвращают свой первый аргумент - адрес буфера, в который была записана очередная строка файла.

Фрагмент для обрубания символа перевода строки может выглядеть еще так:

```
#include <stdio.h>
#include <string.h>
char buffer[512]; FILE *fp = ... ;
      ...
while(fgets(buffer, sizeof buffer, fp) != NULL){
  char *sptr;
  if(sptr = strchr(buffer, '\n'))
    *sptr = '\0';
  printf("%s\n", buffer);
}
```

4.18. В чем отличие **puts**(*s*); и **fputs**(*s*,*fp*); ?

Ответ: **puts** выдает строку *s* в канал *stdout*. При этом **puts** выдает сначала строку *s*, а затем - дополнительно - символ перевода строки '**\n**'. Функция же **fputs** символ перевода строки не добавляет. Упрощенно:

```
fputs(s, fp) char *s; FILE *fp;
{ while(*s) putc(*s++, fp); }
puts(s) char *s;
{ fputs(s, stdout); putchar('\n'); }
```

4.19. Найдите ошибки в программе:

```
#include <stdio.h>
main() {
    int fp;
    int i;
    char str[20];

    fp = fopen("файл");
    fgets(stdin, str, sizeof str);
    for( i = 0; i < 40; i++  );
         fputs(fp, "Текст, выводимый в файл:%s",str );
    fclose("файл");
}
```

Мораль: надо быть внимательнее к формату вызова и смыслу библиотечных функций.

4.20. Напишите программу, которая распечатывает самую длинную строку из файла ввода и ее длину.

4.21. Напишите программу, которая выдает *n*-ую строку файла. Номер строки и имя файла задаются как аргументы **main**().

4.22. Напишите программу

slice *-сКакой +сколько файл*

которая выдает *сколько* строк файла *файл*, начиная со строки номер *сКакой* (нумерация строк с единицы).

```
#include <stdio.h>
#include <ctype.h>
long line, count, nline, ncount; /* нули */
char buf[512];

void main(int argc, char **argv){
  char c; FILE *fp;

  argc--; argv++;
  /* Разбор ключей */
  while((c = **argv) == '-' || c == '+'){
    long atol(), val; char *s = &(*argv)[1];
    if( isdigit(*s)){
       val = atol(s);
       if(c == '-')     nline  = val;
       else             ncount = val;
    } else fprintf(stderr,"Неизвестный ключ %s\n", s-1);
    argc--; ++argv;
```

```
  }
  if( !*argv ) fp = stdin;
  else if((fp = fopen(*argv, "r")) == NULL){
    fprintf(stderr, "Не могу читать %s\n", *argv);
    exit(1);
  }
for(line=1, count=0; fgets(buf, sizeof buf, fp); line++){
      if(line >= nline){
         fputs(buf, stdout); count++;
      }
      if(ncount && count == ncount)
         break;
  }
  fclose(fp); /* это не обязательно писать явно */
}
/* End_Of_File */
```

4.23. Составьте программу, которая распечатывает последние *n* строк файла ввода.

4.24. Напишите программу, которая делит входной файл на файлы по *n* строк в каждом.

4.25. Напишите программу, которая читает 2 файла и печатает их вперемежку: одна строка из первого файла, другая - из второго. Придумайте, как поступить, если файлы содержат разное число строк.

4.26. Напишите программу сравнения двух файлов, которая будет печатать первую из различающихся строк и позицию символа, в котором они различаются.

4.27. Напишите программу для интерактивной работы с файлом. Сначала у вас запрашивается имя файла, а затем вам выдается меню:

```
1. Записать текст в файл.
2. Дописать текст к концу файла.
3. Просмотреть файл.
4. Удалить файл.
5. Закончить работу.
```

Текст вводится в файл построчно с клавиатуры. Конец ввода - EOF (т.е. **CTRL**/**D**), либо одиночный символ '**.**' в начале строки. Выдавайте число введенных строк.

Просмотр файла должен вестись постранично: после выдачи очередной порции строк выдавайте подсказку

```
--more-- _
```

(курсор остается в той же строке и обозначен подчерком) и ожидайте нажатия клавиши. Ответ 'q' завершает просмотр. Если файл, который вы хотите просмотреть, не существует - выдавайте сообщение об ошибке.

После выполнения действия программа вновь запрашивает имя файла. Если вы ответите вводом пустой строки (сразу нажмете <**ENTER**>, то должно использоваться имя файла, введенное на предыдущем шаге. Имя файла, предлагаемое по умолчанию, принято писать в запросе в [] скобках.

```
Введите имя файла [oldfile.txt]: _
```

Когда вы научитесь работать с экраном дисплея (см. главу "Экранные библиотеки"), перепишите меню и выдачу сообщений с использованием позиционирования курсора в заданное место экрана и с выделением текста инверсией. Для выбора имени файла предложите меню: отсортированный список имен всех файлов текущего каталога (по поводу получения списка файлов см. главу про взаимодействие с **UNIX**). Просто для распечатки текущего каталога на экране можно также использовать вызов

system("ls -x");

а для считывания каталога в программу†

---

† Функция

int **system**(char **команда*);

выполняет команду, записанную в строке *команда*, вызывая для этого интерпретатор команд

/bin/sh -c "*команда*"

и возвращает код ответа этой программы. Функция **popen** (pipe open) также запускает интерпретатор команд, при этом перенаправив его стандартный вывод в трубу (pipe). Другой конец этой трубы можно читать через канал *fp*, т.е. можно прочесть в свою программу выдачу запущенной команды.

```
FILE *fp = popen("ls *.c", "r");
... fgets(...,fp); ... // в цикле, пока не EOF
pclose(fp);
```

(в этом примере читаются только имена **.c** файлов).

4.28. Напишите программу удаления *n*-ой строки из файла; вставки строки после *m*-ой. К сожалению, это возможно только путем переписывания всего файла в другое место (без ненужной строки) и последующего его переименования.

4.29. Составьте программу перекодировки текста, набитого в кодировке **КОИ-8**, в альтернативную кодировку и наоборот. Для этого следует составить таблицу перекодировки из 256 символов: *c_new*=TABLE[*c_old*]; Для решения обратной задачи используйте стандартную функцию **strchr**(). Программа читает один файл и создает новый.

4.30. Напишите программу, делящую большой файл на куски заданного размера (не в строках, а в килобайтах). Эта программа может применяться для записи слишком большого файла на дискеты (файл режется на части и записывается на несколько дискет).

```
#include <fcntl.h>
#include <stdio.h>
#define min(a,b)  (((a) < (b)) ? (a) : (b))
#define KB                  1024  /* килобайт */
#define PORTION         (20L* KB)  /* < 32768  */
long    ONEFILESIZE  = (300L* KB);
extern char    *strrchr(char *, char);
extern long     atol   (char *);
extern errno;           /* системный код ошибки  */
char    buf[PORTION];   /* буфер для копирования */

void main (int ac, char *av[]) {
    char    name[128], *s, *prog = av[0];
    int     cnt=0, done=0, fdin, fdout;
/* M_UNIX автоматически определяется
 * компилятором в UNIX */
#ifndef M_UNIX  /* т.е. MS DOS */
    extern int _fmode; _fmode = O_BINARY;
    /* Задает режим открытия и создания ВСЕХ файлов */
#endif
    if(av[1] && *av[1] == '-'){ /* размер одного куска */
        ONEFILESIZE = atol(av[1]+1) * KB; av++; ac--;
    }
    if (ac < 2){
      fprintf(stderr, "Usage: %s [-size] file\n", prog);
      exit(1);
    }
    if ((fdin = open (av[1], O_RDONLY)) < 0) {
fprintf (stderr, "Cannot read %s\n", av[1]); exit (2);
    }
    if ((s = strrchr (av[1], '.'))!= NULL) *s = '\0';
    do { unsigned long sent;
         sprintf (name, "%s.%d", av[1], ++cnt);
         if ((fdout = creat (name, 0644)) < 0) {
fprintf (stderr, "Cannot create %s\n", name); exit (3);
         }
         sent = 0L; /* сколько байт переслано */
         for(;;){ unsigned isRead, /* прочитано read-ом */
            need = min(ONEFILESIZE - sent, PORTION);
            if( need == 0 ) break;
            sent += (isRead = read (fdin, buf, need));
            errno = 0;
            if (write (fdout, buf, isRead) != isRead &&
                errno){ perror("write"); exit(4);
            } else if (isRead < need){ done++; break; }
         }
         if(close (fdout) < 0){
            perror("Мало места на диске"); exit(5);
```

```
            }
            printf("%s\t%lu байт\n", name, sent);
        } while( !done ); exit(0);
    }
```

4.31. Напишите обратную программу, которая склеивает несколько файлов в один. Это аналог команды **cat** с единственным отличием: результат выдается не в стандартный вывод, а в файл, указанный в строке аргументов последним. Для выдачи в стандартный вывод следует указать имя "-".

```
#include <fcntl.h>
#include <stdio.h>
void main (int ac, char **av){
    int     i, err = 0;    FILE *fpin, *fpout;
    if (ac < 3) {
       fprintf(stderr,"Usage: %s from... to\n", av[0]);
       exit(1);
    }
    fpout = strcmp(av[ac-1], "-")  ? /* отлично от "-" */
            fopen (av[ac-1], "wb") : stdout;
    for (i = 1; i < ac-1; i++) {
        register int c;
        fprintf (stderr, "%s\n", av[i]);
        if ((fpin = fopen (av[i], "rb")) == NULL) {
            fprintf (stderr, "Cannot read %s\n", av[i]);
            err++; continue;
        }
        while ((c = getc (fpin)) != EOF)
            putc (c, fpout);
        fclose (fpin);
    }
    fclose (fpout); exit (err);
}
```

Обе эти программы могут без изменений транслироваться и в **MS DOS** и в **UNIX**. **UNIX** просто игнорирует букву **b** в открытии файла "rb", "wb". При работе с **read** мы могли бы открывать файл как

```
#ifdef M_UNIX
# define O_BINARY       0
#endif
int fdin = open( av[1], O_RDONLY | O_BINARY);
```

4.32. Каким образом стандартный ввод переключить на ввод из заданного файла, а стандартный вывод - в файл? Как проверить, существует ли файл; пуст ли он? Как надо открывать файл для дописывания информации в конец существующего файла? Как надо открывать файл, чтобы попеременно записывать и читать тот же файл? Указание: см. **fopen**, **freopen**, **dup2**, **stat**. Ответ про перенаправления ввода:

```
      способ 1        (библиотечные функции)
#include <stdio.h>
...
freopen( "имя_файла", "r", stdin );

      способ 2        (системные вызовы)
#include <fcntl.h>
int fd;
...
fd = open( "имя_файла", O_RDONLY );
dup2 ( fd, 0 ); /* 0 - стандартный ввод    */
close( fd );    /* fd больше не нужен - закрыть
     его, чтоб не занимал место в таблице */
```

```
      способ 3        (системные вызовы)
#include <fcntl.h>
int fd;
...
fd = open( "имя_файла", O_RDONLY );
close (0);               /* 0 - стандартный ввод */
fcntl (fd, F_DUPFD, 0 ); /* 0 - стандартный ввод */
close (fd);
```

Это перенаправление ввода соответствует конструкции

```
$ a.out < имя_файла
```

написанной на командном языке СиШелл. Для перенаправления вывода замените 0 на 1, *stdin* на *stdout*, **open** на **creat**, "**r**" на "**w**".

Рассмотрим механику работы вызова **dup2** †:

```
new = open("файл1",...); dup2(new, old); close(new);

  таблица открытых
   файлов процесса
    ...##                    ##
new----##---> файл1    new---##---> файл1
       ##                    ##
old----##---> файл2    old---##     файл2
       ##                    ##
 0:до вызова     1:разрыв связи old с файл2
   dup2()      (закрытие канала old, если он был открыт)

       ##                    ##
new----##--*--> файл1   new  ##  *----> файл1
       ##  |                 ##  |
old----##--*            old--##--*
       ##                    ##
 2:установка old на файл1  3:после оператора close(new);
   на этом dup2 завершен.    дескриптор new закрыт.
```

Здесь *файл1* и *файл2* - связующие структуры "открытый файл" в ядре, о которых рассказывалось выше (в них содержатся указатели чтения/записи). После вызова **dup2** дескрипторы *new* и *old* ссылаются на общую такую структуру и поэтому имеют один и тот же R/W-указатель. Это означает, что в программе *new* и *old* являются синонимами и могут использоваться даже вперемежку:

```
dup2(new, old);
write(new, "a", 1);
write(old, "b", 1);
write(new, "c", 1);
```

запишет в *файл1* строку "abc". Программа

```
int fd;
printf( "Hi there\n");
fd = creat( "newout", 0640 );
dup2(fd, 1); close(fd);
printf( "Hey, You!\n");
```

выдаст первое сообщение на терминал, а второе - в файл *newout*, поскольку **printf** выдает данные в канал *stdout*, связанный с дескриптором 1.

4.33. Напишите программу, которая будет выдавать подряд в стандартный вывод все файлы, чьи имена указаны в аргументах командной строки. Используйте *argc* для организации цикла. Добавьте сквозную нумерацию строк и печать номера строки.

4.34. Напишите программу, распечатывающую первую директиву препроцессора, встретившуюся в файле ввода.

---

† **dup2** читается как "dup to", в английском жаргоне принято обозначать предлог "to" цифрой 2, поскольку слова "to" и "two" произносятся одинаково: "ту". "From me 2 You". Также 4 читается как "for".

```
#include <stdio.h>
char buf[512], word[] = "#";
main(){  char *s; int len = strlen(word);
  while((s=fgets(buf, sizeof buf, stdin)) &&
         strncmp(s, word, len));
  fputs(s? s: "Не найдено.\n", stdout);
}
```

4.35. Напишите программу, которая переключает свой стандартный вывод в новый файл *имяФайла* каждый раз, когда во входном потоке встречается строка вида

```
>>>имяФайла
```

Ответ:

```
#include <stdio.h>
char line[512];
main(){  FILE *fp = fopen("00", "w");
   while(gets(line) != NULL)
     if( !strncmp(line, ">>>", 3)){
        if( freopen(line+3, "a", fp) == NULL){
          fprintf(stderr, "Can't write to '%s'\n", line+3);
          fp = fopen("00", "a");
        }
     } else fprintf(fp, "%s\n", line);
}
```

4.36. Библиотека буферизованного обмена **stdio** содержит функции, подобные некоторым системным вызовам. Вот функции - аналоги **read** и **write**:

Стандартная функция **fread** из библиотеки стандартных функций Си предназначена для чтения нетекстовой (как правило) информации из файла:

```
int fread(addr, size, count, fp)
   register char *addr; unsigned size, count; FILE *fp;
{  register c; unsigned ndone=0, sz;
   if(size)
     for( ; ndone < count ; ndone++){
        sz = size;
        do{   if((c = getc(fp)) >= 0 )
                    *addr++ = c;
              else  return ndone;
        }while( --sz );
     }
   return ndone;
}
```

Заметьте, что *count* - это не количество БАЙТ (как в **read**), а количество ШТУК размером *size* байт. Функция выдает число целиком прочитанных ею ШТУК. Существует аналогичная функция **fwrite** для записи в файл. Пример:

```
#include <stdio.h>
#define MAXPTS 200
#define N      127
char filename[] = "pts.dat";
struct point { int x,y; } pts[MAXPTS], pp= { -1, -2};
main(){
   int n, i;
   FILE *fp = fopen(filename, "w");

   for(i=0; i < N; i++) /* генерация точек */
      pts[i].x = i, pts[i].y = i * i;
   /* запись массива из N точек в файл */
   fwrite((char *)pts, sizeof(struct point), N, fp);
   fwrite((char *)&pp, sizeof pp,            1, fp);

   fp = freopen(filename, "r", fp);
   /* или fclose(fp); fp=fopen(filename, "r"); */
```

```
   /* чтение точек из файла в массив */
   n = fread(pts, sizeof pts[0], MAXPTS, fp);
   for(i=0; i < n; i++)
      printf("Точка #%d(%d,%d)\n",i,pts[i].x,pts[i].y);
}
```

Файлы, созданные **fwrite**, не переносимы на машины другого типа, поскольку в них хранится не *текст*, а двоичные *данные* в формате, используемом данным процессором. Такой файл не может быть понят человеком - он не содержит *изображений* данных в виде текста, а содержит "сырые" байты. Поэтому чаще пользуются функциями работы с текстовыми файлами: **fprintf**, **fscanf**, **fputs**, **fgets**. Данные, хранимые в виде текста, имеют еще одно преимущество помимо переносимости: их легко при нужде подправить текстовым редактором. Зато они занимают больше места!

Аналогом системного вызова **lseek** служит функция **fseek**:

```
fseek(fp, offset, whence);
```

Она полностью аналогична **lseek**, за исключением возвращаемого ею значения. Она НЕ возвращает новую позицию указателя чтения/записи! Чтобы узнать эту позицию применяется специальная функция

```
long ftell(fp);
```

Она вносит поправку на положение указателя в буфере канала *fp*. **fseek** сбрасывает флаг "был достигнут конец файла", который проверяется макросом **feof**(*fp*);

4.37. Найдите ошибку в программе (программа распечатывает корневой каталог в "старом" формате каталогов - с фиксированной длиной имен):

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>

main(){
  FILE *fp;
  struct direct d;
  char buf[DIRSIZ+1]; buf[DIRSIZ] = '\0';

  fp = fopen( '/', "r" );
  while( fread( &d, sizeof d, 1, fp) == 1 ){
    if( !d.d_ino ) continue;  /* файл стерт */
    strncpy( buf, d.d_name, DIRSIZ);
    printf( "%s\n", buf );
  }
  fclose(fp);
}
```

Указание: смотри в fopen(). Внимательнее к строкам и символам! '/' и "/" - это совершенно разные вещи (хотя синтаксической ошибки нет!).

Переделайте эту программу, чтобы название каталога поступало из аргументов **main** (а если название не задано - используйте текущий каталог ".").

4.38. Функциями

```
   fputs(    строка, fp);
  printf(    формат, ...);
 fprintf(fp, формат, ...);
```

невозможно вывести строку *формат*, содержащую в середине байт '**\0**', поскольку он служит для них признаком конца строки. Однако такой байт может понадобиться в файле, если мы формируем некоторые нетекстовые данные, например управляющую последовательность переключения шрифтов для принтера. Как быть? Есть много вариантов решения. Пусть мы хотим выдать в канал *fp* последовательность из 4х байт "\033e\0\5". Мы можем сделать это посимвольно:

```
putc('\033',fp); putc('e',   fp);
putc('\000',fp); putc('\005',fp);
```

(можно просто в цикле), либо использовать один из способов:

```
fprintf( fp,         "\033e%c\5", '\0');
write  ( fileno(fp), "\033e\0\5",   4 );
fwrite ( "\033e\0\5", sizeof(char), 4, fp);
```

где 4 - количество выводимых байтов.

4.39. Напишите функции для "быстрого доступа" к строкам файла. Идея такова: сначала прочитать весь файл от начала до конца и смещения начал строк (адреса по файлу) запомнить в массив чисел типа **long** (точнее, **off_t**), используя функции **fgets**() и **ftell**(). Для быстрого чтения *n*-ой строки используйте функции **fseek**() и **fgets**().

```
#include <stdio.h>
#define MAXLINES 2000  /* Максим. число строк в файле*/
FILE *fp;              /* Указатель на файл */
int nlines;            /* Число строк в файле */
long offsets[MAXLINES];/* Адреса начал строк */
extern long ftell();/*Выдает смещение от начала файла*/
char buffer[256];      /* Буфер для чтения строк */

/* Разметка массива адресов начал строк */
void getSeeks(){
   int c;
   offsets[0] =0L;
   while((c = getc(fp)) != EOF)
     if(c =='\n') /* Конец строки - начало новой */
        offsets[++nlines] = ftell(fp);

/* Если последняя строка файла не имеет \n на конце, */
/* но не пуста, то ее все равно надо посчитать */
   if(ftell(fp) != offsets[nlines])
        nlines++;
   printf( "%d строк в файле\n", nlines);
}
char *getLine(n){  /* Прочесть строку номер n */
   fseek(fp, offsets[n], 0);
   return fgets(buffer, sizeof buffer, fp);
}

void main(){ /* печать файла задом-наперед */
   int i;
   fp = fopen("INPUT", "r"); getSeeks();
   for( i=nlines-1; i>=0; --i)
        printf( "%3d:%s", i, getLine(i));
}
```

4.40. Что будет выдано на экран в результате выполнения программы?

```
#include <stdio.h>
main(){
    printf( "Hello, " );
    printf( "sunny " );
    write( 1, "world", 5 );
}
```

Ответ: очень хочется ответить, что будет напечатано "*Hello*, *sunny world*", поскольку **printf** выводит в канал *stdout*, связанный с дескриптором 1, а дескриптор 1 связан по-умолчанию с терминалом. Увы, эта догадка верна лишь отчасти! Будет напечатано "*worldHello*, *sunny* ". Это происходит потому, что вывод при помощи функции **printf** буферизован, а при помощи сисвызова **write** - нет. **printf** помещает строку сначала в буфер канала *stdout*, затем **write** выдает свое сообщение непосредственно на экран, затем по окончании программы буфер выталкивается на экран.

Чтобы получить правильный эффект, следует перед **write**() написать вызов явного выталкивания буфера канала *stdout*:

```
fflush( stdout );
```

Еще одно возможное решение - отмена буферизации канала *stdout*: перед первым **printf** можно написать

```
setbuf(stdout, NULL);
```

Имейте в виду, что канал вывода сообщений об ошибках *stderr* не буферизован исходно, поэтому выдаваемые в него сообщения печатаются немедленно.

Мораль: надо быть очень осторожным при смешанном использовании буферизованного и небуферизованного обмена.

Некоторые каналы буферизуются так, что буфер выталкивается не только при заполнении, но и при поступлении символа '**\n**' ("построчная буферизация"). Канал *stdout* именно таков:

```
printf("Hello\n");
```

печатается сразу (т.к. **printf** выводит в *stdout* и есть '**\n**'). Включить такой режим буферизации можно так:

```
setlinebuf(fp);   или в других версиях
setvbuf(fp, NULL, _IOLBF, BUFSIZ);
```

Учтите, что любое изменение способа буферизации должно быть сделано **ДО** первого обращения к каналу!

4.41. Напишите программу, выдающую три звуковых сигнала. Гудок на терминале вызывается выдачей символа '**\7**' ('**\a**' по стандарту **ANSI**). Чтобы гудки звучали раздельно, надо делать паузу после каждого из них. (Учтите, что вывод при помощи **printf**() и **putchar**() буферизован, поэтому после выдачи каждого гудка (в буфер) надо вызывать функцию **fflush**() для сброса буфера).

Ответ:

```
Способ 1:
        register i;
        for(i=0; i<3; i++){
           putchar( '\7' ); fflush(stdout);
           sleep(1);    /* пауза 1 сек. */
        }

Способ 2:
        register i;
        for(i=0; i<3; i++){
           write(1, "\7", 1 );
           sleep(1);
        }
```

4.42. Почему задержка не ощущается?

```
printf( "Пауза...");
sleep ( 5 );            /* ждем 5 сек. */
printf( "продолжаем\n" );
```

Ответ: из-за буферизации канала *stdout*. Первая фраза попадает в буфер и, если он не заполнился, не выдается на экран. Дальше программа "молчаливо" ждет 5 секунд. Обе фразы будут выданы уже после задержки! Чтобы первый **printf**() выдал свою фразу ДО задержки, следует перед функцией **sleep**() вставить вызов **fflush**(*stdout*) для явного выталкивания буфера. Замечание: канал *stderr* не буферизован, поэтому проблему можно решить и так:

```
fprintf( stderr, "Пауза..." );
```

4.43. Еще один пример про буферизацию. Почему программа печатает EOF?

```
#include <stdio.h>
FILE *fwr, *frd;
char b[40], *s; int n = 1917;
main(){
        fwr = fopen( "aFile", "w" );
        frd = fopen( "aFile", "r" );

        fprintf( fwr, "%d: Hello, dude!", n);
        s = fgets( b, sizeof b, frd );
        printf( "%s\n", s ? s : "EOF" );
}
```

Ответ: потому что к моменту чтения буфер канала *fwr* еще не вытолкнут в файл: файл пуст! Надо вставить

```
fflush(fwr);
```

после **fprintf**(). Вот еще подобный случай:

```
FILE *fp = fopen("users", "w");
       ... fprintf(fp, ...); ...
system("sort users | uniq > 00; mv 00 users");
```

К моменту вызова команды сортировки буфер канала *fp* (точнее, последний из накопленных за время работы буферов) может быть еще не вытолкнут в файл. Следует либо закрыть файл **fclose**(*fp*)

непосредственно перед вызовом **system**, либо вставить туда же **fflush**(*fp*);

4.44. В **UNIX** многие внешние устройства (практически все!) с точки зрения программ являются просто файлами. Файлы-устройства имеют *имена*, но не занимают места на диске (не имеют блоков). Зато им соответствуют специальные программы-*драйверы* в ядре. При открытии такого файла-устройства мы на самом деле инициализируем драйвер этого устройства, и в дальнейшем он выполняет наши запросы **read**, **write**, **lseek** аппаратно-зависимым образом. Для операций, специфичных для данного устройства, предусмотрен сисвызов **ioctl** (input/output control):

```
ioctl(fd, РОД_РАБОТЫ, аргумент);
```

где *аргумент* часто бывает адресом структуры, содержащей пакет аргументов, а *РОД_РАБОТЫ* - одно из целых чисел, специфичных для данного устройства (для каждого устр-ва есть *свой собственный* список допустимых операций). Обычно *РОД_РАБОТЫ* имеет некоторое мнемоническое обозначение.

В качестве примера приведем операцию **TCGETA**, применимую только к терминалам и узнающую текущие моды драйвера терминала (см. главу "Экранные библиотеки"). То, что эта операция неприменима к другим устройствам и к обычным файлам (не устройствам), позволяет нам использовать ее для проверки - является ли открытый файл терминалом (или клавиатурой):

```
#include <termio.h>
int isatty(fd){ struct termio tt;
   return ioctl(fd, TCGETA, &tt) < 0 ? 0 : 1;
}
main(){
  printf("%s\n", isatty(0 /* STDIN */)? "term":"no"); }
```

Функция **isatty** является стандартной функцией†.

Есть "псевдоустройства", которые представляют собой драйверы логических устройств, не связанных напрямую с аппаратурой, либо связанных лишь косвенно. Примером такого устройства является *псевдотерминал* (см. пример в приложении). Наиболее употребительны два псевдоустройства:

/dev/null

Это устройство, представляющее собой "черную дыру". Чтение из него немедленно выдает признак конца файла: **read**(...)==0; а записываемая в него информация нигде не сохраняется (пропадает). Этот файл используется, например, в том случае, когда мы хотим проигнорировать вывод какой-либо программы (сообщения об ошибках, трассировку), нигде его не сохраняя. Тогда мы просто перенаправляем ее вывод в **/dev/null**:

```
$ a.out > /dev/null &
```

Еще один пример использования:

```
$ cp /dev/hd00 /dev/null
```

Содержимое всего винчестера копируется "в никуда". При этом, если на диске есть сбойные блоки - система выдает на консоль сообщения об ошибках чтения. Так мы можем быстро выяснить, есть ли на диске плохие блоки.

/dev/tty

Открытие файла с таким именем в действительности открывает для нас *управляющий терминал*, на котором запущена данная программа; даже если ее ввод и вывод были перенаправлены в какие-то другие файлы£. Поэтому, если мы хотим выдать сообщение, которое должно появиться именно на *экране*, мы должны поступать так:

---

† Заметим еще, что если дескриптор *fd* связан с терминалом, то можно узнать полное имя этого устройства вызовом стандартной функции

```
extern char  *ttyname();
char *tname = ttyname(fd);
```

Она выдаст строку, подобную "**/dev/tty***01*". Если *fd* не связан с терминалом - она вернет NULL.

£ Ссылка на управляющий терминал процесса хранится в **u-area** каждого процесса: *u_ttyp*, *u_ttyd*, поэтому ядро в состоянии определить какой *настоящий* терминал следует открыть для вас. Если разные процессы открывают **/dev/tty**, они могут открыть в итоге разные терминалы, т.е. *одно* имя приводит к *разным* устройствам! Смотри главу про **UNIX**.

```
#include <stdio.h>
void message(char *s){
  FILE *fptty = fopen("/dev/tty", "w");
  fprintf(fptty, "%s\n", s);
  fclose (fptty);
}
main(){ message("Tear down the wall!"); }
```

Это устройство доступно и для записи (на экран) и для чтения (с клавиатуры).

Файлы устройств нечувствительны к флагу открытия **O_TRUNC** - он не имеет для них смысла и просто игнорируется. Поэтому невозможно случайно уничтожить файл-устройство (к примеру */dev/tty*) вызовом
fd=**creat**("/dev/tty", 0644);
Файлы-устройства создаются вызовом **mknod**, а уничтожаются обычным **unlink**-ом. Более подробно про это - в главе "Взаимодействие с **UNIX**".

4.45. Эмуляция основ библиотеки STDIO, по мотивам 4.2 BSD.

```
#include <fcntl.h>
#define BUFSIZ  512        /* стандартный размер буфера */
#define _NFILE   20
#define EOF    (-1)        /* признак конца файла     */
#define NULL   ((char *) 0)

#define IOREAD  0x0001     /* для чтения              */
#define IOWRT   0x0002     /* для записи              */
#define IORW    0x0004     /* для чтения и записи     */
#define IONBF   0x0008     /* не буферизован          */
#define IOTTY   0x0010     /* вывод на терминал       */
#define IOALLOC 0x0020     /* выделен буфер malloc-ом */
#define IOEOF   0x0040     /* достигнут конец файла   */
#define IOERR   0x0080     /* ошибка чтения/записи    */

extern char *malloc(); extern long lseek();
typedef unsigned char uchar;

uchar sibuf[BUFSIZ], sobuf[BUFSIZ];

typedef struct _iobuf {
  int cnt;                  /* счетчик */
  uchar *ptr, *base;        /* указатель в буфер и на его начало */
  int bufsiz, flag, file;   /* размер буфера, флаги, дескриптор  */
} FILE;

FILE iob[_NFILE] = {
  { 0, NULL, NULL, 0, IOREAD,       0 },
  { 0, NULL, NULL, 0, IOWRT|IOTTY,  1 },
  { 0, NULL, NULL, 0, IOWRT|IONBF,  2 },
};
```

```
#define stdin        (&iob[0])
#define stdout       (&iob[1])
#define stderr       (&iob[2])
#define putchar(c)   putc((c), stdout)
#define getchar()    getc(stdin)
#define fileno(fp)   ((fp)->file)
#define feof(fp)     (((fp)->flag & IOEOF) != 0)
#define ferror(fp)   (((fp)->flag & IOERR) != 0)
#define clearerr(fp) ((void) ((fp)->flag &= ~(IOERR | IOEOF)))

#define getc(fp)    (--(fp)->cnt < 0 ? \
        filbuf(fp) : (int) *(fp)->ptr++)
#define putc(x, fp) (--(fp)->cnt < 0 ? \
        flsbuf((uchar) (x), (fp)) :    \
        (int) (*(fp)->ptr++ = (uchar) (x)))

int fputc(int c, FILE *fp){ return putc(c, fp); }
int fgetc(       FILE *fp){ return getc(fp);    }

/* Открытие файла */
FILE *fopen(char *name, char *how){
   register FILE *fp; register i, rw;
   for(fp = iob, i=0; i < _NFILE; i++, fp++)
        if(fp->flag == 0) goto found;
   return NULL; /* нет свободного слота */
found:
   rw = how[1] == '+';
   if(*how == 'r'){
      if((fp->file = open(name, rw ? O_RDWR:O_RDONLY)) < 0)
          return NULL;
      fp->flag = IOREAD;
   } else {
      if((fp->file = open(name, (rw ? O_RDWR:O_WRONLY)| O_CREAT |
         (*how == 'a' ? O_APPEND : O_TRUNC), 0666 )) < 0)
           return NULL;
      fp->flag = IOWRT;
   }
   if(rw) fp->flag = IORW;
   fp->bufsiz = fp->cnt = 0; fp->base = fp->ptr = NULL;
   return fp;
}

/* Принудительный сброс буфера */
void fflush(FILE *fp){
   uchar *base; int full= 0;
   if((fp->flag & (IONBF|IOWRT)) == IOWRT &&
      (base = fp->base) != NULL && (full=fp->ptr - base) > 0){
        fp->ptr = base; fp->cnt = fp->bufsiz;
        if(write(fileno(fp), base, full) != full)
           fp->flag |= IOERR;
   }
}

/* Закрытие файла */
void fclose(FILE *fp){
   if((fp->flag & (IOREAD|IOWRT|IORW)) == 0 ) return;
   fflush(fp);
   close(fileno(fp));
   if(fp->flag  & IOALLOC) free(fp->base);
   fp->base = fp->ptr = NULL;
   fp->cnt  = fp->bufsiz = fp->flag = 0; fp->file = (-1);
}
```

```
/* Закрытие файлов при exit()-е */
void _cleanup(){
   register i;
   for(i=0; i < _NFILE; i++)
       fclose(iob + i);
}

/* Завершить текущий процесс */
void exit(uchar code){
   _cleanup();
   _exit(code); /* Собственно системный вызов */
}

/* Прочесть очередной буфер из файла */
int filbuf(FILE *fp){
   static uchar smallbuf[_NFILE];

   if(fp->flag & IORW){
      if(fp->flag & IOWRT){ fflush(fp); fp->flag &= ~IOWRT; }
      fp->flag |=   IOREAD;   /* операция чтения */
   }
   if((fp->flag & IOREAD) == 0 || feof(fp)) return EOF;

   while( fp->base == NULL )   /* отвести буфер */
      if( fp->flag & IONBF ){  /* небуферизованный */
          fp->base = &smallbuf[fileno(fp)];
          fp->bufsiz = sizeof(uchar);
      } else if( fp == stdin ){ /* статический буфер */
          fp->base   =        sibuf;
          fp->bufsiz = sizeof(sibuf);
      } else if((fp->base = malloc(fp->bufsiz = BUFSIZ)) == NULL)
           fp->flag |= IONBF;   /* не будем буферизовать */
      else fp->flag |= IOALLOC; /* буфер выделен         */

   if( fp == stdin && (stdout->flag & IOTTY)) fflush(stdout);
   fp->ptr = fp->base;        /* сбросить на начало буфера */

   if((fp->cnt = read(fileno(fp), fp->base, fp->bufsiz)) == 0 ){
       fp->flag |= IOEOF; if(fp->flag & IORW) fp->flag &= ~IOREAD;
       return EOF;
   } else if( fp->cnt < 0 ){
       fp->flag |= IOERR; fp->cnt = 0; return EOF;
   }
   return getc(fp);
}
```

```
/* Вытолкнуть очередной буфер в файл */
int flsbuf(int c, FILE *fp){
   uchar *base; int full, cret = c;

   if( fp->flag & IORW ){
       fp->flag &= ~(IOEOF|IOREAD);
       fp->flag |=   IOWRT;  /* операция записи */
   }
   if((fp->flag & IOWRT) == 0) return EOF;
tryAgain:
   if(fp->flag & IONBF){ /* не буферизован */
      if(write(fileno(fp), &c, 1) != 1)
        { fp->flag |= IOERR; cret=EOF; }
      fp->cnt = 0;
   } else {   /* канал буферизован */
      if((base = fp->base) == NULL){ /* буфера еще нет */
          if(fp == stdout){
              if(isatty(fileno(stdout))) fp->flag |=  IOTTY;
              else                       fp->flag &= ~IOTTY;
              fp->base = fp->ptr = sobuf; /* статический буфер */
              fp->bufsiz =  sizeof(sobuf);
              goto tryAgain;
          }
          if((base = fp->base = malloc(fp->bufsiz = BUFSIZ))== NULL){
                 fp->bufsiz = 0; fp->flag |= IONBF;  goto tryAgain;
          } else                 fp->flag |= IOALLOC;
      } else if ((full = fp->ptr - base) > 0)
          if(write(fileno(fp), fp->ptr = base, full) != full)
          { fp->flag |= IOERR; cret = EOF;                  }
      fp->cnt = fp->bufsiz - 1;
      *base++ = c;
      fp->ptr = base;
   }
   return cret;
}

/* Вернуть символ в буфер */
int ungetc(int c, FILE *fp){
   if(c == EOF || fp->flag & IONBF || fp->base == NULL) return EOF;
   if((fp->flag & IOREAD)==0 || fp->ptr <= fp->base)
      if(fp->ptr == fp->base && fp->cnt == 0) fp->ptr++;
      else       return EOF;
   fp->cnt++;
   return(* --fp->ptr = c);
}

/* Изменить размер буфера */
void setbuffer(FILE *fp, uchar *buf, int size){
   fflush(fp);
   if(fp->base && (fp->flag & IOALLOC)) free(fp->base);
   fp->flag &= ~(IOALLOC|IONBF);
   if((fp->base = fp->ptr = buf) == NULL){
       fp->flag |= IONBF;  fp->bufsiz = 0;
   } else                  fp->bufsiz = size;
   fp->cnt = 0;
}

/* "Перемотать" файл в начало */
void rewind(FILE *fp){
   fflush(fp);
   lseek(fileno(fp), 0L, 0);
   fp->cnt = 0; fp->ptr = fp->base;
   clearerr(fp);
   if(fp->flag & IORW) fp->flag &= ~(IOREAD|IOWRT);
}
```

```
/* Позиционирование указателя чтения/записи */
#ifdef COMMENT
        base  ptr                       случай IOREAD
        |     |<----cnt---->|
 0L     |б  у |ф  е  р      |
|=======######@@@@@@@@@@@@@@@======== файл file
|             |<-p->|<-dl-->|
|<----pos---->|     |       |
|<----offset(new)-->|       |
|<----RWptr---------------->|

где      pos = RWptr - cnt;  // указатель с поправкой
offset = pos + p = RWptr - cnt + p = lseek(file,0L,1) - cnt + p
отсюда:  (для SEEK_SET)
         p = offset+cnt-lseek(file,0L,1);
или      (для SEEK_CUR)  dl = RWptr - offset = p - cnt
         lseek(file, dl, 1);
Условие, что указатель можно сдвинуть просто в буфере:
if( cnt > 0 && p <= cnt && base <= ptr + p ){
        ptr += p; cnt -= p;                }
#endif /*COMMENT*/

int fseek(FILE *fp, long offset, int whence){
   register resync, c; long p = (-1);
   clearerr(fp);
   if( fp->flag & (IOWRT|IORW)){
       fflush(fp);
       if(fp->flag & IORW){
          fp->cnt = 0; fp->ptr = fp->base; fp->flag &= ~IOWRT;
       }
       p = lseek(fileno(fp), offset, whence);
   } else if( fp->flag & IOREAD ){
       if(whence < 2 && fp->base && !(fp->flag & IONBF)){
          c = fp->cnt; p = offset;
          if(whence == 0) /* SEEK_SET */
             p +=   c - lseek(fileno(fp), 0L, 1);
          else offset -= c;
          if(!(fp->flag & IORW) &&
             c > 0 && p <= c && p >= fp->base - fp->ptr
          ){ fp->ptr += (int) p; fp->cnt -= (int) p;
             return 0;  /* done */
          }
          resync = offset & 01;
       } else resync = 0;
       if(fp->flag & IORW){
          fp->ptr = fp->base; fp->flag &= ~IOREAD; resync = 0;
       }
       p = lseek(fileno(fp), offset-resync, whence);
       fp->cnt = 0;       /* вынудить filbuf(); */
       if(resync) getc(fp);
   }
   return (p== -1 ? -1 : 0);
}
```

```
/* Узнать текущую позицию указателя */
long ftell(FILE *fp){
   long tres; register adjust;
   if(fp->cnt < 0) fp->cnt = 0;
        if(fp->flag &  IOREAD)       adjust = -(fp->cnt);
   else if(fp->flag & (IOWRT|IORW)){ adjust = 0;
        if(fp->flag & IOWRT &&
           fp->base && !(fp->flag & IONBF)) /* буферизован */
                adjust = fp->ptr - fp->base;
   } else return (-1L);
   if((tres = lseek(fileno(fp), 0L, 1)) < 0) return tres;
   return (tres + adjust);
}
```

5. Структуры данных.

Структуры ("записи") представляют собой агрегаты разнородных данных (полей *разного* типа); в отличие от массивов, где все элементы имеют *один и тот же* тип.

```
struct {
        int x, y;    /* два целых поля */
        char s[10]; /* и одно - для строки */
} s1;
```

Структурный тип может иметь имя:

```
struct XYS {
        int x, y;      /* два целых поля */
        char str[10]; /* и одно - для строки  */
};
```

Здесь мы объявили тип, но не отвели ни одной переменной этого типа (хотя могли бы). Теперь опишем переменную этого типа и указатель на нее:

```
struct XYS s2, *sptr = &s2;
```

Доступ к полям структуры производится по имени поля (а не по индексу, как у массивов):

```
имя_структурной_переменной.имя_поля
указатель_на_структуру  -> имя_поля
```

то есть

```
      не                          a
#define ВЕС  0             struct { int вес, рост; } x;
#define РОСТ 1             x.рост = 175;
int x[2]; x[РОСТ] = 175;
```

Например

```
s1.x = 13;
strcpy(s2.str, "Finish");
sptr->y = 27;
```

Структура может содержать структуры *другого* типа в качестве полей:

```
struct XYS_Z {
        struct XYS xys;
        int z;
} a1;
a1.xys.x = 71; a1.z = 12;
```

Структура *того же самого* типа не может содержаться в качестве поля - рекурсивные определения запрещены. Зато нередко используются поля - *ссылки* на структуры такого же типа (или другого). Это позволяет организовывать *списки* структур:

```
struct node {
        int value;
        struct node *next;
};
```

Очень часто используются массивы структур:

```
struct XYS array[20]; int i = 5, j;
array[i].x = 12;
j = array[i].x;
```

Статические структуры можно описывать с инициализацией, перечисляя значения их полей в {} через запятую:

```
extern struct node n2;
struct node n1 = { 1, &n2  },
            n2 = { 2, &n1  },
            n3 = { 3, NULL };
```

В этом примере *n2* описано предварительно для того, чтобы *&n2* в строке инициализации *n1* было определено.

Структуры одинакового типа можно присваивать целиком (что соответствует присваиванию каждого из полей):

```
struct XYS s1, s2;  ...
s2 = s1;
```

в отличие от массивов, которые присваивать целиком нельзя:

```
int a[5], b[5];  a = b; /* ОШИБОЧНО ! */
```

Пример обращения к полям структуры:

```
typedef struct _Point {
    short x, y; /* координаты точки */
    char *s;    /* метка точки      */
} Point;
Point p; Point *pptr; short *iptr;
struct _Curve {
  Point points[25]; /* вершины ломанной */
  int color;        /* цвет линии       */
} aLine[10], *linePtr = & aLine[0];
        ...
pptr = &p; /* указатель на структуру p */
p.x = 1; p.y = 2; p.s = "Grue";
linePtr->points[2].x  = 54; aLine[5].points[0].y  = 17;
```

В ы р а ж е н и е				значение
p.x	pptr->x	(*pptr).x	(&p)->x	1
			&p->x	ошибка
iptr= &p.x	iptr= &pptr->x	iptr= &(pptr->x)		адрес поля
*pptr->s	*(pptr->s)	*p.s	p.s[0]	'G'
pptr->s[1]	(&p)->s[1]	p.s[1]		'r'
&p->s[1]				ошибка
(*pptr).s	pptr->s	p.s		"Grue"
*pptr.s				ошибка

```
Вообще (&p)->field   =  p.field
       pptr->field   =  (*pptr).field
```

Объединения - это агрегаты данных, которые могут хранить в себе значения данных разных типов *на одном и том же месте*.

```
struct a{ int x, y; char *s; } A;
union  b{ int i; char *s; struct a aa; } B;
```

Структура:

```
   _________________________
A: | A.x         int       |   Три поля
   ------------------------   расположены подряд.
   | A.y         int       |   Получается как бы
   ------------------------   "карточка" с графами.
   | A.s         char *    |
   ------------------------
```

```
А у объединений поля расположены "параллельно",
на одном месте в памяти.
   ____________________________________________________
B: | B.i  int | B.s  char *  | B.aa     : B.aa.x  int    |
   -----------|              | struct a : B.aa.y  int    |
              ---------------|          : B.aa.s  char * |
                             |___________________________|
```

Это как бы "ящик" в который можно поместить значение любого типа из перечисленных, но не ВСЕ ВМЕСТЕ ("и то и это", как у структур), а ПО ОЧЕРЕДИ ("или/или"). Размер его достаточно велик, чтоб вместить

самый большой из перечисленных типов данных.

Мы можем занести в **union** значение и интерпретировать его как *другой* тип данных - это иногда используется в машинно-зависимых программах. Вот пример, выясняющий порядок байтов в **short** числах:

```
union lb {
       char s[2]; short i;
} x;
unsigned hi, lo;
x.i = (02 << 8) | 01;
hi = x.s[1]; lo = x.s[0];
printf( "%d %d\n", hi, lo);
```

или так:

```
#include <stdio.h>
union {
        int i;
        unsigned char s[sizeof(int)];
} u;
void main(){
        unsigned char *p;
        int n;

        u.i = 0x12345678;
        for(n=0, p=u.s; n < sizeof(int); n++, p++){
                printf("%02X ", *p);
        }
        putchar('\n');
}
```

или порядок слов в **long** числах:

```
union xx {
  long l;
  struct ab {
    short a;  /* low word  */
    short b;  /* high word */
  } ab;
} c;
main(){ /* На IBM PC 80386 печатает 00020001 */
  c.ab.a = 1; c.ab.b = 2; printf("%08lx\n", c.l );
}
```

5.1. Найдите ошибки в описании структурного шаблона:

```
structure { int arr[12],
            char string,
            int *sum
          }
```

5.2. Разработайте структурный шаблон, который содержал бы название месяца, трехбуквенную аббревиатуру месяца, количество дней в месяце и номер месяца. Инициализируйте его для невисокосного года.

```
struct month {
       char name[10];  /* или char *name; */
       char abbrev[4]; /* или char *abbrev; */
       int  days;
       int  num;
};

struct month months[12] = {            /* индекс */
       {"Январь" , "Янв", 31, 1 },     /*  0     */
       {"Февраль", "Фев", 28, 2 },     /*  1     */
              ...
       {"Декабрь", "Дек", 31, 12},     /* 11     */
}, *mptr = & months[0];  /* или *mptr = months */
```

```
main(){
      struct month *mptr;
      printf( "%s\n", mptr[1].name );
      printf( "%s %d\n", mptr->name, mptr->num );
}
```

Напишите функцию, сохраняющую массив *months* в файл; функцию, считывающую его из файла. Используйте **fprintf** и **fscanf**.

В чем будет разница в функции чтения, когда поле *name* описано как char *name*[10] и как char **name*?

Ответ: во втором случае для сохранения прочитанной строки надо заказывать память динамически при помощи **malloc**() и сохранять в ней строку при помощи **strcpy**(), т.к. память для хранения самой строки в структуре не зарезервирована (а только для указателя на нее).

Найдите ошибку в операторах функции **main**(). Почему печатается не "Февраль", а какой-то мусор? Указание: куда указывает указатель *mptr*, описанный в **main**()? Ответ: в "неизвестно куда" - это локальная переменная (причем не получившая начального значения - в ней содержится мусор), а не то же самое, что указатель *mptr*, описанный выше! Уберите описание *mptr* из **main**.

Заметим, что для распечатки всех или нескольких полей структуры следует ЯВНО перечислить в **printf**() все нужные поля и указать форматы, соответствующие типам этих полей. Не существует формата или стандартной функции, позволяющей распечатать все поля сразу (однако такая функция может быть написана вами для конкретного типа структур). Также не существует формата для **scanf**(), который вводил бы структуру целиком. Вводить можно только по частям - каждое поле отдельно.

5.3. Напишите программу, которая по номеру месяца возвращает общее число дней года вплоть до этого месяца.

5.4. Переделайте предыдущую программу таким образом, чтобы она по написанному буквами названию месяца возвращала общее число дней года вплоть до этого месяца. В программе используйте функцию **strcmp**().

5.5. Переделайте предыдущую программу таким образом, чтобы она запрашивала у пользователя день, месяц, год и выдавала общее количество дней в году вплоть до данного дня. Месяц может обозначаться номером, названием месяца или его аббревиатурой.

5.6. Составьте структуру для учетной картотеки служащего, которая содержала бы следующие сведения: фамилию, имя, отчество; год рождения; домашний адрес; место работы, должность; зарплату; дату поступления на работу.

5.7. Что печатает программа?

```
struct man {
        char name[20];
        int salary;
} workers[] = {
        { "Иванов",  200 },
        { "Петров",  180 },
        { "Сидоров", 150 }
}, *wptr, chief = { "начальник", 550 };

main(){
   struct man *ptr, *cptr, save;

   ptr = wptr = workers + 1;
   cptr = &chief;
   save = workers[2]; workers[2] = *wptr; *wptr = save;
   wptr++; ptr--; ptr->salary = save.salary;

   printf( "%c %s %s %s %s\n%d %d %d %d\n%d %d %c\n",
     *workers[1].name, workers[2].name, cptr->name,
       ptr[1].name, save.name,
         wptr->salary, chief.salary,
           (*ptr).salary, workers->salary,
     wptr - ptr, wptr - workers, *ptr->name );
}
```

Ответ:

```
С Петров начальник Сидоров Сидоров
180 550 150 150
2 2 И
```

5.8. Разберите следующий пример:

```
#include <stdio.h>
struct man{
        char *name, town[4]; int salary;
        int addr[2];
} men[] = {
        { "Вася",  "Msc",     100,  { 12,  7 } },
        { "Гриша", "Len",     120,  { 6,  51 } },
        { "Петя",  "Rig",     140,  { 23, 84 } },
        { NULL,    ""   ,      -1,  { -1, -1 } }
};
main(){
   struct man *ptr, **ptrptr;
   int i;

   ptrptr = &ptr;
   *ptrptr = &men[1];     /* men+1 */

   printf( "%s    %d    %s   %d   %c\n",
            ptr->name,
                  ptr->salary,
                        ptr->town,
                             ptr->addr[1],
                                  ptr[1].town[2] );

   (*ptrptr)++;

   /* копируем *ptr в men[0] */
   men[0].name    = ptr->name;       /* (char *) #1 */
   strcpy( men[0].town, ptr->town ); /* char []  #2 */
   men[0].salary  =     ptr->salary; /* int      #3 */
   for( i=0; i < 2; i++ )
       men[0].addr[i] = ptr->addr[i];  /* массив #4 */

   /* распечатываем массив структур */
   for(ptr=men; ptr->name; ptr++ )
       printf( "%s %s %d\n",
                ptr->name, ptr->town, ptr->addr[0]);
}
```

Обратите внимание на такие моменты:

1) Как производится работа с указателем на указатель (*ptrptr*).

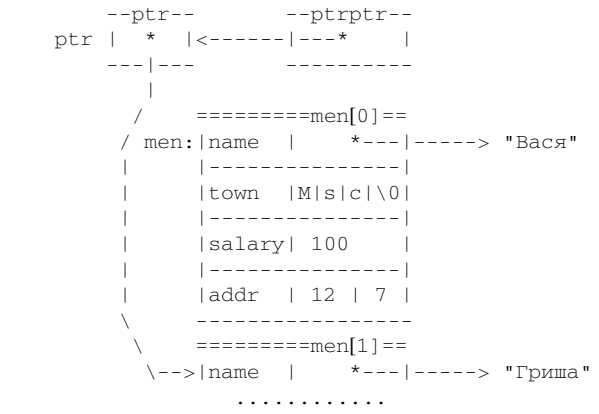
2) При копировании структур отдельными полями, поля скалярных типов (int, char, long, ..., указатели) копируются операцией присваивания (см. строки с пометками #1 и #3). Поля векторных типов (массивы) копируются при помощи цикла, поэлементно пересылающего массив (строка #4). Строки (массивы букв) пересылаются стандартной функцией **strcpy** (строка #2). Все это относится не только к полям структур, но и к переменным таких типов. Структуры можно также копировать не по полям, а целиком: *men*[0]= **ptr*;

3) Запись аргументов функции **printf**() лесенкой позволяет лучше видеть, какому формату соответствует каждый аргумент.

4) При распечатке массива структур мы печатаем не определенное их количество (равное размеру массива), а пользуемся указателем **NULL** в поле *name* последней структуры как признаком конца массива.

5) В поле *town* мы храним строки из 3х букв, однако выделяем для хранения массив из 4х байт. Это необходимо потому, что строка "Msc" состоит не из 3х, а из 4х байтов: 'M','s','c','\0'.

При работе со структурами и указателями большую помощь могут оказать рисунки. Вот как (например) можно нарисовать данные из этого примера (массив *men* изображен не весь):



5.9. Составьте программу "справочник по таблице Менделеева", которая по названию химического элемента выдавала бы его характеристики. Таблицу инициализируйте массивом структур.

5.10. При записи данных в файл (да и вообще) используйте структуры вместо массивов, если элементы массива имеют разное смысловое назначение. Не воспринимайте структуру просто как средство объединения данных разных типов, она может быть и средством объединения данных одного типа, если это добавляет осмысленности нашей программе. Чем плох фрагмент?

```
int data[2];

data[0] = my_key;
data[1] = my_value;

write(fd, (char *) data, 2 * sizeof(int));
```

Во-первых, тогда уж лучше указать размер всего массива сразу (хотя бы на тот случай, если мы изменим его размер на 3 и забудем поправить множитель с 2 на 3).

```
write(fd, (char *) data, sizeof data);
```

Кстати, почему мы пишем *data*, а не **&***data*? (ответ: потому что имя массива и есть его адрес). Во-вторых, элементы массива имеют разный смысл, так не использовать ли тут структуру?

```
struct _data {
        int key;
        int value;
} data;

data.key   = my_key;
data.value = my_value;

write(fd, &data, sizeof data);
```

5.11. Что напечатает следующая программа? Нарисуйте расположение указателей по окончании данной программы.

```
#include <stdio.h>
struct lnk{
   char c;
   struct lnk *prev, *next;
}  chain[20], *head = chain;

add(c) char c;
{
   head->c = c;
   head->next = head+1;
   head->next->prev = head;
   head++;
}
main(){
   char *s = "012345";
```

```
   while( *s ) add( *s++ );
   head->c = '-';
   head->next = (struct lnk *)NULL;
   chain->prev = chain->next;
   while( head->prev ){
        putchar( head->prev->c );
        head = head->prev;
        if( head->next )
            head->next->prev = head->next->next;
   }
}
```

5.12. Напишите программу, составляющую двунаправленный список букв, вводимых с клавиатуры. Конец ввода - буква '\n'. После третьей буквы вставьте букву '+'. Удалите пятую букву. Распечатайте список в обратном порядке. Оформите операции вставки/удаления как функции. Элемент списка должен иметь вид:

```
struct elem{
       char  letter;       /* буква         */
       char  *word;        /* слово         */
       struct elem *prev;  /* ссылка назад  */
       struct elem *next;  /* ссылка вперед */
};
struct elem *head, /* первый элемент списка */
            *tail, /* последний элемент     */
            *ptr,  /* рабочая переменная    */
            *prev; /* предыдущий элемент при просмотре */
int c, cmp;
         ...
while((c = getchar()) != '\n' )
      Insert(c, tail);
for(ptr=head; ptr != NULL; ptr=ptr->next)
      printf("буква %c\n", ptr->letter);
```

Память лучше отводить не из массива, а функцией **calloc()**, которая аналогична функции **malloc**(), но дополнительно расписывает выделенную память байтом '\0' (0, NULL). Вот функции вставки и удаления:

```
extern char *calloc();
/* создать новое звено списка для буквы c */
struct elem *NewElem(c) char c; {
   struct elem *p = (struct elem *)
                    calloc(1, sizeof(struct elem));
   /* calloc автоматически обнуляет все поля,
    * в том числе prev и next
    */
   p->letter = c; return p;
}

/* вставка после ptr (обычно - после tail) */
Insert(c, ptr) char c; struct elem *ptr;
{  struct elem *newelem = NewElem(c), *right;
   if(head == NULL){  /* список был пуст */
      head=tail=newelem; return;  }
   right = ptr->next; ptr->next = newelem;
   newelem->prev = ptr; newelem->next = right;
   if( right ) right->prev = newelem;
   else        tail        = newelem;
}

/* удалить ptr из списка */
Delete( ptr ) struct elem *ptr; {
   struct elem *left=ptr->prev, *right=ptr->next;
   if( right ) right->prev = left;
   if( left  ) left->next  = right;
   if( tail == ptr ) tail  = left;
   if( head == ptr ) head  = right;
   free((char *) ptr);
}
```

Напишите аналогичную программу для списка *слов*.

```
struct elem *NewElem(char *s) {
   struct elem *p = (struct elem *)
     calloc(1, sizeof(struct elem));
   p->word = strdup(s);
   return p;
}
void DeleteElem(struct elem *ptr){
        free(ptr->word);
        free(ptr);
}
```

Усложнение: вставляйте слова в список в *алфавитном порядке*. Используйте для этого функцию **strcmp**(), просматривайте список так:

```
struct elem *newelem;

if (head == NULL){  /* список пуст */
    head = tail = NewElem(новое_слово);
    return;
}
/* поиск места в списке */
for(cmp= -1, ptr=head, prev=NULL;
    ptr;
    prev=ptr, ptr=ptr->next
)
if((cmp = strcmp(новое_слово, ptr->word)) <= 0 )
          break;
```

Если цикл окончился с *cmp*==0, то такое слово уже есть в списке. Если *cmp* < 0, то такого слова не было и *ptr* указывает элемент, *перед* которым надо вставить слово *новое_слово*, а *prev* - *после* которого (*prev*==NULL означает, что надо вставить в начало списка); т.е. слово вставляется между *prev* и *ptr*. Если *cmp* > 0, то слово надо добавить в конец списка (при этом *ptr*==NULL).

```
head ==> "a" ==> "b" ==> "d" ==> NULL
          |               |
         prev    "c"     ptr

if(cmp == 0) return; /* слово уже есть */
newelem = NewElem( новое_слово );
if(prev == NULL){       /* в начало */
   newelem->next = head;
   newelem->prev = NULL;
   head->prev    = newelem;
   head          = newelem;
} else if(ptr == NULL){ /* в конец */
   newelem->next = NULL;
   newelem->prev = tail;
   tail->next    = newelem;
   tail          = newelem;
} else {                /* между prev и ptr */
   newelem->next = ptr;
   newelem->prev = prev;
   prev->next    = newelem;
   ptr ->prev    = newelem;
}
```

5.13. Напишите функции для работы с комплексными числами

```
struct complex {
       double re, im;
};
```

Например, сложение выглядит так:

```
struct complex add( c1, c2 )
       struct complex c1, c2;
{
       struct complex sum;
```

```
        sum.re = c1.re + c2.re;
        sum.im = c1.im + c2.im;
        return sum;
}

struct complex a = { 12.0, 14.0 },
               b = { 13.0, 2.0  };
main(){
        struct complex c;
        c = add( a, b );
        printf( "(%g,%g)\n", c.re, c.im );
}
```

5.14. Массивы в Си нельзя присваивать целиком, зато структуры - можно. Иногда используют такой трюк: структуру из единственного поля-массива

```
typedef struct {
        int ai[5];
} intarray5;
intarray5 a, b = { 1, 2, 3, 4, 5 };
```

и теперь законно

```
a = b;
```

Зато доступ к ячейкам массива выглядит теперь менее изящно:

```
a.ai[2] = 14;
for(i=0; i < 5; i++) printf( "%d\n", a.ai[i] );
```

Также невозможно передать *копию* массива в качестве фактического параметра функции. Даже если мы напишем:

```
typedef int ARR16[16];
ARR16 d;
void f(ARR16 a){
  printf( "%d %d\n", a[3], a[15]);
  a[3] = 2345;
}
void main(void){
  d[3] = 9; d[15] = 98;
  f(d);
  printf("Now it is %d\n", d[3]);
}
```

то последний **printf** напечатает "Now it is 2345", поскольку в **f** передается *адрес* массива, но не его копия; поэтому оператор *a*[3]=2345 изменяет *исходный* массив. Обойти это можно, использовав тот же трюк, поскольку при передаче структуры в качестве параметра передается уже не ее адрес, а копия всей структуры (как это и принято в Си во всех случаях, кроме массивов).

5.15. Напоследок упомянем про *битовые поля* - элементы структуры, занимающие только *часть* машинного слова - только несколько битов в нем. Размер поля в битах задается конструкцией **:***число_битов*. Битовые поля используются для более компактного хранения информации в структурах (для экономии места).

```
struct XYZ {
   /* битовые поля должны быть unsigned */
   unsigned x:2;   /* 0 .. 2**2 - 1 */
   unsigned y:5;   /* 0 .. 2**5 - 1 */
   unsigned z:1;   /* YES=1 NO=0    */
} xyz;
main(){
  printf("%u\n", sizeof(xyz)); /* == sizeof(int) */
  xyz.z = 1; xyz.y = 21; xyz.x = 3;
  printf("%u %u %u\n", xyz.x, ++xyz.y, xyz.z);

  /* Значение битового поля берется по модулю
   * максимально допустимого числа 2**число_битов - 1
   */
xyz.y = 32 /* максимум */ + 7; xyz.x = 16+2; xyz.z = 11;
printf("%u %u %u\n", xyz.x, xyz.y, xyz.z); /* 2 7 1 */
```

```
}
```

Поле ширины 1 часто используется в качестве битового флага: вместо

```
#define FLAG1   01
#define FLAG2   02
#define FLAG3   04
int x;  /* слово для нескольких флагов */
x |= FLAG1; x &= ~FLAG2; if(x & FLAG3) ...;
```

используется

```
struct flags {
       unsigned flag1:1, flag2:1, flag3:1;
} x;
x.flag1 = 1; x.flag2 = 0; if( x.flag3 ) ...;
```

Следует однако учесть, что машинный код для работы с битовыми полями более сложен и занимает больше команд (т.е. медленнее и длиннее).

К битовым полям нельзя применить операцию взятия адреса "&", у них нет адресов и смещений!

5.16. Пример на использование структур с полем переменного размера. Часть переменной длины может быть лишь одна и обязана быть последним полем структуры. Внимание: это программистский трюк, использовать осторожно!

```
#include <stdio.h>
#define SZ 5
extern char *malloc();
#define VARTYPE char

struct obj {
        struct header {   /* постоянная часть */
                int cls;
                int size; /* размер переменной части */
        } hdr;

        VARTYPE body [1];    /* часть переменного размера:
                               в описании ровно ОДИН элемент массива */
} *items [SZ];            /* указатели на структуры */

#define OFFSET(field, ptr)        ((char *) &ptr->field - (char *)ptr)
int body_offset;
```

```
/* создание новой структуры */
struct obj *newObj( int cl, char *s )
{
    char *ptr; struct obj *op;
    int n = strlen(s);  /* длина переменной части (штук VARTYPE) */
    int newsize = sizeof(struct header) + n * sizeof(VARTYPE);

    printf("[n=%d newsize=%d]\n", n, newsize);

    /* newsize = (sizeof(struct obj) - sizeof(op->body)) + n * sizeof(op->body);

       При использовании этого размера не учитывается, что struct(obj)
       выровнена на границу sizeof(int).
       Но в частности следует учитывать и то, на границу чего выровнено
       начало поля op->body. То есть самым правильным будет

       newsize = body_offset + n * sizeof(op->body);

    */

    /* отвести массив байт без внутренней структуры */
    ptr = (char *) malloc(newsize);

    /* наложить поверх него структуру */
    op = (struct obj *) ptr;

    op->hdr.cls  = cl;
    op->hdr.size = n;

    strncpy(op->body, s, n);

    return op;
}

void printobj( struct obj *p )
{
    register i;

    printf( "OBJECT(cls=%d,size=%d)\n", p->hdr.cls, p->hdr.size);
    for(i=0; i < p->hdr.size; i++ )
        putchar( p->body[i] );
    putchar( '\n' );
}

char *strs[] = { "a tree", "a maple", "an oak", "the birch", "the fir" };

int main(int ac, char *av[]){
   int i;

   printf("sizeof(struct header)=%d sizeof(struct obj)=%d\n",
           sizeof(struct header),   sizeof(struct obj));

   {
           struct obj *sample;
           printf("offset(cls)=%d\n",               OFFSET(hdr.cls,  sample));
           printf("offset(size)=%d\n",              OFFSET(hdr.size, sample));
           printf("offset(body)=%d\n", body_offset = OFFSET(body,     sample));
   }
```

```
   for( i=0; i < SZ; i++ )
      items[i] = newObj( i, strs[i] );

   for( i=0; i < SZ; i++ ){
      printobj( items[i] ); free( items[i] ); items[i] = NULL;
   }
   return 0;
}
```

5.17. Напишите программу, реализующую список со "старением". Элемент списка, к которому обращались последним, находится в голове списка. Самый старый элемент вытесняется к хвосту списка и в конечном счете из списка удаляется. Такой алгоритм использует ядро **UNIX** для кэширования блоков файла в оперативной памяти: блоки, к которым часто бывают обращения оседают в памяти (а не на диске).

```
/* Список строк, упорядоченных по времени их добавления в список,
 * т.е. самая "свежая" строка - в начале, самая "древняя" - в конце.
 * Строки при поступлении могут и повторяться! По подобному принципу
 * можно организовать буферизацию блоков при обмене с диском.
 */

#include <stdio.h>
extern char *malloc(), *gets();
#define MAX 3   /* максимальная длина списка */
int nelems = 0; /* текущая длина списка      */

struct elem {           /* СТРУКТУРА ЭЛЕМЕНТА СПИСКА            */
    char *key;          /* Для блоков - это целое - номер блока */
    struct elem *next;  /* следующий элемент списка             */
    /* ... и может что-то еще ...                               */
} *head;                /* голова списка                        */

void printList(), addList(char *), forget();

void main(){ /* Введите a b c d b a c */
    char buf[128];
    while(gets(buf)) addList(buf), printList();
}

/* Распечатка списка */
void printList(){    register struct elem *ptr;
    printf( "В списке %d элементов\n", nelems );
    for(ptr = head; ptr != NULL; ptr = ptr->next )
        printf( "\t\"%s\"\n", ptr->key );
}

/* Добавление в начало списка */
void addList(char *s)
{   register struct elem *p, *new;
    /* Анализ - нет ли уже в списке */
    for(p = head; p != NULL; p = p->next )
        if( !strcmp(s, p->key)){ /* Есть. Перенести в начало списка */
            if( head == p ) return; /* Уже в начале */
            /* Удаляем из середины списка */
            new = p;    /* Удаляемый элемент */
            for(p = head; p->next != new; p = p->next );
            /* p указывает на предшественника new */
            p->next = new->next; goto Insert;
        }
    /* Нет в списке */
    if( nelems >= MAX ) forget(); /* Забыть старейший */
    if((new = (struct elem *) malloc(sizeof(struct elem)))==NULL) goto bad;
    if((new->key = malloc(strlen(s) + 1)) == NULL) goto bad;
    strcpy(new->key, s); nelems++;
Insert:         new->next = head;   head = new;  return;
bad:            printf( "Нет памяти\n" ); exit(13);
}
```

```
/* Забыть хвост списка */
void forget(){       struct elem *prev = head, *tail;
    if( head == NULL ) return;  /* Список пуст */
    /* Единственный элемент ? */
    if((tail = head->next) == NULL){ tail=head; head=NULL; goto Del; }
    for( ; tail->next != NULL; prev = tail, tail = tail->next );
    prev->next = NULL;
Del:    free(tail->key);  free(tail);   nelems--;
}
```

6. Системные вызовы и взаимодействие с UNIX.

В этой главе речь пойдет о процессах. Скомпилированная программа хранится на диске как обычный нетекстовый файл. Когда она будет загружена в память компьютера и начнет выполняться - она станет *процессом*.

UNIX - многозадачная система (мультипрограммная). Это означает, что одновременно может быть запущено много процессов. Процессор выполняет их в режиме *разделения времени* - выделяя по очереди квант времени одному процессу, затем другому, третьему... В результате создается впечатление *параллельного* выполнения всех процессов (на многопроцессорных машинах параллельность истинная). Процессам, ожидающим некоторого события, время процессора не выделяется. Более того, "спящий" процесс может быть временно откачан (т.е. скопирован из памяти машины) на диск, чтобы освободить память для других процессов. Когда "спящий" процесс дождется события, он будет "разбужен" системой, переведен в ранг "готовых к выполнению" и, если был откачан - будет возвращен с диска в память (но, может быть, на другое место в памяти!). Эта процедура носит название "своппинг" (*swapping*).

Можно запустить несколько процессов, выполняющих программу из *одного и того же* файла; при этом все они будут (если только специально не было предусмотрено иначе) независимыми друг от друга. Так, у каждого пользователя, работающего в системе, имеется свой собственный процесс-*интерпретатор команд* (своя копия), выполняющий программу из файла **/bin/csh** (или **/bin/sh**).

Процесс представляет собой изолированный "мир", общающийся с другими "мирами" во Вселенной при помощи:

a) Аргументов функции **main**:
void **main**(int *argc*, char **argv*[], char **envp*[]);
Если мы наберем команду
$ **a.out** *a1 a2 a3*
то функция **main** программы из файла *a.out* вызовется с

```
argc    = 4  /* количество аргументов */
argv[0] = "a.out"       argv[1] = "a1"
argv[2] = "a2"          argv[3] = "a3"
argv[4] = NULL
```

По соглашению *argv*[0] содержит имя выполняемого файла из которого загружена эта программа†.

b) Так называемого "окружения" (или "среды") char **envp*[], продублированного также в предопределенной переменной
extern char ****environ**;
Окружение состоит из строк вида
"*ИМЯПЕРЕМЕННОЙ=значение*"
Массив этих строк завершается **NULL** (как и *argv*). Для получения значения переменной с именем *ИМЯ* существует стандартная функция
char ***getenv**(char **ИМЯ*);
Она выдает либо *значение*, либо **NULL** если переменной с таким именем нет.

c) Открытых файлов. По умолчанию (неявно) всегда открыты 3 канала:

```
                    ВВОД          В Ы В О Д
FILE *              stdin      stdout    stderr
соответствует fd      0          1         2
связан с         клавиатурой     дисплеем
```

Эти каналы достаются процессу "в наследство" от запускающего процесса и связаны с дисплеем и клавиатурой, если только не были перенаправлены. Кроме того, программа может сама явно открывать файлы (при помощи **open**, **creat**, **pipe**, **fopen**). Всего программа может одновременно открыть до 20 файлов (считая стандартные каналы), а в некоторых системах и больше (например, 64). В **MS DOS** есть еще 2 предопределенных канала вывода: *stdaux* - в последовательный коммуникационный порт, *stdprn* - на принтер.

d) Процесс имеет уникальный номер, который он может узнать вызовом
int *pid* = **getpid**();

---

† Именно это имя показывает команда **ps -ef**

```
#include <stdio.h>
main(ac, av) char **av; {
  execl("/bin/sleep", "Take it easy", "1000", NULL);
}
```

а также узнать номер "родителя" вызовом
int *ppid* = **getppid**();
Процессы могут по этому номеру посылать друг другу сигналы:
kill(*pid* /* кому */, *sig* /* номер сигнала */);
и реагировать на них
signal (*sig* /*по сигналу*/, *f* /*вызывать f(sig)*/);

e) Существуют и другие средства коммуникации процессов: семафоры, сообщения, общая память, сетевые коммуникации.

f) Существуют некоторые другие параметры (контекст) процесса: например, его текущий каталог, который достается в наследство от процесса-"родителя", и может быть затем изменен системным вызовом
chdir(char **имя_нового_каталога*);
У каждого процесса есть свой *собственный* текущий рабочий каталог (в отличие от **MS DOS**, где текущий каталог одинаков для всех задач). К "прочим" характеристикам отнесем также: управляющий терминал; группу процессов (*pgrp*); идентификатор (номер) владельца процесса (*uid*), идентификатор группы владельца (*gid*), реакции и маски, заданные на различные сигналы; и.т.п.

g) Издания других запросов (системных вызовов) к операционной системе ("богу") для выполнения различных "внешних" операций.

h) Все остальные действия происходят внутри процесса и никак не влияют на другие процессы и устройства ("миры"). В частности, один процесс НИКАК не может получить доступ к памяти другого процесса, если тот не позволил ему это явно (механизм *shared memory*); адресные пространства процессов независимы и изолированы (равно и пространство ядра изолировано от памяти процессов).

Операционная система выступает в качестве *коммуникационной среды*, связывающей "миры"-процессы, "миры"-внешние устройства (включая терминал пользователя); а также в качестве распорядителя ресурсов "Вселенной", в частности - времени (по очереди выделяемого активным процессам) и пространства (в памяти компьютера и на дисках).

Мы уже неоднократно упоминали "системные вызовы". Что же это такое? С точки зрения Си-программиста - это обычные функции. В них передают аргументы, они возвращают значения. Внешне они ничем не отличаются от написанных нами или библиотечных функций и вызываются из программ одинаковым с ними способом.

С точки же зрения реализации - есть глубокое различие. Тело функции-сисвызова расположено не в нашей программе, а в резидентной (т.е. постоянно находящейся в памяти компьютера) управляющей программе, называемой *ядром операционной системы*†. Сам термин "системный вызов" как раз означает "вызов системы для выполнения действия", т.е. вызов функции в ядре системы. Ядро работает в *привелегированном режиме*, в котором имеет доступ к некоторым системным таблицам£, регистрам и портам

---

† Собственно, операционная система характеризуется набором предоставляемых ею системных вызовов, поскольку все концепции, заложенные в системе, доступны нам *только* через них. Если мы имеем две реализации системы с *разным* внутренним устройством ядер, но предоставляющие *одинаковый интерфейс* системных вызовов (их набор, смысл и поведение), то это все-таки **одна и та же** система! Ядра могут не просто отличаться, но и быть построенными на совершенно различных принципах: так обстоит дело с **UNIX**-ами на однопроцессорных и многопроцессорных машинах. Но для нас ядро - это "черный ящик", полностью определяемый его *поведением*, т.е. своим *интерфейсом* с программами, но не внутренним устройством. Вторым параметром, характеризующим ОС, являются *форматы данных*, используемые системой: форматы данных для сисвызовов и формат информации в различных файлах, в том числе формат оформления выполняемых файлов (формат данных *в физической памяти* машины в этот список **не** входит - он зависим от реализации и от процессора). Как правило, программа пишется так, чтобы использовать соглашения, принятые в данной системе, для чего она просто включает ряд стандартных include-файлов с описанием этих форматов. *Имена* этих файлов также можно отнести к интерфейсу системы.

Поведение всех программ в системе вытекает из поведения системных вызовов, которыми они пользуются. Даже то, что **UNIX** является многозадачной системой, непосредственно вытекает из наличия системных вызовов **fork**, **exec**, **wait** и спецификации их функционирования!

То же можно сказать про язык Си - мобильность программы зависит в основном от набора используемых в ней *библиотечных функций* (и, в меньшей степени, от диалекта самого языка, который должен удовлетворять *стандарту* на язык Си). Если две разные системы предоставляют все эти функции (которые могут быть по-разному реализованы, но должны делать одно и то же), то программа будет компилироваться и работать в обоих системах, более того, работать в них *одинаково*.

£ Таким как таблица процессов, таблица открытых файлов (всех вместе и для каждого процесса), и.т.п.

внешних устройств и диспетчера памяти, к которым обычным программам доступ аппаратно запрещен (в отличие от **MS DOS**, где все таблицы ядра доступны пользовательским программам, что создает раздолье для вирусов). Системный вызов происходит в 2 этапа: сначала в пользовательской программе вызывается библиотечная функция-"корешок", тело которой написано на ассемблере и содержит команду генерации *программного прерывания*. Это - главное отличие от нормальных Си-функций - вызов по прерыванию. Вторым этапом является реакция ядра на прерывание:

1. переход в привелегированный режим;
2. разбирательство, КТО обратился к ядру, и подключение **u-area** этого процесса к адресному пространству ядра (*context switching*);
3. извлечение аргументов из памяти запросившего процесса;
4. выяснение, ЧТО же хотят от ядра (один из аргументов, невидимый нам - это *номер* системного вызова);
5. проверка корректности остальных аргументов;
6. проверка прав процесса на допустимость выполнения такого запроса;
7. вызов тела требуемого системного вызова - это обычная Си-функция в ядре;
8. возврат ответа в память процесса;
9. выключение привелегированного режима;
10. возврат из прерывания.

Во время системного вызова (шаг 7) процесс может "заснуть", дожидаясь некоторого события (например, нажатия кнопки на клавиатуре). В это время ядро передаст управление другому процессу. Когда наш процесс будет "разбужен" (событие произошло) - он продолжит выполнение шагов системного вызова.

Большинство системных вызовов возвращают в программу в качестве своего значения признак успеха: 0 - все сделано, (-1) - сисвызов завершился неудачей; либо некоторое содержательное значение при успехе (вроде дескриптора файла в **open**(), и (-1) при неудаче. В случае неудачного завершения в предопределенную переменную **errno** заносится номер ошибки, описывающий причину неудачи (коды ошибок предопределены, описаны в include-файле <*errno.h*> и имеют вид **Ечтото**). Заметим, что при УДАЧЕ эта переменная просто *не изменяется* и может содержать любой мусор, поэтому проверять ее имеет смысл лишь в случае, если ошибка действительно произошла:

```
#include <errno.h>      /* коды ошибок */
extern int errno;
extern char *sys_errlist[];
int value;
if((value = sys_call(...)) < 0 ){
   printf("Error:%s(%d)\n", sys_errlist[errno],
                            errno );
   exit(errno); /* принудительное завершение программы */
}
```

Предопределенный массив **sys_errlist**, хранящийся в стандартной библиотеке, содержит строки-расшифровку смысла ошибок (по-английски). Посмотрите описание функции **perror**().

6.1. Файлы и каталоги.

6.1.1. Используя системный вызов **stat**, напишите программу, определяющую тип файла: обычный файл, каталог, устройство, FIFO-файл. Ответ:

```
#include <sys/types.h>
#include <sys/stat.h>

typeOf( name ) char *name;
{  int type; struct stat st;
   if( stat( name, &st ) < 0 ){
           printf( "%s не существует\n", name );
           return 0;
   }
   printf("Файл имеет %d имен\n", st.st_nlink);

   switch(type = (st.st_mode & S_IFMT)){
   case S_IFREG:
```

```
            printf( "Обычный файл размером %ld байт\n",
                        st.st_size ); break;
      case S_IFDIR:
              printf( "Каталог\n" );      break;
      case S_IFCHR:   /* байтоориентированное  */
      case S_IFBLK:   /* блочноориентированное */
              printf( "Устройство\n" );   break;
      case S_IFIFO:
              printf( "FIFO-файл\n" );    break;
      default:
              printf( "Другой тип\n" );   break;
      }       return type;
 }
```

6.1.2. Напишите программу, печатающую: свои аргументы, переменные окружения, информацию о всех открытых ею файлах и используемых трубах. Для этой цели используйте системный вызов

```
struct stat st; int used, fd;
for(fd=0; fd < NOFILE; fd++ ){
  used = fstat(fd, &st) < 0 ? 0 : 1;
  ...
}
```

Программа может использовать дескрипторы файлов с номерами 0..**NOFILE**-1 (обычно 0..19). Если **fstat** для какого-то *fd* вернул код ошибки (<0), это означает, что данный дескриптор не связан с открытым файлом (т.е. не используется). **NOFILE** определено в include-файле <*sys/param.h*>, содержащем разнообразные параметры данной системы.

6.1.3. Напишите упрощенный аналог команды **ls**, распечатывающий содержимое текущего каталога (файла с именем ".") без сортировки имен по алфавиту. Предусмотрите чтение каталога, чье имя задается как аргумент программы. Имена "." и ".." не выдавать.

Формат каталога описан в header-файле <*sys/dir.h*> и в "канонической" версии выглядит так: каталог - это файл, состоящий из структур **direct**, каждая описывает одно имя файла, входящего в каталог:

```
struct  direct {
   unsigned short d_ino;   /* 2 байта: номер I-узла */
   char    d_name[DIRSIZ]; /* имя файла             */
};
```

В семействе **BSD** формат каталога несколько иной - там записи имеют разную длину, зависящую от длины имени файла, которое может иметь длину от 1 до 256 символов.

Имя файла может состоять из *любых* символов, кроме '\0', служащего признаком конца имени и '/', служащего разделителем. В имени допустимы пробелы, управляющие символы (но не рекомендуются!), *любое* число точек (в отличие от **MS DOS**, где допустима *единственная* точка, отделяющая собственно имя от суффикса (расширения)), разрешены даже непечатные (т.е. управляющие) символы! Если имя файла имеет длину 14 (**DIRSIZ**) символов, то оно *не оканчивается* байтом '\0'. В этом случае для печати имени файла возможны три подхода:

1. Выводить символы при помощи **putchar**()-а в цикле. Цикл прерывать по индексу равному **DIRSIZ**, либо по достижению байта '**\0**'.

2. Скопировать поле *d_name* в другое место:

   ```
   char buf[ DIRSIZ + 1 ];
   strncpy(buf, d.d_name, DIRSIZ);
   buf[ DIRSIZ ] = '\0';
   ```

   Этот способ лучший, если имя файла надо не просто напечатать, но и запомнить на будущее, чтобы использовать в своей программе.

3. Использовать такую особенность функции **printf**():

   ```
   #include <sys/types.h>
   #include <sys/dir.h>

   struct direct d;
      ...
   printf( "%*.*s\n", DIRSIZ, DIRSIZ, d.d_name );
   ```

Если файл был стерт, то в поле *d_ino* записи каталога будет содержаться 0 (именно поэтому I-узлы нумеруются начиная с 1, а не с 0). При удалении файла содержимое его (блоки) уничтожается, I-узел освобождается, но имя в каталоге не затирается физически, а просто помечается как стертое: *d_ino*=0; Каталог при этом никак не уплотняется и не укорачивается! Поэтому имена с *d_ino*==0 выдавать не следует - это имена уже уничтоженных файлов.

При создании нового имени (**creat**, **link**, **mknod**) система просматривает каталог и переиспользует первый от начала свободный слот (ячейку каталога) где *d_ino*==0, записывая новое имя в него (только в этот момент старое имя-призрак окончательно исчезнет физически). Если пустых мест нет - каталог удлиняется.

Любой каталог *всегда* содержит два стандартных имени: "**.**" - ссылка на этот же каталог (на его собственный I-node), "**..**" - на вышележащий каталог. У корневого каталога "/" оба этих имени ссылаются на него же самого (т.е. содержат *d_ino*==2).

Имя каталога не содержится в нем самом. Оно содержится в "родительском" каталоге **..**.

Каталог в **UNIX** - это обычный дисковый файл. Вы можете *читать* его из своих программ. Однако *никто* (включая суперпользователя£) не может *записывать* что-либо в каталог при помощи **write**. Изменения содержимого каталогов выполняет *только ядро*, отвечая на запросы в виде системных вызовов **creat**, **unlink**, **link**, **mkdir**, **rmdir**, **rename**, **mknod**. Коды доступа для каталога интерпретируются следующим образом:

w *запись*

S_IWRITE. Означает право создавать и уничтожать в каталоге имена файлов при помощи этих вызовов. То есть: право создавать, удалять и переименовывать файлы в каталоге. Отметим, что для переименования или удаления файла вам не требуется иметь доступ по записи к самому файлу - достаточно иметь доступ по записи к каталогу, содержащему его имя!

r *чтение*

S_IREAD. Право читать каталог как обычный файл (право выполнять **opendir**, см. ниже): благодаря этому мы можем получить список имен файлов, содержащихся в каталоге. Однако, если мы ЗАРАНЕЕ знаем имена файлов в каталоге, мы МОЖЕМ работать с ними - если имеем право доступа "*выполнение*" для этого каталога!

x *выполнение*

S_IEXEC. Разрешает *поиск* в каталоге. Для открытия файла, создания/удаления файла, перехода в другой каталог (**chdir**), система выполняет следующие действия (осуществляемые функцией **namei**() в ядре): чтение каталога и поиск в нем указанного имени файла или каталога; найденному имени соответствует номер I-узла *d_ino*; по номеру узла система считывает с диска сам I-узел нужного файла и по нему добирается до содержимого файла. Код "*выполнение*" - это как раз разрешение такого просмотра каталога **системой**. Если каталог имеет доступ на чтение - мы можем получить список файлов (т.е. применить команду **ls**); но если он при этом не имеет кода доступа "выполнение" - мы не сможем получить доступа ни к одному из файлов каталога (ни открыть, ни удалить, ни создать, ни сделать **stat**, ни **chdir**). Т.е. "*чтение*" разрешает применение вызова **read**, а "*выполнение*" - функции ядра **namei**. Фактически "*выполнение*" означает "доступ к файлам в данном каталоге"; еще более точно - к I-nodam файлов этого каталога.

t *sticky bit*

S_ISVTX - для каталога он означает, что удалить или переименовать некий файл в данном каталоге могут только: владелец каталога, владелец данного файла, суперпользователь. И никто другой. Это исключает удаление файлов чужими.

Совет: для каталога полезно иметь такие коды доступа:

```
chmod o-w,+t каталог
```

В системах **BSD** используется, как уже было упомянуто, формат каталога с переменной длиной записей. Чтобы иметь удобный доступ к именам в каталоге, возникли специальные функции чтения каталога: **opendir**, **closedir**, **readdir**. Покажем, как простейшая команда **ls** реализуется через эти функции.

£ Суперпользователь (*superuser*) имеет *uid*==0. Это "привелегированный" пользователь, который имеет право делать ВСЕ. Ему доступны любые сисвызовы и файлы, несмотря на коды доступа и.т.п.

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int listdir(char *dirname){
    register struct dirent *dirbuf;
    DIR *fddir;
    ino_t dot_ino = 0, dotdot_ino = 0;

    if((fddir = opendir (dirname)) == NULL){
        fprintf(stderr, "Can't read %s\n", dirname);
        return 1;
    }
    /* Без сортировки по алфавиту */
    while ((dirbuf = readdir (fddir)) != NULL ) {
        if (dirbuf->d_ino == 0) continue;
        if (strcmp (dirbuf->d_name, "." ) == 0){
                dot_ino = dirbuf->d_ino;
                continue;
        } else if(strcmp (dirbuf->d_name, "..") == 0){
                dotdot_ino = dirbuf->d_ino;
                continue;
        } else printf("%s\n", dirbuf->d_name);
    }
    closedir (fddir);

    if(dot_ino    == 0) printf("Поврежденный каталог: нет имени \".\"\n");
    if(dotdot_ino == 0) printf("Поврежденный каталог: нет имени \"..\"\n");
    if(dot_ino && dot_ino == dotdot_ino)  printf("Это корневой каталог диска\n");

    return 0;
}

int main(int ac, char *av[]){
    int i;

    if(ac > 1) for(i=1; i < ac; i++) listdir(av[i]);
    else                             listdir(".");

    return 0;
}
```

Обратите внимание, что тут не требуется добавление '\0' в конец поля **d_name**, поскольку его предоставляет нам сама функция **readdir**().

6.1.4. Напишите программу удаления файлов и каталогов, заданных в *argv*. Делайте **stat**, чтобы определить тип файла (файл/каталог). Программа должна отказываться удалять файлы устройств.

Для удаления пустого каталога (не содержащего иных имен, кроме "." и "..") следует использовать сисвызов

rmdir(*имя_каталога*);

(если каталог не пуст - *errno* получит значение **EEXIST**); а для удаления обычных файлов (не каталогов)

unlink(*имя_файла*);

Программа должна запрашивать подтверждение на удаление каждого файла, выдавая его имя, тип, размер в килобайтах и вопрос "удалить ?".

6.1.5. Напишите функцию рекурсивного обхода дерева подкаталогов и печати имен всех файлов в нем. Ключ U42 означает файловую систему с длинными именами файлов (BSD 4.2).

```
/*#!/bin/cc -DFIND -DU42 -DMATCHONLY treemk.c match.c -o tree -lx
 * Обход поддерева каталогов (по мотивам Керниган & Ритчи).
 *              Ключи компиляции:
 * BSD-4.2 BSD-4.3                      -DU42
 * XENIX с канонической файл.сист.       ничего
 * XENIX с библиотекой  -lx             -DU42
 *      программа поиска файлов                          -DFIND
 *      программа рекурсивного удаления                  -DRM_REC
 *      программа подсчета используемого места на диске  БЕЗ_КЛЮЧА
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/param.h>         /* для MAXPATHLEN */

#if defined(M_XENIX) && defined(U42)
# include <sys/ndir.h>  /* XENIX + U42 эмуляция */
#else
# include <dirent.h>
# define stat(f,s) lstat(f,s)  /* не проходить по символьным ссылкам */
# define d_namlen d_reclen
#endif

/* проверка: каталог ли это */
#define  isdir(st) ((st.st_mode & S_IFMT) == S_IFDIR)
struct   stat st;               /* для сисвызова stat() */
char     buf[MAXPATHLEN+1];     /* буфер для имени файла */

#define FAILURE (-1)            /* код неудачи */
#define SUCCESS   1             /* код успеха  */
#define WARNING   0             /* нефатальная ошибка */
/* Сообщения об ошибках во время обхода дерева: */
#ifndef ERR_CANT_READ
# define ERR_CANT_READ(name)  \
         fprintf( stderr, "\tНе могу читать \"%s\"\n", name), WARNING
# define ERR_NAME_TOO_LONG()  \
         fprintf( stderr, "\tСлишком длинное полное имя\n" ), WARNING
#endif

/* Прототипы для предварительного объявления функций. */
extern char *strrchr(char *, char);
int directory (char *name, int level,
    int (*enter)(char *full, int level, struct stat *st),
    int (*leave)(char *full, int level),
    int (*touch)(char *full, int level, struct stat *st));
/* Функции-обработчики enter, leave, touch должны
 * возвращать (-1) для прерывания просмотра дерева,
 * либо значение >= 0 для продолжения. */

/* Обойти дерево с корнем в rootdir */
int walktree (
    char *rootdir,      /* корень дерева */
    int (*enter)(char *full, int level, struct stat *st),
    int (*leave)(char *full, int level),
    int (*touch)(char *full, int level, struct stat *st)
){
    /* проверка корректности корня */
    if( stat(rootdir, &st) < 0 || !isdir(st)){
        fprintf( stderr, "\tПлохой корень дерева \"%s\"\n", rootdir );
        return   FAILURE;  /* неудача */
    }
    strcpy     (buf, rootdir);
    return act (buf, 0, enter, leave, touch);
}
```

```
/* Оценка файла с именем name.
 */
int act (char *name, int level,
    int (*enter)(char *full, int level, struct stat *st),
    int (*leave)(char *full, int level),
    int (*touch)(char *full, int level, struct stat *st))
{
    if (stat (name, &st) < 0)
        return WARNING; /* ошибка, но не фатальная      */
    if(isdir(st)){      /* позвать обработчик каталогов */
       if(enter)
          if( enter(name, level, &st) == FAILURE ) return FAILURE;
       return directory (name, level+1, enter, leave, touch);

    } else {            /* позвать обработчик файлов    */
       if(touch) return touch (name, level, &st);
       else      return SUCCESS;
    }
}
```

```
/* Обработать каталог: прочитать его и найти подкаталоги */
int directory (char *name, int level,
    int (*enter)(char *full, int level, struct stat *st),
    int (*leave)(char *full, int level),
    int (*touch)(char *full, int level, struct stat *st))
{
#ifndef U42
    struct direct   dirbuf;
    int        fd;
#else
    register struct dirent *dirbuf;
    DIR    *fd;
    extern DIR *opendir();
#endif
    char   *nbp, *tail, *nep;
    int     i, retcode = SUCCESS;

#ifndef U42
    if ((fd = open (name, 0)) < 0) {
#else
    if ((fd = opendir (name)) == NULL) {
#endif
        return ERR_CANT_READ(name);
    }

    tail = nbp = name + strlen (name);  /* указатель на закрывающий \0 */
    if( strcmp( name, "/" ))  /* если не "/" */
        *nbp++ = '/';
    *nbp = '\0';

#ifndef U42
    if (nbp + DIRSIZ + 2 >= name + MAXPATHLEN) {
        *tail = '\0';
        return ERR_NAME_TOO_LONG();
    }
#endif

#ifndef U42
    while (read(fd, (char *) &dirbuf, sizeof(dirbuf)) == sizeof(dirbuf)){
        if (dirbuf.d_ino == 0)  /* стертый файл */
            continue;
        if (strcmp (dirbuf.d_name, "." ) == 0  ||
            strcmp (dirbuf.d_name, "..") == 0)  /* не интересуют */
            continue;
        for (i = 0, nep = nbp; i < DIRSIZ; i++)
            *nep++ = dirbuf.d_name[i];

# else /*U42*/
    while ((dirbuf = readdir (fd)) != NULL ) {
        if (dirbuf->d_ino == 0)
            continue;
        if (strcmp (dirbuf->d_name, "." ) == 0  ||
            strcmp (dirbuf->d_name, "..") == 0)
            continue;
        for (i = 0, nep = nbp; i < dirbuf->d_namlen ; i++)
            *nep++ = dirbuf->d_name[i];
#endif /*U42*/
        *nep = '\0';
        if( act(name, level,  enter, leave, touch) == FAILURE) {
            retcode = FAILURE; break;                          }
    }
#ifndef U42
    close (fd);
#else
    closedir(fd);
#endif
```

```
    *tail = '\0';       /* восстановить старое name */

    if(retcode != FAILURE   &&   leave)
       if( leave(name, level) == FAILURE) retcode = FAILURE;
    return retcode;
}

/* ------------------------------------------------------------ */
/* Disk Usage -- Оценка места, занимаемого файлами поддерева      */
/* ------------------------------------------------------------ */
/* Пересчет байтов в килобайты */
#define KB(s)  (((s)/1024L) + ((s)%1024L ? 1L:0L))
/* или #define KB(s)   (((s) + 1024L - 1) / 1024L)  */
long size;                      /* общий размер     */
long nfiles;                    /* всего файлов     */
long ndirs;                     /* из них каталогов */
#define WARNING_LIMIT 150L      /* подозрительно большой файл */

static int du_touch (char *name, int level, struct stat *st){
     long sz;
     size += (sz = KB(st->st_size));  /* размер файла в Кб. */
     nfiles++;
#ifndef TREEONLY
     if( sz >= WARNING_LIMIT )
        fprintf(stderr,"\tВнимание! \"%s\" очень большой: %ld Кб.\n",
                                     name,               sz);
#endif /*TREEONLY*/
     return SUCCESS;
}
static int du_enter (char *name, int level, struct stat *st){
#ifndef TREEONLY
     fprintf( stderr, "Каталог \"%s\"\n", name );
#endif
     size += KB(st->st_size);  /* размер каталога в Кб. */
     nfiles++; ++ndirs; return SUCCESS;
}
long du (char *name){
     size = nfiles = ndirs = 0L;
     walktree(name, du_enter, NULL, du_touch );
     return size;
}

/* ------------------------------------------------------------ */
/* Рекурсивное удаление файлов и каталогов                      */
/* ------------------------------------------------------------ */
int  deleted;    /* сколько файлов и каталогов удалено */
static int recrm_dir (char *name, int level){
     if( rmdir(name) >= 0){ deleted++; return SUCCESS; }
     fprintf(stderr, "Не могу rmdir '%s'\n", name); return WARNING;
}

static int recrm_file(char *name, int level, struct stat *st){
     if( unlink(name) >= 0){ deleted++; return SUCCESS; }
     fprintf(stderr, "Не могу rm    '%s'\n", name); return WARNING;
}
int recrmdir(char *name){
    int ok_code; deleted = 0;
    ok_code = walktree(name, NULL, recrm_dir, recrm_file);
    printf("Удалено %d файлов и каталогов в %s\n", deleted, name);
    return ok_code;
}
```

```
/* ------------------------------------------------------------ */
/* Поиск файлов с подходящим именем (по шаблону имени)          */
/* ------------------------------------------------------------ */
char *find_PATTERN;
static int find_check(char *fullname, int level, struct stat *st){
    char *basename = strrchr(fullname, '/');
    if(basename) basename++;
    else         basename = fullname;
    if( match(basename, find_PATTERN))
        printf("Level#%02d %s\n", level, fullname);
    if( !strcmp( basename, "core")){
        printf("Найден дамп %s, поиск прекращен.\n", fullname);
        return FAILURE;
    }
    return SUCCESS;
}
void find (char *root, char *pattern){
     find_PATTERN = pattern;
     walktree(root, find_check, NULL, find_check);
}

/* ------------------------------------------------------------ */
#ifndef TREEONLY
void main(int argc, char *argv[]){
#ifdef FIND
     if(argc != 3){ fprintf(stderr, "Arg count\n"); exit(1); }
     find(argv[1], argv[2]);
#else
# ifdef RM_REC
     for(argv++; *argv; argv++)
         recrmdir(*argv);
# else
     du( argc == 1 ? "." : argv[1] );
     printf( "%ld килобайт в %ld файлах.\n", size, nfiles );
     printf( "%ld каталогов.\n", ndirs );
# endif
#endif
     exit(0);
}
#endif /*TREEONLY*/
```

6.1.6. Используя предыдущий алгоритм, напишите программу рекурсивного копирования поддерева каталогов в другое место. Для создания новых каталогов используйте системный вызов
mkdir(*имя_каталога*, *коды_доступа*);

6.1.7. Используя тот же алгоритм, напишите программу удаления каталога, которая удаляет все файлы в нем и, рекурсивно, все его подкаталоги. Таким образом, удаляется дерево каталогов. В **UNIX** подобную операцию выполняет команда
rm -r *имя_каталога_корня_дерева*

6.1.8. Используя все тот же алгоритм обхода, напишите аналог команды **find**, который будет позволять:

- находить все файлы, чьи имена удовлетворяют заданному шаблону (используйте функцию **match**() из главы "Текстовая обработка");
- находить все выполняемые файлы: обычные файлы **S_IFREG**, у которых

```
(st.st_mode & 0111) != 0
```

Как уже ясно, следует пользоваться вызовом **stat** для проверки каждого файла.

6.2. Время в UNIX.

6.2.1. Напишите функцию, переводящую год, месяц, день, часы, минуты и секунды в число секунд, прошедшее до указанного момента с 00 часов 00 минут 00 секунд 1 Января 1970 года. Внимание: результат должен иметь тип **long** (точнее **time_t**).

Эта функция облегчит вам сравнение двух моментов времени, заданных в общепринятом "человеческом" формате, поскольку сравнить два **long** числа гораздо проще, чем сравнивать по очереди годы, затем, если они равны - месяцы, если месяцы равны - даты, и.т.д.; а также облегчит измерение интервала между двумя событиями - он вычисляется просто как разность двух чисел. В системе **UNIX** время обрабатывается и хранится именно в виде числа секунд; в частности текущее астрономическое время можно узнать системным вызовом

```
#include <sys/types.h>
#include <time.h>
time_t t = time(NULL);  /* time(&t); */
```

Функция

```
struct tm *tm = localtime( &t );
```

разлагает число секунд на отдельные составляющие, содержащиеся в int-полях структуры:

```
tm_year  год           (надо прибавлять 1900)
tm_yday  день в году   0..365
tm_mon   номер месяца  0..11 (0 - Январь)
tm_mday  дата месяца   1..31
tm_wday  день недели   0..6  (0 - Воскресенье)
tm_hour  часы          0..23
tm_min   минуты        0..59
tm_sec   секунды       0..59
```

Номера месяца и дня недели начинаются с нуля, чтобы вы могли использовать их в качестве индексов:

```
char *months[] = { "Январь", "Февраль", ..., "Декабрь" };
printf( "%s\n", months[ tm->tm_mon ] );
```

Пример использования этих функций есть в приложении.

Установить время в системе может суперпользователь вызовом
stime(&*t*);

6.2.2. Напишите функцию печати текущего времени в формате ЧЧ:ММ:СС ДД-МЕС-ГГ. Используйте системный вызов **time**() и функцию **localtime**().

Существует стандартная функция **ctime**(), которая печатает время в формате:

```
/* Mon Mar 25 18:56:36 1991 */
#include <stdio.h>
#include <time.h>
main(){ /* команда date */
        time_t t = time(NULL);
        char *s  = ctime(&t);
        printf("%s", s);
}
```

Обратите внимание, что строка *s* уже содержит на конце символ '**\n**'.

6.2.3. Структура **stat**, заполняемая системным вызовом **stat**(), кроме прочих полей содержит поля типа **time_t** *st_ctime*, *st_mtime* и *st_atime* - время последнего изменения содержимого I-узла файла, время последнего изменения файла и время последнего доступа к файлу.

- Поле *st_ctime* изменяется (устанавливается равным текущему астрономическому времени) при применении к файлу вызовов **creat**, **chmod**, **chown**, **link**, **unlink**, **mknod**, **utime**†, **write** (т.к. изменяется длина файла); Это поле следует рассматривать как время модификации прав доступа к файлу;
- *st_mtime* - **write**, **creat**, **mknod**, **utime**; Это поле следует рассматривать как время модификации содержимого файла (данных);
- *st_atime* - **read**, **creat**, **mknod**, **utime**; Это поле следует рассматривать как время чтения содержимого файла (данных).

---

† Время модификации файла можно изменить на текущее астрономическое время и не производя записи в файл. Для этого используется вызов

utime(*имяФайла*, NULL);

Он используется для взаимодействия с программой **make** - в команде **touch**. Изменить время можно только своему файлу.

Модифицируйте функцию **typeOf**(), чтобы она печатала еще и эти даты.

6.2.4. Напишите аналог команды **ls -tm**, выдающей список имен файлов текущего каталога, отсортированный по убыванию поля *st_mtime*, то есть недавно модифицированные файлы выдаются первыми. Для каждого прочитанного из каталога имени надо сделать **stat**; имена файлов и времена следует сохранить в массиве структур, а затем отсортировать его.

6.2.5. Напишите аналогичную программу, сортирующую файлы в порядке возрастания их размера (*st_size*).

6.2.6. Напишите аналог команды **ls -l**, выдающий имена файлов каталога и их коды доступа в формате *rwxrw-r--*. Для получения кодов доступа используйте вызов **stat**

```
stat( имяФайла, &st);
кодыДоступа = st.st_mode & 0777;
```

Для изменения кодов доступа используется вызов

```
chmod(имя_файла, новые_коды);
```

Можно изменять коды доступа, соответствующие битовой маске

```
0777 | S_ISUID | S_ISGID | S_ISVTX
```

(смотри <*sys/stat.h*>). *Тип* файла (см. функцию **typeOf**) не может быть изменен. Изменить коды доступа к файлу может только его владелец.

Печатайте еще номер I-узла файла: поле *d_ino* каталога либо поле *st_ino* структуры **stat**.

6.2.7. Вот программа, которая каждые 2 секунды проверяет - не изменилось ли содержимое текущего каталога:

```
#include <sys/types.h>
#include <sys/stat.h>
extern char *ctime();
main(){
   time_t last; struct stat st;
   for( stat(".", &st), last=st.st_mtime; ; sleep(2)){
        stat(".", &st);
        if(last != st.st_mtime){
           last  = st.st_mtime;
printf("Был создан или удален какой-то файл: %s",
                   ctime(&last));
        }
   }
}
```

Модифицируйте ее, чтобы она сообщала *какое* имя (имена) было удалено или создано (для этого надо при запуске программы прочитать и запомнить содержимое каталога, а при обнаружении модификации - перечитать каталог и сравнить его с прежним содержимым).

6.2.8. Напишите по аналогии программу, которая выдает сообщение, если указанный вами файл был кем-то прочитан, записан или удален. Вам следует отслеживать изменение полей *st_atime*, *st_mtime* и значение **stat()** < 0 соответственно. Если файл удален - программа завершается.

6.2.9. Современные **UNIX**-машины имеют встроенные таймеры (как правило несколько) с довольно высоким разрешением. Некоторые из них могут использоваться как "будильники" с обратным отсчетом времени: в таймер загружается некоторое значение; таймер ведет обратный отсчет, уменьшая загруженный счетчик; как только это время истекает - посылается сигнал процессу, загрузившему таймер.

Вот как, к примеру, выглядит функция задержки в микросекундах (миллионных долях секунды). Примечание: эту функцию не следует использовать вперемежку с функциями **sleep** и **alarm** (смотри статью про них ниже, в главе про сигналы).

```
#include <sys/types.h>
#include <signal.h>
#include <sys/time.h>

void do_nothing() {}

/* Задержка на usec миллионных долей секунды (микросекунд) */
void usleep(unsigned int usec) {

        struct itimerval        new, old;
        /*  struct itimerval  содержит поля:
            struct timeval    it_interval;
            struct timeval    it_value;

            Где struct timeval содержит поля:
            long    tv_sec;    -- число целых секунд
            long    tv_usec;   -- число микросекунд
         */
        struct sigaction        new_vec, old_vec;

        if (usec == 0) return;

        /* Поле tv_sec  содержит число целых секунд.
           Поле tv_usec содержит число микросекунд.

           it_value    - это время, через которое В ПЕРВЫЙ раз
                         таймер "прозвонит",
                         то есть пошлет нашему процессу
                         сигнал SIGALRM.

                         Время, равное нулю, немедленно остановит таймер.

           it_interval - это интервал времени, который будет загружаться
                         в таймер после каждого "звонка"
                         (но не в первый раз).

                         Время, равное нулю, остановит таймер
                         после его первого "звонка".
         */
        new.it_interval.tv_sec  = 0;
        new.it_interval.tv_usec = 0;
        new.it_value.tv_sec  = usec / 1000000;
        new.it_value.tv_usec = usec % 1000000;

        /* Сохраняем прежнюю реакцию на сигнал SIGALRM в old_vec,
           заносим в качестве новой реакции do_nothing()
         */
        new_vec.sa_handler = do_nothing;
        sigemptyset(&new_vec.sa_mask);
        new_vec.sa_flags = 0;

        sighold(SIGALRM);
        sigaction(SIGALRM, &new_vec, &old_vec);

        /* Загрузка интервального таймера значением new, начало отсчета.
         * Прежнее значение спасти в old.
         * Вместо &old можно также NULL - не спасать.
         */
        setitimer(ITIMER_REAL, &new, &old);

        /* Ждать прихода сигнала SIGALRM */
        sigpause(SIGALRM);
```

```
        /* Восстановить реакцию на SIGALRM */
        sigaction(SIGALRM, &old_vec, (struct sigaction *) 0);
        sigrelse(SIGALRM);

        /* Восстановить прежние параметры таймера */
        setitimer(ITIMER_REAL, &old, (struct itimerval *) 0);
}
```

6.2.10. Второй пример использования таймера - это таймер, отсчитывающий текущее время суток (а также дату). Чтобы получить значение этого таймера используется вызов функции **gettimeofday**

```
#include <time.h>

void main(){
        struct timeval timenow;

        gettimeofday(&timenow, NULL);
        printf("%u sec, %u msec\n",
                timenow.tv_sec,
                timenow.tv_usec
        );
        printf("%s", ctime(&timenow.tv_sec));
        exit(0);
}
```

Поле **tv_sec** содержит число секунд, прошедшее с полуночи 1 января 1970 года до данного момента; в чем полностью соответствует системному вызову **time**. Однако плюс к тому поле **tv_usec** содержит число миллионных долей текущей секунды (значение этого поля всегда меньше 1000000).

6.2.11. К данному параграфу вернитесь, изучив раздел про **fork**() и **exit**(). Каждый процесс может пребывать в двух фазах: системной (внутри тела системного вызова - его выполняет для нас ядро операционной системы) и пользовательской (внутри кода самой программы). Время, затраченное процессом в каждой фазе, может быть измеряно системным вызовом **times**(). Кроме того, этот вызов позволяет узнать суммарное время, затраченное порожденными процессами (порожденными при помощи **fork**). Системный вызов заполняет структуру

```
struct tms {
        clock_t tms_utime;
        clock_t tms_stime;
        clock_t tms_cutime;
        clock_t tms_cstime;
};
```

и возвращает значение

```
#include <sys/times.h>

struct tms time_buf;
clock_t real_time = times(&time_buf);
```

Все времена измеряются в "тиках" - некоторых долях секунды. Число тиков в секунде можно узнать таким системным вызовом (в системе **Solaris**):

```
#include <unistd.h>
clock_t HZ = sysconf(_SC_CLK_TCK);
```

В старых системах, где таймер работал от сети переменного тока, это число получалось равным 60 (60 Герц - частота сети переменного тока). В современных системах это 100.

Поля структуры содержат:

tms_utime
: время, затраченное вызывающим процессом в пользовательской фазе.

tms_stime
: время, затраченное вызывающим процессом в системной фазе.

tms_cutime
: время, затраченное порожденными процессами в пользовательской фазе: оно равно сумме всех *tms_utime* и *tms_cutime* порожденных процессов (рекурсивное суммирование).

tms_cstime
: время, затраченное порожденными процессами в системной фазе: оно равно сумме всех *tms_stime* и

tms_cstime порожденных процессов (рекурсивное суммирование).

real_time
время, соответствующее астрономическому времени системы. Имеет смысл мерять только их разность.

Вот пример программы:

```
#include <stdio.h>
#include <unistd.h>     /* _SC_CLK_TCK */
#include <signal.h>     /* SIGALRM */
#include <sys/time.h>   /* не используется */
#include <sys/times.h>  /* struct tms */

struct tms tms_stop,  tms_start;
clock_t    real_stop, real_start;

clock_t HZ;     /* число ticks в секунде */

/* Засечь время момента старта процесса */
void hello(void){
        real_start = times(&tms_start);
}
/* Засечь время окончания процесса */
void bye(int n){
        real_stop = times(&tms_stop);
#ifdef CRONO
        /* Разность времен */
        tms_stop.tms_utime -= tms_start.tms_utime;
        tms_stop.tms_stime -= tms_start.tms_stime;
#endif

        /* Распечатать времена */
        printf("User   time          = %g seconds [%lu ticks]\n",
          tms_stop.tms_utime / (double)HZ, tms_stop.tms_utime);
        printf("System time          = %g seconds [%lu ticks]\n",
          tms_stop.tms_stime / (double)HZ, tms_stop.tms_stime);
        printf("Children user   time = %g seconds [%lu ticks]\n",
          tms_stop.tms_cutime / (double)HZ, tms_stop.tms_cutime);
        printf("Children system time = %g seconds [%lu ticks]\n",
          tms_stop.tms_cstime / (double)HZ, tms_stop.tms_cstime);
        printf("Real time            = %g seconds [%lu ticks]\n",
          (real_stop - real_start) / (double)HZ, real_stop - real_start);
        exit(n);
}

/* По сигналу SIGALRM - завершить процесс */
void onalarm(int nsig){
        printf("Выход #%d ================\n", getpid());
        bye(0);
}
/* Порожденный процесс */
void dochild(int n){
        hello();
        printf("Старт #%d ================\n", getpid());
        signal(SIGALRM, onalarm);

        /* Заказать сигнал SIGALRM через 1 + n*3 секунд */
        alarm(1 + n*3);

        for(;;){}       /* зациклиться в user mode */
}
```

```
#define NCHLD 4
int main(int ac, char *av[]){
        int i;

        /* Узнать число тиков в секунде */
        HZ = sysconf(_SC_CLK_TCK);
        setbuf(stdout, NULL);

        hello();
        for(i=0; i < NCHLD; i++)
                if(fork() == 0)
                        dochild(i);
        while(wait(NULL) > 0);
        printf("Выход MAIN =================\n");
        bye(0);
        return 0;
}
```

и ее выдача:

```
Старт #3883 ================
Старт #3884 ================
Старт #3885 ================
Старт #3886 ================
Выход #3883 ================
User   time          = 0.72 seconds [72 ticks]
System time          = 0.01 seconds [1 ticks]
Children user   time = 0 seconds [0 ticks]
Children system time = 0 seconds [0 ticks]
Real time            = 1.01 seconds [101 ticks]
Выход #3884 ================
User   time          = 1.88 seconds [188 ticks]
System time          = 0.01 seconds [1 ticks]
Children user   time = 0 seconds [0 ticks]
Children system time = 0 seconds [0 ticks]
Real time            = 4.09 seconds [409 ticks]
Выход #3885 ================
User   time          = 4.41 seconds [441 ticks]
System time          = 0.01 seconds [1 ticks]
Children user   time = 0 seconds [0 ticks]
Children system time = 0 seconds [0 ticks]
Real time            = 7.01 seconds [701 ticks]
Выход #3886 ================
User   time          = 8.9 seconds [890 ticks]
System time          = 0 seconds [0 ticks]
Children user   time = 0 seconds [0 ticks]
Children system time = 0 seconds [0 ticks]
Real time            = 10.01 seconds [1001 ticks]
Выход MAIN =================
User   time          = 0.01 seconds [1 ticks]
System time          = 0.04 seconds [4 ticks]
Children user   time = 15.91 seconds [1591 ticks]
Children system time = 0.03 seconds [3 ticks]
Real time            = 10.41 seconds [1041 ticks]
```

Обратите внимание, что 72+188+441+890=1591 (поле tms_cutime для main).

6.2.12. Еще одна программа: хронометрирование выполнения другой программы. Пример: **timer** ls -l

```
/* Хронометрирование выполнения программы */
#include <stdio.h>
#include <unistd.h>
#include <sys/times.h>

extern errno;

typedef struct _timeStamp {
        clock_t real_time;
        clock_t cpu_time;
        clock_t child_time;
        clock_t child_sys, child_user;
} TimeStamp;

TimeStamp TIME(){
        struct tms tms;
        TimeStamp  st;

        st.real_time  = times(&tms);
        st.cpu_time   = tms.tms_utime +
                        tms.tms_stime +
                        tms.tms_cutime +
                        tms.tms_cstime;
        st.child_time = tms.tms_cutime +
                        tms.tms_cstime;
        st.child_sys  = tms.tms_cstime;
        st.child_user = tms.tms_cutime;
        return st;
}

void PRTIME(TimeStamp start, TimeStamp stop){
        clock_t HZ = sysconf(_SC_CLK_TCK);
        clock_t real_time  = stop.real_time  - start.real_time;
        clock_t cpu_time   = stop.cpu_time   - start.cpu_time;
        clock_t child_time = stop.child_time - start.child_time;

        printf("%g real, %g cpu, %g child (%g user, %g sys), %ld%%\n",
                real_time       / (double)HZ,
                cpu_time        / (double)HZ,
                child_time      / (double)HZ,
                stop.child_user / (double)HZ,
                stop.child_sys  / (double)HZ,
                (child_time * 100L) / (real_time ? real_time : 1)
        );
}

TimeStamp start, stop;

int main(int ac, char *av[]){
        char *prog = *av++;
        if(*av == NULL){
                fprintf(stderr, "Usage: %s command [args...]\n", prog);
                return(1);
        }
        start = TIME();
        if(fork() == 0){
                execvp(av[0], av);
                perror(av[0]);
                exit(errno);
        }
        while(wait(NULL) > 0);
        stop = TIME();
        PRTIME(start, stop);
        return(0);
}
```

6.3. Свободное место на диске.

6.3.1. Системный вызов **ustat**() позволяет узнать количество свободного места в файловой системе, содержащей заданный файл (в примере ниже - текущий каталог):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <ustat.h>
struct stat st; struct ustat ust;
void main(int ac, char *av[]){
     char *file = (ac==1 ? "." : av[1]);
     if( stat(file, &st) < 0) exit(1);
     ustat(st.st_dev, &ust);
     printf("На диске %*.*s\n"
       "%ld свободных блоков (%ld Кб)\n"
       "%d свободных I-узлов\n",
     sizeof ust.f_fname, sizeof ust.f_fname,
     ust.f_fname, /* название файловой системы (метка) */
     ust.f_tfree, /* блоки по 512 байт */
    (ust.f_tfree * 512L) / 1024,
     ust.f_tinode );
}
```

Обратите внимание на запись длинной строки в **printf**: строки, перечисленные последовательно, склеиваются **ANSI C** компилятором в одну длинную строку:

```
char s[] = "This is"  " a line "  "of words";
        совпадает с
char s[] = "This is a line of words";
```

6.3.2. Более правильно, однако, пользоваться сисвызовом **statvfs** - статистика по виртуальной файловой системе. Рассмотрим его в следующем примере: копирование файла с проверкой на наличие свободного места.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>              /* O_RDONLY */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/statvfs.h>
#include <sys/param.h>          /* MAXPATHLEN */

char *progname;                 /* имя программы */

void error(char *fmt, ...){
        va_list args;

        va_start(args, fmt);
        fprintf(stderr, "%s: ", progname);
        vfprintf(stderr, fmt, args);
        fputc('\n', stderr);
        va_end(args);
}
```

```
int copyFile(char *to, char *from){       /* куда, откуда */
        char newname[MAXPATHLEN+1];
        char answer[20];
        struct stat stf, stt;
        int fdin, fdout;
        int n, code = 0;
        char iobuf[64 * 1024];
        char *dirname = NULL, *s;

        if((fdin = open(from, O_RDONLY)) < 0){
                error("Cannot read %s", from);
                return (-1);
        }
        fstat(fdin, &stf);
        if((stf.st_mode & S_IFMT) == S_IFDIR){
                close(fdin);
                error("%s is a directory", from);
                return (-2);
        }

        if(stat(to, &stt) >= 0){
                /* Файл уже существует */

                if((stt.st_mode & S_IFMT) == S_IFDIR){
                        /* И это каталог */

                        /* Выделить последнюю компоненту пути from */
                        if((s = strrchr(from, '/')) && s[1])
                                s++;
                        else    s = from;

                        dirname = to;

                        /* Целевой файл - файл в этом каталоге */
                        sprintf(newname, "%s/%s", to, s);
                        to = newname;

                        if(stat(to, &stt) < 0)
                                goto not_exist;
                }

                if(stt.st_dev == stf.st_dev && stt.st_ino == stf.st_ino){
                        error("%s: cannot copy file to itself", from);
                        return (-3);
                }
                switch(stt.st_mode & S_IFMT){
                case S_IFBLK:
                case S_IFCHR:
                case S_IFIFO:
                        break;

                default:
                        printf("%s already exists, overwrite ? ", to);
                        fflush(stdout);

                        *answer = '\0';
                        gets(answer);

                        if(*answer != 'y'){     /* NO */
                                close(fdin);
                                return (-4);
                        }
                        break;
                }
        }
```

```
not_exist:
        printf("COPY %s TO %s\n", from, to);

        if((stf.st_mode & S_IFMT) == S_IFREG){
                /* Проверка наличия свободного места в каталоге dirname */
                struct statvfs fs;
                char tmpbuf[MAXPATHLEN+1];

                if(dirname == NULL){
                        /* To 'to' - это имя файла, а не каталога */
                        strcpy(tmpbuf, to);
                        if(s = strrchr(tmpbuf, '/')){
                                if(*tmpbuf != '/' || s != tmpbuf){
                                        /* Имена "../xxx"
                                         * и второй случай:
                                         * абсолютные имена не в корне,
                                         * то есть не "/" и не "/xxx"
                                         */
                                        *s = '\0';
                                }else{
                                        /* "/" или "/xxx" */
                                        if(s[1]) s[1] = '\0';
                                }
                                dirname = tmpbuf;
                        } else  dirname = ".";
                }

                if(statvfs(dirname, &fs) >= 0){
                        size_t size = (geteuid() == 0 ) ?
                                /* Доступно суперпользователю: байт */
                                fs.f_frsize * fs.f_bfree :
                                /* Доступно обычному пользователю: байт */
                                fs.f_frsize * fs.f_bavail;

                        if(size < stf.st_size){
                           error("Not enough free space on %s: have %lu, need %lu",
                                  dirname, size, stf.st_size);
                           close(fdin);
                           return (-5);
                        }
                }
        }

        if((fdout = creat(to, stf.st_mode)) < 0){
                error("Can't create %s", to);
                close(fdin);
                return (-6);
        } else {
                fchmod(fdout, stf.st_mode);
                fchown(fdout, stf.st_uid, stf.st_gid);
        }

        while (n = read (fdin, iobuf, sizeof iobuf)) {
                if(n < 0){
                        error ("read error");
                        code = (-7);
                        goto done;
                }
                if(write (fdout, iobuf, n) != n) {
                        error ("write error");
                        code = (-8);
                        goto done;
                }
        }
```

```
done:
        close (fdin);
        close (fdout);

        /* Проверить: соответствует ли результат ожиданиям */
        if(stat(to, &stt) >= 0 && (stt.st_mode & S_IFMT) == S_IFREG){
                if(stf.st_size < stt.st_size){
                        error("File has grown at the time of copying");
                } else if(stf.st_size > stt.st_size){
                        error("File too short, target %s removed", to);
                        unlink(to);
                        code = (-9);
                }
        }
        return code;
}

int main(int argc, char *argv[]){
        int i, code = 0;

        progname = argv[0];

        if(argc < 3){
                error("Usage: %s from... to", argv[0]);
                return 1;
        }
        for(i=1; i < argc-1; i++)
                code |= copyFile(argv[argc-1], argv[i]) < 0 ? 1 : 0;
        return code;
}
```

Возвращаемая структура **struct statvfs** содержит такие поля (в частности):

```
Типа long:
f_frsize                размер блока
f_blocks                размер файловой системы в блоках
f_bfree                 свободных блоков (для суперпользователя)
f_bavail                свободных блоков (для всех остальных)

f_files                 число I-nodes в файловой системе
f_ffree                 свободных I-nodes (для суперпользователя)
f_favail                свободных I-nodes (для всех остальных)

Типа char *
f_basetype              тип файловой системы: ufs, nfs, ...
```

По два значения дано потому, что операционная система резервирует часть файловой системы для использования ТОЛЬКО суперпользователем (чтобы администратор смог распихать файлы в случае переполнения диска, и имел резерв на это). **ufs** - это UNIX file system из BSD 4.x

6.4. Сигналы.

Процессы в **UNIX** используют много разных механизмов взаимодействия. Одним из них являются *сигналы*.

Сигналы - это *асинхронные* события. Что это значит? Сначала объясним, что такое *синхронные* события: я два раза в день подхожу к почтовому ящику и проверяю - нет ли в нем почты (событий). Во-первых, я произвожу *опрос* - "нет ли для меня события?", в программе это выглядело бы как вызов функции опроса и, может быть, ожидания события. Во-вторых, я знаю, что почта *может* ко мне прийти, поскольку я подписался на какие-то газеты. То есть я предварительно *заказывал* эти события.

Схема с синхронными событиями очень распространена. Кассир сидит у кассы и ожидает, пока к нему в окошечко не заглянет клиент. Поезд периодически проезжает мимо светофора и останавливается, если горит красный. Функция Си пассивно "спит" до тех пор, пока ее не вызовут; однако она всегда готова выполнить свою работу (обслужить клиента). Такое ожидающее заказа (события) действующее лицо называется *сервер*. После выполнения заказа сервер вновь переходит в состояние ожидания вызова. Итак, если событие *ожидается* в специальном месте и в определенные моменты времени (издается некий вызов для ОПРОСА) - это синхронные события. Канонический пример - функция **gets**, которая задержит выполнение программы, пока с клавиатуры не будет введена строка. Большинство ожиданий внутри системных

вызовов - *синхронны*. Ядро ОС выступает для программ пользователей в роли сервера, выполняющего сисвызовы (хотя и не только в этой роли - ядро иногда предпринимает и активные действия: передача процессора другому процессу через определенное время (режим разделения времени), убивание процесса при ошибке, и.т.п.).

Сигналы - это асинхронные события. Они приходят неожиданно, в любой момент времени - вроде телефонного звонка. Кроме того, их не требуется заказывать - сигнал процессу может поступить совсем без повода. Аналогия из жизни такова: человек сидит и пишет письмо. Вдруг его окликают посреди фразы - он отвлекается, отвечает на вопрос, и вновь продолжает прерванное занятие. Человек *не ожидал* этого оклика (быть может, он *готов* к нему, но он не озирался по сторонам специально). Кроме того, сигнал мог поступить когда он писал 5-ое предложение, а мог - когда 34-ое. Момент времени, в который произойдет прерывание, не фиксирован.

Сигналы имеют *номера*, причем их количество ограничено - есть определенный список допустимых сигналов. Номера и мнемонические имена сигналов перечислены в include-файле <*signal.h*> и имеют вид **SIG***нечто*. Допустимы сигналы с номерами 1..**NSIG**-1, где **NSIG** определено в этом файле. При получении сигнала мы узнаем его номер, но не узнаем никакой иной информации: ни *от кого* поступил сигнал, ни что от нас хотят. Просто "звонит телефон". Чтобы получить дополнительную информацию, наш процесс должен взять ее из другого известного места; например - прочесть заказ из некоторого файла, об имени которого все наши программы заранее "договорились". Сигналы процессу могут поступать тремя путями:

- От другого процесса, который *явно* посылает его нам вызовом
  kill(*pid*, *sig*);
  где *pid* - идентификатор (номер) процесса-получателя, а *sig* - номер сигнала. Послать сигнал можно только родственному процессу - запущенному тем же пользователем.
- От операционной системы. Система может посылать процессу ряд сигналов, сигнализирующих об ошибках, например при обращении программы по несуществующему адресу или при ошибочном номере системного вызова. Такие сигналы обычно прекращают наш процесс.
- От пользователя - с клавиатуры терминала можно нажимом некоторых клавиш послать сигналы **SIGINT** и **SIGQUIT**. Собственно, сигнал посылается *драйвером терминала* при получении им с клавиатуры определенных символов. Так можно прервать зациклившуюся или надоевшую программу.

Процесс-получатель должен как-то отреагировать на сигнал. Программа может:

- проигнорировать сигнал (не ответить на звонок);
- перехватить сигнал (снять трубку), выполнить какие-то действия, затем продолжить прерванное занятие;
- быть убитой сигналом (звонок был подкреплен броском гранаты в окно);

В большинстве случаев сигнал по умолчанию убивает процесс-получатель. Однако процесс может изменить это умолчание и задать свою реакцию явно. Это делается вызовом **signal**:

```
#include <signal.h>
void (*signal(int sig, void (*react)() )) ();
```

Параметр *react* может иметь значение:

SIG_IGN
: сигнал *sig* будет отныне игнорироваться. Некоторые сигналы (например **SIGKILL**) невозможно перехватить или проигнорировать.

SIG_DFL
: восстановить реакцию по умолчанию (обычно - смерть получателя).

имя_функции
: Например

```
void fr(gotsig){ ..... }  /* обработчик */
... signal (sig, fr); ... /* задание реакции */
```

Тогда при получении сигнала *sig* будет вызвана функция **fr**, в которую в качестве аргумента *системой* будет передан номер сигнала, действительно вызвавшего ее - *gotsig==sig*. Это полезно, т.к. можно задать одну и ту же функцию в качестве реакции для *нескольких* сигналов:

```
... signal (sig1, fr); signal(sig2, fr); ...
```

После возврата из функции **fr**() программа продолжится с прерванного места. *Перед* вызовом функции-обработчика реакция автоматически сбрасывается в реакцию по умолчанию **SIG_DFL**, а после выхода из обработчика снова восстанавливается в **fr**. Это значит, что во время работы функции-обработчика может прийти сигнал, который *убьет* программу.

Приведем список *некоторых* сигналов; полное описание посмотрите в документации. Колонки таблицы: **G** - может быть перехвачен; **D** - по умолчанию убивает процесс (**k**), игнорируется (**i**); **C** - образуется дамп

памяти процесса: файл **core**, который затем может быть исследован отладчиком **adb**; **F** - реакция на сигнал сбрасывается; **S** - посылается обычно системой, а не явно.

сигнал	G	D	C	F	S	смысл
SIGTERM	+	k	-	+	-	завершить процесс
SIGKILL	-	k	-	+	-	убить процесс
SIGINT	+	k	-	+	-	прерывание с клавиш
SIGQUIT	+	k	+	+	-	прерывание с клавиш
SIGALRM	+	k	-	+	+	будильник
SIGILL	+	k	+	-	+	запрещенная команда
SIGBUS	+	k	+	+	+	обращение по неверному
SIGSEGV	+	k	+	+	+	адресу
SIGUSR1, **USR2**	+	i	-	+	-	пользовательские
SIGCLD	+	i	-	+	+	смерть потомка

- Сигнал **SIGILL** используется иногда для эмуляции команд с плавающей точкой, что происходит примерно так: при обнаружении "запрещенной" команды для отсутствующего процессора "плавающей" арифметики аппаратура дает прерывание и система посылает процессу сигнал **SIGILL**. По сигналу вызывается функция-эмулятор плавающей арифметики (подключаемая к выполняемому файлу автоматически), которая и обрабатывает требуемую команду. Это может происходить много раз, именно поэтому реакция на этот сигнал *не сбрасывается*.
- **SIGALRM** посылается в результате его заказа вызовом **alarm**() (см. ниже).
- Сигнал **SIGCLD** посылается процессу-родителю при выполнении процессом-потомком сисвызова **exit** (или при смерти вследствие получения сигнала). Обычно процесс-родитель при получении такого сигнала (если он его заказывал) реагирует, выполняя в обработчике сигнала вызов **wait** (см. ниже). По-умолчанию этот сигнал игнорируется.
- Реакция **SIG_IGN** *не* сбрасывается в **SIG_DFL** при приходе сигнала, т.е. сигнал игнорируется постоянно.
- Вызов **signal** возвращает старое значение реакции, которое может быть запомнено в переменную вида void (*f)(); а потом восстановлено.
- Синхронное ожидание (сисвызов) может иногда быть прервано асинхронным событием (сигналом), но об этом ниже.

Некоторые версии **UNIX** предоставляют более развитые средства работы с сигналами. Опишем некоторые из средств, имеющихся в **BSD** (в других системах они могут быть смоделированы другими способами).

Пусть у нас в программе есть "критическая секция", во время выполнения которой приход сигналов нежелателен. Мы можем "заморозить" (заблокировать) сигнал, отложив момент его поступления до "разморозки":

```
   |
sighold(sig);  заблокировать сигнал
   |           :
 КРИТИЧЕСКАЯ   :<---процессу послан сигнал sig,
 СЕКЦИЯ        :  но он не вызывает реакцию немедленно,
   |           :  а "висит", ожидая разрешения.
   |           :
sigrelse(sig); разблокировать
   |<----------- sig
   |    накопившиеся сигналы доходят,
   |    вызывается реакция.
```

Если во время блокировки процессу было послано *несколько* одинаковых сигналов *sig*, то при разблокировании поступит *только один*. Поступление сигналов во время блокировки просто отмечается в специальной битовой шкале в паспорте процесса (примерно так):

```
mask |= (1 << (sig - 1));
```

и при разблокировании сигнала *sig*, если соответствующий бит выставлен, то приходит один такой сигнал (система вызывает функцию реакции).

То есть **sighold** заставляет приходящие сигналы "накапливаться" в специальной маске, вместо того, чтобы немедленно вызывать реакцию на них. А **sigrelse** разрешает "накопившимся" сигналам (если они есть) прийти и вызывает реакцию на них.

Функция

sigset(*sig*, *react*);

аналогична функции **signal**, за исключением того, что на время работы обработчика сигнала *react*, приход сигнала *sig* блокируется; то есть перед вызовом *react* как бы делается **sighold**, а при выходе из

обработчика - **sigrelse**. Это значит, что если во время работы обработчика сигнала придет такой же сигнал, то программа не будет убита, а "запомнит" пришедший сигнал, и обработчик будет вызван повторно (когда сработает **sigrelse**).

Функция
sigpause(*sig*);
вызывается внутри "рамки"

```
sighold(sig);
        ...
    sigpause(sig);
        ...
sigrelse(sig);
```

и вызывает задержку выполнения процесса до прихода сигнала *sig*. Функция разрешает приход сигнала *sig* (обычно на него должна быть задана реакция при помощи **sigset**), и "засыпает" до прихода сигнала *sig*.

В **UNIX** стандарта **POSIX** для управления сигналами есть вызовы **sigaction**, **sigprocmask**, **sigpending**, **sigsuspend**. Посмотрите в документацию!

6.4.1. Напишите программу, выдающую на экран файл */etc/termcap*. Перехватывайте сигнал **SIGINT**, при получении сигнала запрашивайте "*Продолжать?*". По ответу 'y' - продолжить выдачу; по 'n' - завершить программу; по 'r' - начать выдавать файл с начала: **lseek**(*fd*,0L,0). Не забудьте заново переустановить реакцию на **SIGINT**, поскольку после получения сигнала реакция автоматически сбрасывается.

```
#include <signal.h>
void onintr(sig){       /* sig - номер сигнала  */
  signal (sig, onintr); /* восстановить реакцию */
  ... запрос и действия ...
}
main(){ signal (SIGINT, onintr); ... }
```

Сигнал прерывания можно игнорировать. Это делается так:

```
signal (SIGINT, SIG_IGN);
```

Такую программу нельзя прервать с клавиатуры. Напомним, что реакция **SIG_IGN** сохраняется при приходе сигнала.

6.4.2. Системный вызов, находящийся в состоянии ожидания какого-то события (**read** ждущий нажатия кнопки на клавиатуре, **wait** ждущий окончания процесса-потомка, и.т.п.), может быть *прерван* сигналом. При этом сисвызов вернет значение "ошибка" (-1) и **errno** станет равно **EINTR**. Это позволяет нам писать системные вызовы с выставлением *таймаута*: если событие не происходит в течение заданного времени, то завершить ожидание и прервать сисвызов. Для этой цели используется вызов **alarm**(*sec*), заказывающий посылку сигнала **SIGALRM** нашей программе через целое число *sec* секунд (0 - отменяет заказ):

```
#include <signal.h>
void (*oldaction)(); int alarmed;
/* прозвонил будильник */
void onalarm(nsig){ alarmed++; }
    ...
/* установить реакцию на сигнал */
oldaction = signal (SIGALRM, onalarm);
/* заказать будильник через TIMEOUT сек. */
alarmed = 0; alarm ( TIMEOUT /* sec */ );

     sys_call(...);  /* ждет события */
  // если нас сбил сигнал, то по сигналу будет
  // еще вызвана реакция на него - onalarm

if(alarmed){
  // событие так и не произошло.
  // вызов прерван сигналом т.к. истекло время.
}else{
  alarm(0); /* отменить заказ сигнала */
  // событие произошло, сисвызов успел
  // завершиться до истечения времени.
}
signal (SIGALRM, oldaction);
```

Напишите программу, которая ожидает ввода с клавиатуры в течение 10 секунд. Если ничего не введено -

печатает "*Нет ввода*", иначе - печатает "*Спасибо*". Для ввода можно использовать как вызов **read**, так и функцию **gets** (или **getchar**), поскольку функция эта все равно внутри себя издает системный вызов **read**. Исследуйте, какое значение возвращает **fgets** (**gets**) в случае прерывания ее системным вызовом.

```
/* Копирование стандартного ввода на стандартный вывод
 * с установленным тайм-аутом.
 * Это позволяет использовать программу для чтения из FIFO-файлов
 * и с клавиатуры.
 * Небольшая модификация позволяет использовать программу
 * для копирования "растущего" файла (т.е. такого, который в
 * настоящий момент еще продолжает записываться).
 * Замечание:
 *       В ДЕМОС-2.2 сигнал НЕ сбивает чтение из FIFO-файла,
 *       а получение сигнала откладывается до выхода из read()
 *       по успешному чтению информации. Пользуйтесь open()-ом
 *       с флагом O_NDELAY, чтобы получить требуемый эффект.
 *
 *      Вызов: a.out /dev/tty
 *
 * По мотивам книги М.Дансмура и Г.Дейвиса.
 */

#define WAIT_TIME 5 /* ждать 5 секунд */
#define MAX_TRYS  5 /* максимум 5 попыток */
#define BSIZE     256
#define STDIN     0 /* дескриптор стандартного ввода  */
#define STDOUT    1 /* дескриптор стандартного вывода */

#include <signal.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
char buffer [ BSIZE ];
extern int errno;       /* код ошибки */

void timeout(nsig){ signal( SIGALRM, timeout ); }
void main(argc, argv) char **argv;{
   int fd, n, trys = 0;  struct stat stin, stout;

   if( argc != 2 ){
       fprintf(stderr, "Вызов: %s файл\n", argv[0]); exit(1);
   }
   if((fd = !strcmp(argv[1],"-")? STDIN : open(argv[1],O_RDONLY)) < 0){
       fprintf(stderr, "Не могу читать %s\n", argv[1]); exit(2);
   }
   /* Проверить, что ввод не совпадает с выводом,
    *     hardcat aFile >> aFile
    * кроме случая, когда вывод - терминал.
    * Такая проверка полезна для программ-фильтров (STDIN->STDOUT),
    * чтобы исключить порчу исходной информации */
   fstat(fd, &stin); fstat(STDOUT, &stout);
   if( !isatty(STDOUT) && stin.st_ino == stout.st_ino &&
                          stin.st_dev == stout.st_dev
   ){ fprintf(stderr,
      "\aВвод == выводу, возможно потеряна информация в %s.\n",argv[1]);
      exit(33);
   }
```

```
    signal( SIGALRM, timeout );
    while( trys < MAX_TRYS ){
            alarm( WAIT_TIME ); /* заказать сигнал через 5 сек */

            /* и ждем ввода ... */
            n = read( fd, buffer, BSIZE );

            alarm(0);       /* отменили заказ сигнала */
                  /* (хотя, возможно, он уже получен) */

            /* проверяем: почему мы слезли с вызова read() ? */
            if( n < 0 && errno == EINTR ){
                /* Мы были сбиты сигналом SIGALRM,
                 * код ошибки EINTR - сисвызов прерван
                 * неким сигналом.
                 */
                fprintf( stderr, "\7timed out (%d раз)\n", ++trys );
                continue;
            }

            if( n < 0 ){
                /* ошибка чтения */
                fprintf( stderr, "read error.\n" ); exit(4);
            }
            if( n == 0 ){
                /* достигнут конец файла */
                fprintf( stderr, "Достигнут EOF.\n\n" ); exit(0);
            }
            /* копируем прочитанную информацию */
            write( STDOUT, buffer, n );
            trys = 0;
    }
    fprintf( stderr, "Все попытки провалились.\n" ); exit(5);
}
```

Если мы хотим, чтобы сисвызов не мог прерываться сигналом, мы должны защитить его:

```
#include <signal.h>
void (*fsaved)();
        ...
fsaved = signal (sig, SIG_IGN);
   sys_call(...);
signal (sig, fsaved);
```

или так:

```
sighold(sig);
   sys_call(...);
sigrelse(sig);
```

Сигналами могут быть прерваны *не все* системные вызовы и не при всех обстоятельствах.

6.4.3. Напишите функцию **sleep**(*n*), задерживающую выполнение программы на *n* секунд. Воспользуйтесь системным вызовом **alarm**(*n*) (будильник) и вызовом **pause**(), который задерживает программу до получения *любого* сигнала. Предусмотрите рестарт при получении во время ожидания другого сигнала, нежели **SIGALRM**. Сохраняйте заказ **alarm**, сделанный до вызова **sleep** (**alarm** выдает число секунд, оставшееся до завершения предыдущего заказа). На самом деле есть такая СТАНДАРТНАЯ функция. Ответ:

```
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>

int got;   /* пришел ли сигнал */

void onalarm(int sig)
{ printf( "Будильник\n" ); got++; } /* сигнал получен */
```

```
void sleep(int n){
   time_t time(), start = time(NULL);
   void (*save)();
   int oldalarm, during = n;

   if( n <= 0 ) return;
   got = 0;
   save = signal(SIGALRM, onalarm);
   oldalarm = alarm(3600); /* Узнать старый заказ */
   if( oldalarm ){
       printf( "Был заказан сигнал, который придет через %d сек.\n",
                oldalarm );
       if(oldalarm > n) oldalarm -= n;
       else { during = n = oldalarm; oldalarm = 1; }
   }
   printf( "n=%d oldalarm=%d\n", n, oldalarm );
   while( n > 0 ){
     printf( "alarm(%d)\n", n );
     alarm(n);  /* заказать SIGALRM через n секунд */

     pause();

     if(got) break;
     /* иначе мы сбиты с pause другим сигналом */
     n = during - (time(NULL) - start); /* прошло времени */
   }
   printf( "alarm(%d) при выходе\n", oldalarm );
   alarm(oldalarm);  /* alarm(0) - отмена заказа сигнала */
   signal(SIGALRM, save); /* восстановить реакцию */
}

void onintr(int nsig){
   printf( "Сигнал SIGINT\n"); signal(SIGINT, onintr);
}

void onOldAlarm(int nsig){
   printf( "Звонит старый будильник\n");
}

void main(){
        int time1 = 0;  /* 5, 10, 20 */
        setbuf(stdout, NULL);
        signal(SIGINT, onintr);
        signal(SIGALRM, onOldAlarm); alarm(time1);
                    sleep(10);
        if(time1) pause();
        printf("Чао!\n");
}
```

6.4.4. Напишите "часы", выдающие текущее время каждые 3 секунды.

```
#include <signal.h>
#include <time.h>
#include <stdio.h>
void tick(nsig){
   time_t tim; char *s;
   signal (SIGALRM, tick);
   alarm(3); time(&tim);
   s = ctime(&tim);
   s[ strlen(s)-1 ] = '\0'; /* обрубить '\n' */
   fprintf(stderr, "\r%s", s);
}
main(){ tick(0);
        for(;;) pause();
}
```

6.5. Жизнь процессов.

6.5.1. Какие классы памяти имеют данные, в каких сегментах программы они расположены?

```
char x[] = "hello";
int y[25];
char *p;
main(){
        int z = 12;
        int v;
        static int w = 25;
        static int q;
        char s[20];
        char *pp;
        ...
        v = w + z;      /* #1 */
}
```

Ответ:

Переменная	Класс памяти	Сегмент	Начальное значение
x	static	data/DATA	"hello"
y	static	data/BSS	{0, ..., 0}
p	static	data/BSS	NULL
z	auto	stack	12
v	auto	stack	не определено
w	static	data/DATA	25
q	static	data/BSS	0
s	auto	stack	не определено
pp	auto	stack	не определено
main	static	text/TEXT	

Большими буквами обозначены сегменты, хранимые в выполняемом файле:

DATA - это *инициализированные* статические данные (которым присвоены начальные значения). Они помещаются компилятором в файл в виде готовых констант, а при запуске программы (при ее загрузке в память машины), просто копируются в память из файла.

BSS (Block Started by Symbol)
- неинициализированные статические данные. Они по умолчанию имеют начальное значение 0 (NULL, "", '\0'). Эта память расписывается нулями при запуске программы, а в файле хранится лишь ее *размер*.

TEXT - сегмент, содержащий машинные команды (код).

Хранящаяся в файле выполняемая программа имеет также *заголовок* - в нем в частности содержатся размеры перечисленных сегментов и их местоположение в файле; и еще - в самом конце файла - *таблицу имен*. В ней содержатся имена всех функций и переменных, используемых в программе, и их адреса. Эта таблица используется отладчиками **adb** и **sdb**, а также при сборке программы из нескольких объектных файлов программой **ld**. Просмотреть ее можно командой

nm *имяФайла*

Для экономии дискового пространства эту таблицу часто удаляют, что делается командой

strip *имяФайла*

Размеры сегментов можно узнать командой

size *имяФайла*

Программа, загруженная в память компьютера (т.е. процесс), состоит из 3х сегментов, относящихся непосредственно к программе:

stack - стек для локальных переменных функций (автоматических переменных). Этот сегмент существует только у выполняющейся программы, поскольку отведение памяти в стеке производится выполнением некоторых машинных команд (поэтому описание автоматических переменных в Си - это на самом деле *выполняемые* операторы, хотя и не с точки зрения языка). Сегмент стека *автоматически* растет по мере надобности (если мы вызываем новые и новые функции, отводящие переменные в стеке). За этим следит аппаратура диспетчера памяти.

data - сегмент, в который склеены сегменты статических данных **DATA** и **BSS**, загруженные из файла. Этот сегмент также может изменять свой размер, но делать это надо явно - системными вызовами **sbrk** или **brk**. В частности, функция **malloc**() для размещения динамически отводимых данных увеличивает размер этого сегмента.

text - это выполняемые команды, копия сегмента **TEXT** из файла. Так строка с меткой #1 содержится в виде машинных команд именно в этом сегменте.

Кроме того, каждый процесс имеет еще:

proc - это резидентная часть паспорта процесса в таблице процессов в ядре операционной системы;

user - это 4-ый сегмент процесса - нерезидентная часть паспорта (**u-area**). К этому сегменту имеет доступ только ядро, но не сама программа.

Паспорт процесса был поделен на 2 части только из соображений экономии памяти в ядре: контекст процесса (таблица открытых файлов, ссылка на I-узел текущего каталога, таблица реакций на сигналы, ссылка на I-узел управляющего терминала, и.т.п.) нужен ядру только при обслуживании текущего активного процесса. Когда активен другой процесс - эта информация в памяти ядра не нужна. Более того, если процесс из-за нехватки места в памяти машины был откачан на диск, эта информация также может быть откачана на диск и подкачана назад лишь вместе с процессом. Поэтому контекст был выделен в отдельный сегмент, и сегмент этот подключается к адресному пространству ядра лишь при выполнении процессом какого-либо системного вызова (это подключение называется "переключение контекста" - *context switch*). Четыре сегмента процесса могут располагаться в памяти машины не обязательно подряд - между ними могут лежать сегменты других процессов.

Схема составных частей процесса:

```
          П  Р  О  Ц  Е  С  С
    таблица процессов:
    паспорт  в ядре       сегменты в памяти

     struct proc[]
          ####----------------> stack      1
          ####                  data       2
                                text       3
                  контекст:   struct user  4
```

Каждый процесс имеет уникальный номер, хранящийся в поле *p_pid* в структуре **proc**†. В ней также хранятся: адреса сегментов процесса в памяти машины (или на диске, если процесс откачан); *p_uid* - номер владельца процесса; *p_ppid* - номер процесса-родителя; *p_pri*, *p_nice* - приоритеты процесса; *p_pgrp* - группа процесса; *p_wchan* - ожидаемое процессом событие; *p_flag* и *p_stat* - состояние процесса; и многое другое. Структура **proc** определена в include-файле <*sys/proc.h*>, а структура **user** - в <*sys/user.h*>.

6.5.2. Системный вызов **fork**() (вилка) создает *новый* процесс: *копию* процесса, издавшего вызов. Отличие этих процессов состоит только в возвращаемом **fork**-ом значении:

```
0                    - в новом процессе.
pid нового процесса - в исходном.
```

Вызов **fork** может завершиться неудачей если таблица процессов переполнена. Простейший способ сделать это:

```
main(){
      while(1)
        if( ! fork()) pause();
}
```

Одно гнездо таблицы процессов зарезервировано - его может использовать только суперпользователь (в целях жизнеспособности системы: хотя бы для того, чтобы запустить программу, убивающую все эти процессы-варвары).

Вызов **fork** создает копию всех 4х сегментов процесса и выделяет порожденному процессу новый паспорт и номер. Иногда сегмент **text** не копируется, а используется процессами совместно ("*разделяемый сегмент*") в целях экономии памяти. При копировании сегмента **user** контекст порождающего процесса *наследуется* порожденным процессом (см. ниже).

Проведите опыт, доказывающий что порожденный системным вызовом **fork**() процесс и породивший его - равноправны. Повторите несколько раз программу:

```
#include <stdio.h>
int pid, i, fd; char c;
```

---

† Процесс может узнать его вызовом *pid*=**getpid**();

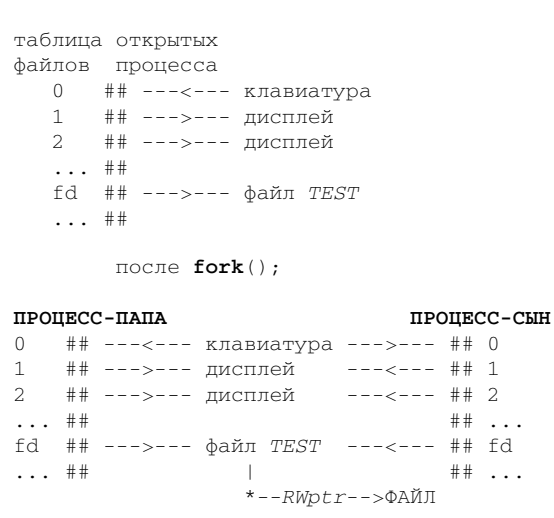
```
main(){
   fd = creat( "TEST", 0644);
   if( !(pid = fork())){ /* сын: порожденный процесс */
           c = 'a';
           for(i=0; i < 5; i++){
             write(fd, &c, 1); c++; sleep(1);
           }
           printf("Сын %d окончен\n", getpid());
           exit(0);
   }
   /* else процесс-отец */
   c = 'A';
   for(i=0; i < 5; i++){
           write(fd, &c, 1); c++; sleep(1);
   }
   printf("Родитель %d процесса %d окончен\n",
           getpid(), pid );
}
```

В файле *TEST* мы будем от случая к случаю получать строки вида

```
aABbCcDdEe  или  AaBbcdCDEe
```

что говорит о том, что первым "проснуться" после **fork**() может *любой* из двух процессов. Если же опыт дает устойчиво строки, начинающиеся с одной и той же буквы - значит в данной реализации системы один из процессов все же запускается раньше. Но не стоит использовать этот эффект - при переносе на другую систему его может не быть!

Данный опыт основан на следующем свойстве системы **UNIX**: при системном вызове **fork**() порожденный процесс получает все открытые порождающим процессом файлы "в наследство" - это соответствует тому, что таблица открытых процессом файлов копируется в процесс-потомок. Именно так, в частности, передаются от отца к сыну стандартные каналы 0, 1, 2: порожденному процессу не нужно открывать стандартные ввод, вывод и вывод ошибок явно. Изначально же они открываются специальной программой при вашем входе в систему.



Ссылки из таблиц открытых файлов в процессах указывают на структуры "открытый файл" в ядре (см. главу про файлы). Таким образом, два процесса получают доступ к *одной и той же* структуре и, следовательно, имеют *общий указатель чтения/записи* для этого файла. Поэтому, когда процессы "отец" и "сын" пишут по дескриптору *fd*, они пользуются одним и тем же указателем R/W, т.е. информация от обоих процессов записывается последовательно. На принципе наследования и совместного использования открытых файлов основан также системный вызов **pipe**.

Порожденный процесс наследует также: реакции на сигналы (!!!), текущий каталог, управляющий терминал, номер владельца процесса и группу владельца, и.т.п.

При системном вызове **exec**() (который заменяет *программу*, выполняемую процессом, на программу из указанного файла) все открытые каналы также достаются в наследство новой программе (а не закрываются).

6.5.3. Процесс-копия это хорошо, но не совсем то, что нам хотелось бы. Нам хочется запустить программу, содержащуюся в выполняемом файле (например **a.out**). Для этого существует системный вызов **exec**, который имеет несколько разновидностей. Рассмотрим только две:

```
char *path;
char *argv[], *envp[], *arg0, ..., *argn;
execle(path, arg0, arg1, ..., argn, NULL, envp);
execve(path, argv, envp);
```

Системный вызов **exec** заменяет *программу*, выполняемую данным процессом, на программу, загружаемую из файла *path*. В данном случае *path* должно быть полным именем файла или именем файла от текущего каталога:

```
/usr/bin/vi   a.out  ../mybin/xkick
```

Файл должен иметь код доступа "выполнение". Первые два байта файла (в его заголовке), рассматриваемые как **short int**, содержат так называемое "магическое число" (**A_MAGIC**), свое для каждого типа машин (смотри include-файл <*a.out.h*>). Его помещает в начало выполняемого файла редактор связей **ld** при компоновке программы из объектных файлов. Это число должно быть правильным, иначе система откажется запускать программу из этого файла. Бывает несколько разных магических чисел, обозначающих разные способы организации программы в памяти. Например, есть вариант, в котором сегменты **text** и **data** склеены вместе (тогда **text** не разделяем между процессами и не защищен от модификации программой), а есть - где данные и текст находятся в раздельных адресных пространствах и запись в **text** запрещена (аппаратно).

Остальные аргументы вызова - *arg0*, ..., *argn* - это аргументы функции **main** новой программы. Во второй форме вызова аргументы не перечисляются явно, а заносятся в массив. Это позволяет формировать произвольный массив строк-аргументов во время работы программы:

```
char *argv[20];
argv[0]="ls"; argv[1]="-l"; argv[2]="-i"; argv[3]=NULL;
execv( "/bin/ls", argv);
        либо
execl( "/bin/ls", "ls","-l","-i", NULL):
```

В результате этого вызова текущая программа завершается (но не процесс!) и вместо нее запускается программа из заданного файла: сегменты **stack**, **data**, **text** старой программы уничтожаются; создаются *новые* сегменты **data** и **text**, загружаемые из файла *path*; отводится сегмент **stack** (первоначально - не очень большого размера); сегмент **user** сохраняется от старой программы (за исключением реакций на сигналы, отличных от **SIG_DFL** и **SIG_IGN** - они будут сброшены в **SIG_DFL**). Затем будет вызвана функция **main** новой программы с аргументами *argv*:

```
void main( argc, argv )
    int argc; char *argv[]; { ... }
```

Количество аргументов - *argc* - подсчитает сама система. Строка NULL не подсчитывается.

Процесс остается тем же самым - он имеет тот же паспорт (только адреса сегментов изменились); тот же номер (*pid*); все открытые прежней программой файлы остаются открытыми (с теми же дескрипторами); текущий каталог также наследуется от старой программы; сигналы, которые игнорировались ею, также будут игнорироваться (остальные сбрасываются в **SIG_DFL**). Зато "сущность" процесса подвергается перерождению - он выполняет теперь *иную* программу. Таким образом, системный вызов **exec** осуществляет вызов функции **main**, находящейся в *другой программе*, передавая ей свои аргументы в качестве входных.

Системный вызов **exec** может не удаться, если указанный файл *path* не существует, либо вы не имеете права его выполнять (такие коды доступа), либо он не является выполняемой программой (неверное магическое число), либо слишком велик для данной машины (системы), либо файл открыт каким-нибудь процессом (например еще записывается компилятором). В этом случае продолжится выполнение прежней программы. Если же вызов успешен - возврата из **exec** не происходит вообще (поскольку управление передается в другую программу).

Аргумент *argv*[0] обычно полагают равным *path*. По нему программа, имеющая несколько имен (в файловой системе), может выбрать ЧТО она должна делать. Так программа **/bin/ls** имеет альтернативные имена **lr**, **lf**, **lx**, **ll**. Запускается *одна и та же* программа, но в зависимости от *argv*[0] она далее делает разную работу.

Аргумент *envp* - это "окружение" программы (см. начало этой главы). Если он не задан - передается окружение текущей программы (наследуется содержимое массива, на который указывает переменная **environ**); если же задан явно (например, окружение скопировано в какой-то массив и часть переменных подправлена или добавлены новые переменные) - новая программа получит новое окружение. Напомним, что окружение можно прочесть из предопределенной переменной char ****environ**, либо из третьего аргумента функции **main** (см. начало главы), либо функцией **getenv**().

Системные вызовы **fork** и **exec** не склеены в один вызов потому, что между **fork** и **exec** в процессе-сыне могут происходить некоторые действия, нарушающие симметрию процесса-отца и порожденного процесса: установка реакций на сигналы, перенаправление ввода/вывода, и.т.п. Смотри пример "интерпретатор команд" в приложении. В **MS DOS**, не имеющей параллельных процессов, вызовы **fork**, **exec** и **wait** склеены в *один* вызов **spawn**. Зато при этом приходится делать перенаправления ввода-вывода в *порождающем* процессе перед **spawn**, а после него - восстанавливать все как было.

6.5.4. Завершить *процесс* можно системным вызовом

```
void exit( unsigned char retcode );
```

Из этого вызова не бывает возврата. Процесс завершается: сегменты **stack**, **data**, **text**, **user** уничтожаются (при этом все открытые процессом файлы закрываются); память, которую они занимали, считается свободной и в нее может быть помещен другой процесс. Причина смерти отмечается в паспорте процесса - в структуре **proc** в таблице процессов внутри ядра. Но паспорт еще не уничтожается! Это состояние процесса называется "зомби" - живой мертвец.

В паспорт процесса заносится код ответа *retcode*. Этот код может быть прочитан процессом-родителем (тем, кто создал этот процесс вызовом **fork**). Принято, что код 0 означает успешное завершение процесса, а любое положительное значение 1..255 означает неудачное завершение с таким кодом ошибки. Коды ошибок заранее не предопределены: это личное дело процессов отца и сына - установить между собой какие-то соглашения по этому поводу. В старых программах иногда писалось **exit**(-1); Это некорректно - код ответа должен быть неотрицателен; код -1 превращается в код 255. Часто используется конструкция **exit**(*errno*);

Программа может завершиться не только *явно* вызывая **exit**, но и еще двумя способами:

- если происходит возврат управления из функции **main**(), т.е. она кончилась - то вызов **exit**() делается неявно, но с непредсказуемым значением *retcode*;
- процесс может быть *убит* сигналом. В этом случае он не выдает никакого кода ответа в процесс-родитель, а выдает признак "процесс убит".

6.5.5. В действительности **exit**() - это еще не сам системный вызов завершения, а стандартная *функция*. Сам системный вызов называется **_exit**(). Мы можем переопределить функцию **exit**() так, чтобы по окончании программы происходили некоторые действия:

```
void exit(unsigned code){
  /* Добавленный мной дополнительный оператор: */
  printf("Закончить работу, "
         "код ответа=%u\n", code);

  /* Стандартные операторы: */
  _cleanup();  /* закрыть все открытые файлы.
                * Это стандартная функция £ */
  _exit(code); /* собственно сисвызов */
}

int f(){ return 17; }
void main(){
  printf("aaaa\n"); printf("bbbb\n"); f();
  /* потом откомментируйте это:  exit(77); */
}
```

Здесь функция **exit** вызывается *неявно* по окончании **main**, ее подставляет в программу компилятор. Дело в том, что при запуске программы **exec**-ом, первым начинает выполняться код так называемого "*стартера*", подклеенного при сборке программы из файла **/lib/crt0.o**. Он выглядит примерно так (в действительности он написан на ассемблере):

```
... // вычислить argc, настроить некоторые параметры.
main(argc, argv, envp);
exit();
```

£ **_cleanup**() закрывает файлы, открытые **fopen**()ом, "вытряхая" при этом данные, накопленные в буферах, в файл. При аварийном завершении программы файлы все равно закрываются, но уже не явно, а операционной системой (в вызове **_exit**). При этом содержимое недосброшенных буферов будет *утеряно*.

или так (взято из проекта **GNU**††):

```
int errno = 0;
char **environ;
_start(int argc, int arga)
{
/* OS and Compiler dependent!!!! */
char **argv = (char **) &arga;
char **envp = environ = argv + argc + 1;
/* ... возможно еще какие-то инициализации,
 * наподобие setlocale( LC_ALL, "" ); в SCO UNIX */
exit (main(argc, argv, envp));
}
```

Где должно быть

```
int main(int argc, char *argv[], char *envp[]){
          ...
        return 0;  /* вместо exit(0); */
}
```

Адрес функции **_start**() помечается в одном из полей заголовка файла формата **a.out** как адрес, на который система должна передать управление после загрузки программы в память (точка входа).

Какой код ответа попадет в **exit**() в этих примерах (если отсутствует *явный* вызов **exit** или **return**) - непредсказуемо. На **IBM PC** в вышенаписанном примере этот код равен 17, то есть значению, возвращенному последней вызывавшейся функцией. Однако это не какое-то специальное соглашение, а случайный эффект (так уж устроен код, создаваемый этим компилятором).

6.5.6. Процесс-отец может дождаться окончания своего потомка. Это делается системным вызовом **wait** и нужно по следующей причине: пусть отец - это интерпретатор команд. Если он запустил процесс и продолжил свою работу, то *оба* процесса будут предпринимать попытки читать ввод с клавиатуры терминала - интерпретатор ждет команд, а запущенная программа ждет данных. Кому из них будет поступать набираемый нами текст - непредсказуемо! Вывод: интерпретатор команд должен "заснуть" на то время, пока работает порожденный им процесс:

```
int pid;  unsigned short status;
          ...
if((pid = fork()) == 0 ){
        /* порожденный процесс */
         ...  // перенаправления ввода-вывода.
         ...  // настройка сигналов.
        exec(....);
        perror("exec не удался"); exit(1);
}
/* иначе это породивший процесс */
while((pid = wait(&status)) > 0 )
  printf("Окончился сын pid=%d с кодом %d\n",
          pid, status >> 8);
printf( "Больше нет сыновей\n");
```

wait приостанавливает† выполнение вызвавшего процесса до момента окончания *любого* из порожденных им процессов (ведь можно было запустить и нескольких сыновей!). Как только какой-то потомок окончится - **wait** проснется и выдаст номер (*pid*) этого потомка. Когда никого из живых "сыновей" не осталось - он выдаст (-1). Ясно, что процессы могут оканчиваться не в том порядке, в котором их порождали. В

---

†† **GNU** - программы, распространяемые в исходных текстах из **Free Software Foundation** (FSF). Среди них - **C++** компилятор **g++** и редактор **emacs**. Смысл слов **GNU** - "generally not **UNIX**" - проект был основан как противодействие начавшейся коммерциализации **UNIX** и закрытию его исходных текстов. "Сделать как в **UNIX**, но лучше".

† "Живой" процесс может пребывать в одном из нескольких состояний: процесс ожидает наступления какого-то события ("спит"), при этом ему не выделяется время процессора, т.к. он не готов к выполнению; процесс готов к выполнению и стоит в очереди к процессору (поскольку процессор выполняет другой процесс); процесс готов и выполняется процессором в данный момент. Последнее состояние может происходить в двух режимах - пользовательском (выполняются команды сегмента **text**) и системном (процессом был издан системный вызов, и сейчас выполняется функция в ядре). Ожидание события бывает только в системной фазе - внутри системного вызова (т.е. это "синхронное" ожидание). Неактивные процессы ("спящие" или ждущие ресурса процессора) могут быть временно откачаны на диск.

переменную *status* заносится в специальном виде код ответа окончившегося процесса, либо номер сигнала, которым он был убит.

```
#include <sys/types.h>
#include <sys/wait.h>
        ...
int status, pid;
        ...
while((pid = wait(&status)) > 0){
    if( WIFEXITED(status)){
      printf( "Процесс %d умер с кодом %d\n",
                      pid,            WEXITSTATUS(status));
    } else if( WIFSIGNALED(status)){
      printf( "Процесс %d убит сигналом %d\n",
                      pid,             WTERMSIG(status));
      if(WCOREDUMP(status)) printf( "Образовался core\n" );
      /* core - образ памяти процесса для отладчика adb */
    } else if( WIFSTOPPED(status)){
      printf( "Процесс %d остановлен сигналом %d\n",
                      pid,            WSTOPSIG(status));
    } else if( WIFCONTINUED(status)){
      printf( "Процесс %d продолжен\n",
                      pid);
    }
}
        ...
```

Если код ответа нас не интересует, мы можем писать **wait**(NULL).

Если у нашего процесса не было или больше нет живых сыновей - вызов **wait** ничего не ждет, а возвращает значение (-1). В написанном примере цикл **while** позволяет дождаться окончания *всех* потомков.

В тот момент, когда процесс-отец получает информацию о причине смерти потомка, паспорт умершего процесса наконец *вычеркивается* из таблицы процессов и может быть переиспользован новым процессом. До того, он хранится в таблице процессов в состоянии "zombie" - "живой мертвец". Только для того, чтобы кто-нибудь мог узать статус его завершения.

Если процесс-отец завершился *раньше* своих сыновей, то кто же сделает **wait** и вычеркнет паспорт? Это сделает процесс номер 1: **/etc/init**. Если отец умер раньше процессов-сыновей, то система заставляет процесс номер 1 "усыновить" эти процессы. **init** обычно находится в цикле, содержащем в начале вызов **wait**(), то есть ожидает окончания любого из своих сыновей (а они у него всегда есть, о чем мы поговорим подробнее чуть погодя). Таким образом **init** занимается чисткой таблицы процессов, хотя это не единственная его функция.

Вот схема, поясняющая жизненный цикл любого процесса:

```
    |pid=719,csh
    |
if(!fork())------->--------* pid=723,csh
    |                       |                  загрузить
 wait(&status)           exec("a.out",...) <-- a.out
    :                    main(...){           с диска
    :                      |
    :pid=719,csh           | pid=723,a.out
  спит(ждет)             работает
    :                      |
    :                    exit(status) умер
    :                    }
 проснулся <---проснись!--RIP
    |
    |pid=719,csh
```

Заметьте, что номер порожденного процесса не обязан быть *следующим* за номером родителя, а только *больше* него. Это связано с тем, что *другие* процессы могли создать в системе новые процессы *до* того, как наш процесс издал свой вызов **fork**.

6.5.7. Кроме того, **wait** позволяет отслеживать остановку процесса. Процесс может быть приостановлен при помощи посылки ему сигналов **SIGSTOP**, **SIGTTIN**, **SIGTTOU**, **SIGTSTP**. Последние три сигнала посылает при определенных обстоятельствах драйвер терминала, к примеру **SIGTSTP** - при нажатии клавиши **CTRL/Z**. Продолжается процесс посылкой ему сигнала **SIGCONT**.

В данном контексте, однако, нас интересуют не сами эти сигналы, а другая схема манипуляции с отслеживанием статуса порожденных процессов. Если указано *явно*, система может посылать процессу-родителю сигнал **SIGCLD** в момент изменения статуса любого из его потомков. Это позволит процессу-родителю *немедленно* сделать **wait** и немедленно отразить изменение состояние процесса-потомка в своих внутренних списках. Данная схема программируется так:

```
void pchild(){
        int pid, status;

        sighold(SIGCLD);
        while((pid = waitpid((pid_t) -1, &status, WNOHANG|WUNTRACED)) > 0){
          dorecord:
                записать_информацию_об_изменениях;
        }
        sigrelse(SIGCLD);

        /* Reset */
        signal(SIGCLD, pchild);
}
        ...
main(){
        ...
        /* По сигналу SIGCLD вызывать функцию pchild */
        signal(SIGCLD, pchild);
        ...
        главный_цикл;
}
```

Секция с вызовом **waitpid** (разновидность вызова **wait**), прикрыта парой функций **sighold**-**sigrelse**, запрещающих приход сигнала **SIGCLD** внутри этой критической секции. Сделано это вот для чего: если процесс начнет модифицировать таблицы или списки в районе метки dorecord:, а в этот момент придет еще один сигнал, то функция **pchild** будет вызвана рекурсивно и тоже попытается модифицировать таблицы и списки, в которых еще остались незавершенными перестановки ссылок, элементов, счетчиков. Это приведет к разрушению данных.

Поэтому сигналы должны приходить последовательно, и функции **pchild** вызываться также последовательно, а не рекурсивно. Функция **sighold** откладывает доставку сигнала (если он случится), а **sigrelse** - разрешает доставить накопившиеся сигналы (но если их пришло несколько одного типа - все они доставляются как *один* такой сигнал. Отсюда - цикл вокруг **waitpid**).

Флаг **WNOHANG** - означает "не ждать внутри вызова **wait**", если ни один из потомков не изменил своего состояния; а просто вернуть код (-1)". Это позволяет вызывать **pchild** даже без получения сигнала: ничего не произойдет. Флаг **WUNTRACED** - означает "выдавать информацию также об остановленных процессах".

6.5.8. Как уже было сказано, при **exec** все открытые файлы достаются в наследство новой программе (в частности, если между **fork** и **exec** были перенаправлены вызовом **dup2** стандартные ввод и вывод, то они останутся перенаправленными и у новой программы). Что делать, если мы не хотим, чтобы наследовались *все* открытые файлы? (Хотя бы потому, что большинством из них новая программа пользоваться не будет - в основном она будет использовать лишь *fd* 0, 1 и 2; а ячейки в таблице открытых файлов процесса они занимают). Во-первых, ненужные дескрипторы можно явно закрыть **close** в промежутке между **fork**-ом и **exec**-ом. Однако не всегда мы помним номера дескрипторов для этой операции. Более радикальной мерой является тотальная чистка:

```
for(f = 3; f < NOFILE; f++)
        close(f);
```

Есть более элегантный путь. Можно пометить дескриптор файла специальным флагом, означающим, что во время вызова **exec** этот дескриптор должен быть *автоматически* закрыт (режим file-close-on-exec - **fclex**):

```
#include <fcntl.h>
int fd = open(.....);
fcntl (fd, F_SETFD, 1);
```

Отменить этот режим можно так:

```
fcntl (fd, F_SETFD, 0);
```

Здесь есть одна тонкость: этот флаг устанавливается не для структуры **file** - "открытый файл", а непосредственно для *дескриптора* в таблице открытых процессом файлов (массив флагов: char *u_pofile*[NOFILE]). Он *не* сбрасывается при закрытии файла, поэтому нас может ожидать сюрприз:

```
... fcntl (fd, F_SETFD, 1); ... close(fd);
...
int fd1 = open( ... );
```

Если *fd1* окажется равным *fd*, то дескриптор *fd1* будет при **exec**-е закрыт, чего мы явно не ожидали! Поэтому перед **close**(*fd*) полезно было бы отменить режим **fclex**.

6.5.9. Каждый процесс имеет *управляющий терминал* (short **u_ttyp*). Он достается процессу в наследство от родителя (при **fork** и **exec**) и обычно совпадает с терминалом, с на котором работает данный пользователь.

Каждый процесс относится к некоторой *группе процессов* (int *p_pgrp*), которая также наследуется. Можно послать сигнал всем процессам указанной группы *pgrp*:

kill(-*pgrp*, *sig*);

Вызов

kill(0, *sig*);

посылает сигнал *sig* всем процессам, чья группа совпадает с группой посылающего процесса. Процесс может узнать свою группу:

int *pgrp* = **getpgrp**();

а может стать "лидером" новой группы. Вызов

setpgrp();

делает следующие операции:

```
/* У процесса больше нет управл. терминала: */
if(p_pgrp != p_pid) u_ttyp = NULL;
/* Группа процесса полагается равной его ид-у: */
p_pgrp = p_pid;  /* new group */
```

В свою очередь, управляющий терминал тоже имеет некоторую группу (*t_pgrp*). Это значение устанавливается равным группе процесса, первым открывшего этот терминал:

```
/* часть процедуры открытия терминала */
if( p_pid == p_pgrp // лидер группы
 && u_ttyp == NULL  // еще нет упр.терм.
 && t_pgrp == 0 ){  // у терминала нет группы
        u_ttyp = &t_pgrp;
        t_pgrp =  p_pgrp;
}
```

Таким процессом обычно является процесс регистрации пользователя в системе (который спрашивает у вас имя и пароль). При закрытии терминала всеми процессами (что бывает при выходе пользователя из системы) терминал теряет группу: *t_pgrp*=0;

При нажатии на клавиатуре терминала некоторых клавиш:

```
c_cc[ VINTR ]      обычно DEL или CTRL/C
c_cc[ VQUIT ]      обычно CTRL/\
```

драйвер терминала посылает соответственно сигналы **SIGINT** и **SIGQUIT** всем процессам группы терминала, т.е. как бы делает

kill(-*t_pgrp*, *sig*);

Именно поэтому мы можем прервать процесс нажатием клавиши **DEL**. Поэтому, если процесс сделал **setpgrp**(), то сигнал с клавиатуры ему послать невозможно (т.к. он имеет свой уникальный номер группы != группе терминала).

Если процесс еще не имеет управляющего терминала (или уже его не имеет после **setpgrp**), то он может сделать *любой* терминал (который он имеет право открыть) управляющим для себя. Первый же файл-устройство, являющийся интерфейсом драйвера терминалов, который будет открыт этим процессом, станет для него управляющим терминалом. Так процесс может иметь каналы 0, 1, 2 связанные с одним терминалом, а прерывания получать с клавиатуры другого (который он сделал управляющим для себя).

Процесс регистрации пользователя в системе - /**etc**/**getty** (название происходит от "*get tty*" - получить терминал) - запускается процессом номер 1 - /**etc**/**init**-ом - на каждом из терминалов, зарегистрированных в системе, когда

- система только что была запущена;
- либо когда пользователь на каком-то терминале вышел из системы (интерпретатор команд завершился).

В сильном упрощении **getty** может быть описан так:

```
void main(ac, av) char *av[];
{   int f; struct termio tmodes;

    for(f=0; f < NOFILE; f++) close(f);
```

```
        /* Отказ от управляющего терминала,
         * основание новой группы процессов.
         */
        setpgrp();

        /* Первоначальное явное открытие терминала */
        /* При этом терминал av[1] станет упр. терминалом */
            open( av[1], O_RDONLY ); /* fd = 0 */
            open( av[1], O_RDWR   ); /* fd = 1 */
        f = open( av[1], O_RDWR   ); /* fd = 2 */

        // ... Считывание параметров терминала из файла
        // /etc/gettydefs. Тип требуемых параметров линии
        // задается меткой, указываемой в av[2].
        // Заполнение структуры tmodes требуемыми
        // значениями ... и установка мод терминала.
        ioctl (f, TCSETA, &tmodes);

        // ... запрос имени и пароля ...

        chdir (домашний_каталог_пользователя);

        execl ("/bin/csh", "-csh", NULL);
        /* Запуск интерпретатора команд. Группа процессов,
         * управл. терминал, дескрипторы 0,1,2 наследуются.
         */
    }
```

Здесь последовательные вызовы **open** занимают последовательные ячейки в таблице открытых процессом файлов (поиск каждой новой незанятой ячейки производится с начала таблицы) - в итоге по дескрипторам 0,1,2 открывается файл-терминал. После этого дескрипторы 0,1,2 наследуются всеми потомками интерпретатора команд. Процесс **init** запускает по одному процессу **getty** на каждый терминал, как бы делая

```
        /etc/getty /dev/tty01 m &
        /etc/getty /dev/tty02 m &
                ...
```

и ожидает окончания любого из них. После входа пользователя в систему на каком-то терминале, соответствующий **getty** превращается в интерпретатор команд (*pid* процесса сохраняется). Как только кто-то из них умрет - **init** перезапустит **getty** на соответствующем терминале (все они - его сыновья, поэтому он знает - на каком именно терминале).

6.6. Трубы и FIFO-файлы.

Процессы могут обмениваться между собой информацией через файлы. Существуют файлы с необычным поведением - так называемые **FIFO**-файлы (*first in*, *first out*), ведущие себя подобно очереди. У них указатели чтения и записи *разделены*. Работа с таким файлом напоминает проталкивание шаров через трубу - с одного конца мы вталкиваем данные, с другого конца - вынимаем их. Операция чтения из *пустой* "трубы" проиостановит вызов **read** (и издавший его процесс) до тех пор, пока кто-нибудь не запишет в FIFO-файл какие-нибудь данные. Операция позиционирования указателя - **lseek**() - *неприменима* к FIFO-файлам. FIFO-файл создается системным вызовом

```
#include <sys/types.h>
#include <sys/stat.h>
   mknod( имяФайла, S_IFIFO | 0666, 0 );
```

где 0666 - коды доступа к файлу. При помощи FIFO-файла могут общаться даже неродственные процессы.

Разновидностью FIFO-файла является *безымянный* FIFO-файл, предназначенный для обмена информацией между процессом-отцом и процессом-сыном. Такой файл - канал связи как раз и называется термином "труба" или **pipe**. Он создается вызовом **pipe**:

```
int conn[2];   pipe(conn);
```

Если бы файл-труба имел имя *PIPEFILE*, то вызов **pipe** можно было бы описать как

```
mknod("PIPEFILE", S_IFIFO | 0600, 0);
conn[0] = open("PIPEFILE", O_RDONLY);
conn[1] = open("PIPEFILE", O_WRONLY);
unlink("PIPEFILE");
```

При вызове **fork** каждому из двух процессов достанется в наследство пара дескрипторов:

```
                pipe(conn);
                  fork();

conn[0]----<----     ----<-----conn[1]
                 FIFO
conn[1]---->----     ---->-----conn[0]
 процесс A                  процесс B
```

Пусть процесс **A** будет посылать информацию в процесс **B**. Тогда процесс **A** сделает:

```
close(conn[0]);
// т.к. не собирается ничего читать
write(conn[1], ... );
```

а процесс **B**

```
close(conn[1]);
// т.к. не собирается ничего писать
read (conn[0], ... );
```

Получаем в итоге:

```
conn[1]---->----FIFO---->-----conn[0]
 процесс A                  процесс B
```

Обычно поступают еще более элегантно, перенаправляя стандартный вывод **A** в канал *conn*[1]

```
dup2 (conn[1], 1); close(conn[1]);
write(1, ... );   /* или printf */
```

а стандартный ввод **B** - из канала *conn*[0]

```
dup2(conn[0], 0); close(conn[0]);
read(0, ... );    /* или gets */
```

Это соответствует конструкции

```
$   A | B
```

записанной на языке СиШелл.

Файл, выделяемый под **pipe**, имеет ограниченный размер (и поэтому обычно целиком оседает в буферах в памяти машины). Как только он заполнен целиком - процесс, пишущий в трубу вызовом **write**, приостанавливается до появления свободного места в трубе. Это может привести к возникновению тупиковой ситуации, если писать программу неаккуратно. Пусть процесс **A** является сыном процесса **B**, и пусть процесс **B** издает вызов **wait**, не закрыв канал *conn*[0]. Процесс же **A** очень много пишет в трубу *conn*[1]. Мы получаем ситуацию, когда оба процесса спят:

A потому что труба переполнена, а процесс **B** ничего из нее не читает, так как ждет окончания **A**;

B потому что процесс-сын **A** не окончился, а он не может окончиться пока не допишет свое сообщение.

Решением служит запрет процессу **B** делать вызов **wait** до тех пор, пока он не прочитает ВСЮ информацию из трубы (не получит EOF). Только сделав после этого **close**(*conn*[0]); процесс **B** имеет право сделать **wait**.

Если процесс **B** закроет свою сторону трубы **close**(*conn*[0]) *прежде*, чем процесс **A** закончит запись в нее, то при вызове **write** в процессе **A**, система пришлет процессу **A** сигнал **SIGPIPE** - "запись в канал, из которого никто не читает".

6.6.1. Открытие **FIFO** файла приведет к блокированию процесса ("засыпанию"), если в буфере **FIFO** файла пусто. Процесс заснет внутри вызова **open** до тех пор, пока в буфере что-нибудь не появится.

Чтобы избежать такой ситуации, а, например, сделать что-нибудь иное полезное в это время, нам надо было бы *опросить* файл на предмет того - можно ли его открыть? Это делается при помощи флага **O_NDELAY** у вызова **open**.

```
int fd = open(filename, O_RDONLY|O_NDELAY);
```

Если **open** ведет к блокировке процесса внутри вызова, вместо этого будет возвращено значение (-1). Если же файл может быть немедленно открыт - возвращается нормальный дескриптор со значением >=0, и файл открыт.

O_NDELAY является зависимым от семантики того файла, который мы открываем. К примеру, можно использовать его с файлами устройств, например именами, ведущими к последовательным портам. Эти файлы устройств (порты) обладают тем свойством, что одновременно их может открыть только один процесс (так устроена реализация функции **open** внутри драйвера этих устройств). Поэтому, если один процесс уже работает с портом, а в это время второй пытается его же открыть, второй "заснет" внутри **open**, и будет дожидаться освобождения порта **close** первым процессом. Чтобы не ждать - следует открывать порт

с флагом **O_NDELAY**.

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

/* Убрать больше не нужный O_NDELAY */
void nondelay(int fd){
        fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) & ~O_NDELAY);
}
int main(int ac, char *av[]){
        int fd;
        char *port = ac > 1 ? "/dev/term/a" : "/dev/cua/a";


retry:  if((fd = open(port, O_RDWR|O_NDELAY)) < 0){
                perror(port);
                sleep(10);
                goto retry;
        }
        printf("Порт %s открыт.\n", port);
        nondelay(fd);

        printf("Работа с портом, вызови эту программу еще раз!\n");
        sleep(60);
        printf("Все.\n");
        return 0;
}
```

Вот протокол:

```
su# a.out & a.out xxx
[1] 22202
Порт /dev/term/a открыт.
Работа с портом, вызови эту программу еще раз!
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
Все.
Порт /dev/cua/a открыт.
Работа с портом, вызови эту программу еще раз!
su#
```

6.7. Нелокальный переход.

Теперь поговорим про *нелокальный переход*. Стандартная функция **setjmp** позволяет установить в программе "контрольную точку"†, а функция **longjmp** осуществляет прыжок в эту точку, выполняя за один раз выход *сразу из нескольких* вызванных функций (если надо)£. Эти функции не являются системными вызовами, но поскольку они реализуются машинно-зависимым образом, а используются чаще всего как реакция на некоторый сигнал, речь о них идет в этом разделе. Вот как, например, выглядит рестарт программы по прерыванию с клавиатуры:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf jmp;  /* контрольная точка */

/* прыгнуть в контрольную точку */
void onintr(nsig){ longjmp(jmp, nsig); }
```

---

† В некотором буфере запоминается текущее состояние процесса: положение вершины стека вызовов функций (*stack pointer*); состояние всех регистров процессора, включая регистр адреса текущей машинной команды (*instruction pointer*).

£ Это достигается восстановлением состояния процесса из буфера. Изменения, происшедшие за время между **setjmp** и **longjmp** в статических данных не отменяются (т.к. они не сохранялись).

```
main(){
   int n;
   n = setjmp(jmp);  /* установить контрольную точку */
   if( n ) printf( "Рестарт после сигнала %d\n", n);
   signal (SIGINT, onintr);     /* реакция на сигнал */
   printf("Начали\n");
   ...
}
```

setjmp возвращает 0 при *запоминании* контрольной точки. При прыжке в контрольную точку при помощи **longjmp**, мы оказываемся снова в функции **setjmp**, и эта функция возвращает нам значение второго аргумента **longjmp**, в этом примере - *nsig*.

Прыжок в контрольную точку очень удобно использовать в алгоритмах перебора с возвратом (*backtracking*): либо - если ответ найден - прыжок на печать ответа, либо - если ветвь перебора зашла в тупик - прыжок в точку ветвления и выбор другой альтернативы. При этом можно делать прыжки и в рекурсивных вызовах одной и той же функции: с более высокого уровня рекурсии в вызов более низкого уровня (в этом случае **jmp_buf** лучше делать автоматической переменной - своей для каждого уровня вызова функции).

6.7.1. Перепишите следующий алгоритм при помощи **longjmp**.

```
#define FOUND    1 /* ответ найден    */
#define NOTFOUND 0 /* ответ не найден */
int value;         /* результат */
main(){    int i;
  for(i=2; i < 10; i++){
      printf( "пробуем i=%d\n", i);
      if( test1(i) == FOUND ){
          printf("ответ %d\n", value); break;
      }
  }
}
test1(i){  int j;
  for(j=1; j < 10 ; j++ ){
      printf( "пробуем j=%d\n", j);
      if( test2(i,j) == FOUND ) return FOUND;
      /* "сквозной" return */
  }
  return NOTFOUND;
}
test2(i, j){
  printf( "пробуем(%d,%d)\n", i, j);
  if( i * j == 21 ){
      printf( "  Годятся (%d,%d)\n", i,j);
      value = j; return FOUND;
  }
  return NOTFOUND;
}
```

Вот ответ, использующий нелокальный переход вместо цепочки **return**-ов:

```
#include <setjmp.h>
jmp_buf jmp;
main(){   int i;
  if( i = setjmp(jmp))  /* после прыжка */
        printf("Ответ %d\n", --i);
  else  /* установка точки */
    for(i=2; i < 10; i++)
      printf( "пробуем i=%d\n", i), test1(i);
}
test1(i){ int j;
  for(j=1; j < 10 ; j++ )
      printf( "пробуем j=%d\n", j), test2(i,j);
}
test2(i, j){
  printf( "пробуем(%d,%d)\n", i, j);
  if( i * j == 21 ){
     printf( "  Годятся (%d,%d)\n", i,j);
```

```
        longjmp(jmp, j + 1);
    }
}
```

Обратите внимание, что при возврате ответа через второй аргумент **longjmp** мы прибавили 1, а при печати ответа мы эту единицу отняли. Это сделано на случай ответа *j*==0, чтобы функция **setjmp** не вернула бы в этом случае значение 0 (признак *установки* контрольной точки).

6.7.2. В чем ошибка?

```
#include <setjmp.h>
jmp_buf jmp;
main(){
     g();
     longjmp(jmp,1);
}
g(){ printf("Вызвана g\n");
     f();
     printf("Выхожу из g\n");
}
f(){
     static n;
     printf( "Вызвана f\n");
     setjmp(jmp);
     printf( "Выхожу из f %d-ый раз\n", ++n);
}
```

Ответ: **longjmp** делает прыжок в функцию **f**(), из которой уже произошел возврат управления. При переходе в тело функции в обход ее заголовка не выполняются машинные команды "пролога" функции - функция остается "неактивированной". При возврате из вызванной таким "нелегальным" путем функции возникает ошибка, и программа падает. Мораль: в функцию, которая НИКЕМ НЕ ВЫЗВАНА, нельзя передавать управление. Обратный прыжок - из **f**() в **main**() - был бы законен, поскольку функция **main**() является активной, когда управление находится в теле функции **f**(). Т.е. можно "прыгать" из вызванной функции в вызывающую: из **f**() в **main**() или в **g**(); и из **g**() в **main**();

```
--         --
 |   f     |  стек      прыгать
 |   g     |  вызовов   сверху вниз
 |   main  |  функций   можно - это соответствует
 ----------             выкидыванию нескольких
                        верхних слоев стека
```

но нельзя наоборот: из **main**() в **g**() или **f**(); а также из **g**() в **f**(). Можно также совершать прыжок в пределах одной и той же функции:

```
f(){ ...
        A:   setjmp(jmp);
             ...
             longjmp(jmp, ...); ...
             /* это как бы goto A; */
}
```

6.8. Хозяин файла, процесса, и проверка привелегий.

UNIX - многопользовательская система. Это значит, что одновременно на разных терминалах, подключенных к машине, могут работать *разные* пользователи (а может и один на нескольких терминалах). На каждом терминале работает *свой* интерпретатор команд, являющийся потомком процесса **/etc/init**.

6.8.1. Теперь - про функции, позволяющие узнать некоторые данные про любого пользователя системы. Каждый пользователь в **UNIX** имеет уникальный *номер*: идентификатор пользователя (*user id*), а также уникальное *имя*: регистрационное имя, которое он набирает для входа в систему. Вся информация о пользователях хранится в файле **/etc/passwd**. Существуют функции, позволяющие по номеру пользователя узнать регистрационное имя и наоборот, а заодно получить еще некоторую информацию из **passwd**:

```
#include <stdio.h>
#include <pwd.h>
struct passwd *p;
int   uid;   /* номер */
char *uname; /* рег. имя */

uid = getuid();
p   = getpwuid( uid   );
        ...
p   = getpwnam( uname );
```

Эти функции возвращают указатели на статические структуры, скрытые внутри этих функций. Структуры эти имеют поля:

```
p->pw_uid     идентиф. пользователя (int uid);
p->pw_gid     идентиф. группы пользователя;

        и ряд полей типа char[]
p->pw_name    регистрационное имя пользователя (uname);
p->pw_dir     полное имя домашнего каталога
  (каталога, становящегося текущим при входе в систему);
p->pw_shell   интерпретатор команд
  (если "", то имеется в виду /bin/sh);
p->pw_comment произвольная учетная информация (не используется);
p->pw_gecos   произвольная учетная информация (обычно ФИО);
p->pw_passwd  зашифрованный пароль для входа в
   систему. Истинный пароль нигде не хранится вовсе!
```

Функции возвращают значение *p*==**NULL**, если указанный пользователь не существует (например, если задан неверный *uid*). *uid* хозяина данного процесса можно узнать вызовом **getuid**, а *uid* владельца файла - из поля *st_uid* структуры, заполняемой системным вызовом **stat** (а идентификатор группы владельца - из поля *st_gid*). Задание: модифицируйте наш аналог программы **ls**, чтобы он выдавал в текстовом виде имя владельца каждого файла в каталоге.

6.8.2. Владелец файла может изменить своему файлу идентификаторы владельца и группы вызовом

```
chown(char *имяФайла, int uid, int gid);
```

т.е. "подарить" файл другому пользователю. Забрать чужой файл себе невозможно. При этой операции биты **S_ISUID** и **S_ISGID** в кодах доступа к файлу (см. ниже) сбрасываются, поэтому создать "Троянского коня" и, сделав его хозяином суперпользователя, получить неограниченные привелегии - не удастся!

6.8.3. Каждый файл имеет своего владельца (поле *di_uid* в I-узле на диске или поле *i_uid* в копии I-узла в памяти ядра†). Каждый процесс также имеет своего владельца (поля *u_uid* и *u_ruid* в **u-area**). Как мы видим, процесс имеет *два* параметра, обозначающие владельца. Поле *ruid* называется "**реальным идентификатором**" пользователя, а *uid* - "**эффективным идентификатором**". При вызове **exec**() заменяется программа, выполняемая данным процессом:

```
старая программа  exec    новая программа
    ruid -->-----------------><---> ruid
    uid  -->--------*--------><---> uid (new)
                    |
              выполняемый файл
               i_uid (st_uid)
```

Как видно из этой схемы, реальный идентификатор хозяина процесса наследуется. Эффективный идентификатор обычно также наследуется, за исключением одного случая: если в кодах доступа файла (*i_mode*) выставлен бит **S_ISUID** (set-uid bit), то значение поля *u_uid* в новом процессе станет равно значению *i_uid* файла с программой:

---

† При открытии файла и вообще при любой операции с файлом, в таблицах ядра заводится *копия* I-узла (для ускорения доступа, чтобы постоянно не обращаться к диску). Если I-узел в памяти будет изменен, то при закрытии файла (а также периодически через некоторые промежутки времени) эта копия будет записана обратно на диск. Структура I-узла в памяти - struct **inode** - описана в файле <*sys/inode.h*>, а на диске - struct **dinode** - в файле <*sys/ino.h*>.

```
/* ... во время exec ... */
p_suid = u_uid;     /* спасти */
if( i_mode & S_ISUID ) u_uid = i_uid;
if( i_mode & S_ISGID ) u_gid = i_gid;
```

т.е. эффективным владельцем процесса станет владелец файла. Здесь *gid* - это идентификаторы *группы владельца* (которые тоже есть и у файла и у процесса, причем у процесса - реальный и эффективный).

Зачем все это надо? Во-первых затем, что ПРАВА процесса на доступ к какому-либо файлу проверяются именно для *эффективного* владельца процесса. Т.е. например, если файл имеет коды доступа

```
mode = i_mode & 0777;
             /* rwx rwx rwx */
```

и владельца *i_uid*, то процесс, пытающийся открыть этот файл, будет "проэкзаменован" в таком порядке:

```
if( u_uid == 0 )  /* super user */
     то доступ разрешен;
else if( u_uid == i_uid )
     проверить коды (mode & 0700);
else if( u_gid == i_gid )
     проверить коды (mode & 0070);
else проверить коды (mode & 0007);
```

Процесс может узнать свои параметры:

```
unsigned short uid  = geteuid();  /* u_uid  */
unsigned short ruid = getuid();   /* u_ruid */
unsigned short gid  = getegid();  /* u_gid  */
unsigned short rgid = getuid();   /* u_rgid */
```

а также установить их:

```
setuid(newuid);  setgid(newgid);
```

Рассмотрим вызов **setuid**. Он работает так (*u_uid* - относится к процессу, издавшему этот вызов):

```
if(      u_uid == 0 /* superuser */ )
         u_uid = u_ruid =    p_suid =  newuid;
else if( u_ruid == newuid || p_suid == newuid )
         u_uid = newuid;
else     неудача;
```

Поле *p_suid* позволяет set-uid-ной программе восстановить эффективного владельца, который был у нее до **exec**-а.

Во-вторых, все это надо для следующего случая: пусть у меня есть некоторый файл *BASE* с хранящимися в нем секретными сведениями. Я являюсь владельцем этого файла и устанавливаю ему коды доступа 0600 (чтение и запись разрешены *только* мне). Тем не менее, я хочу дать другим пользователям возможность работать с этим файлом, однако контролируя их деятельность. Для этого я пишу *программу*, которая выполняет некоторые действия с файлом *BASE*, при этом проверяя законность этих действий, т.е. позволяя делать не все что попало, а лишь то, что я в ней предусмотрел, и под жестким контролем. Владельцем файла *PROG*, в котором хранится эта программа, также являюсь я, и я задаю этому файлу коды доступа 0711 (rwx--x--x) - всем можно выполнять эту программу. Все ли я сделал, чтобы позволить другим пользоваться базой *BASE* через программу (и только нее) *PROG*? Нет!

Если кто-то другой запустит программу *PROG*, то эффективный идентификатор процесса будет равен идентификатору этого *другого* пользователя, и программа *не сможет* открыть мой файл *BASE*. Чтобы все работало, процесс, выполняющий программу *PROG*, должен работать как бы от моего имени. Для этого я должен вызовом **chmod** либо командой

chmod u+s *PROG*

добавить к кодам доступа файла *PROG* бит **S_ISUID**.

После этого, при запуске программы *PROG*, она будет получать эффективный идентификатор, равный *моему* идентификатору, и таким образом сможет открыть и работать с файлом *BASE*. Вызов **getuid** позволяет выяснить, кто вызвал мою программу (и занести это в протокол, если надо).

Программы такого типа - не редкость в **UNIX**, если владельцем программы (файла ее содержащего) является суперпользователь. В таком случае программа, имеющая бит доступа **S_ISUID** работает *от имени суперпользователя* и может выполнять некоторые действия, запрещенные обычным пользователям. При этом программа внутри себя делает всяческие проверки и периодически спрашивает пароли, то есть при работе защищает систему от дураков и преднамеренных вредителей. Простейшим примером служит команда **ps**, которая считывает таблицу процессов из памяти ядра и распечатывает ее. Доступ к физической памяти машины производится через файл-псевдоустройство /**dev**/**mem**, а к памяти ядра - /**dev**/**kmem**. Чтение и запись в них позволены *только* суперпользователю, поэтому программы "общего пользования",

обращающиеся к этим файлам, должны иметь бит set-uid.

Откуда же изначально берутся значения *uid* и *ruid* (а также *gid* и *rgid*) у процесса? Они берутся из процесса регистрации пользователя в системе: **/etc/getty**. Этот процесс запускается на каждом терминале как процесс, принадлежащий суперпользователю (*u_uid*==0). Сначала он запрашивает имя и пароль пользователя:

```
#include <stdio.h>  /* cc -lc_s */
#include <pwd.h>
#include <signal.h>
struct passwd *p;
char userName[80], *pass, *crpass;
extern char *getpass(), *crypt();
  ...
/* Не прерываться по сигналам с клавиатуры */
signal (SIGINT, SIG_IGN);
for(;;){
  /* Запросить имя пользователя: */
  printf("Login: "); gets(userName);
  /* Запросить пароль (без эха): */
  pass = getpass("Password: ");
  /* Проверить имя: */
  if(p = getpwnam(userName)){
     /* есть такой пользователь */
     crpass = (p->pw_passwd[0]) ? /* если есть пароль */
               crypt(pass, p->pw_passwd) : pass;
     if( !strcmp( crpass, p->pw_passwd))
               break; /* верный пароль */
  }
  printf("Login incorrect.\a\n");
}
signal (SIGINT, SIG_DFL);
```

Затем он выполняет:

```
// ... запись информации о входе пользователя в систему
// в файлы /etc/utmp (кто работает в системе сейчас)
// и       /etc/wtmp (список всех входов в систему)
        ...
setuid( p->pw_uid ); setgid( p->pw_gid );
chdir ( p->pw_dir ); /* GO HOME! */
// эти параметры будут унаследованы
// интерпретатором команд.
        ...
// настройка некоторых переменных окружения envp:
// HOME     = p->pw_dir
// SHELL    = p->pw_shell
// PATH     = нечто по умолчанию, вроде :/bin:/usr/bin
// LOGNAME (USER) = p->pw_name
// TERM     = считывается из файла
//            /etc/ttytype по имени устройства av[1]
// Делается это как-то подобно
//   char *envp[MAXENV], buffer[512]; int envc = 0;
//   ...
//   sprintf(buffer, "HOME=%s", p->pw_dir);
//   envp[envc++] = strdup(buffer);
//   ...
//   envp[envc] = NULL;
        ...
// настройка кодов доступа к терминалу. Имя устройства
// содержится в параметре av[1] функции main.
chown (av[1], p->pw_uid, p->pw_gid);
chmod (av[1], 0600 );  /* -rw------- */
// теперь доступ к данному терминалу имеют только
// вошедший в систему пользователь и суперпользователь.
// В случае смерти интерпретатора команд,
// которым заменится getty, процесс init сойдет
// с системного вызова ожидания wait() и выполнит
```

```
//  chown ( этот_терминал, 2 /*bin*/, 15 /*terminal*/ );
//  chmod ( этот_терминал, 0600 );
// и, если терминал числится в файле описания линий
// связи /etc/inittab как активный (метка respawn), то
// init перезапустит на этом_терминале новый
// процесс getty при помощи пары вызовов fork() и exec().
        ...
// запуск интерпретатора команд:
execle( *p->pw_shell ? p->pw_shell : "/bin/sh",
                    "-", NULL, envp );
```

В результате он становится процессом пользователя, вошедшего в систему. Таковым же после **exec**-а, выполняемого **getty**, остается и интерпретатор команд *p->pw_shell* (обычно /**bin**/**sh** или /**bin**/**csh**) и все его потомки.

На самом деле, в описании регистрации пользователя при входе в систему, сознательно было допущено упрощение. Дело в том, что все то, что мы приписали процессу **getty**, в действительности выполняется *двумя* программами: /**etc**/**getty** и /**bin**/**login**.

Сначала процесс **getty** занимается настройкой параметров линии связи (т.е. терминала) в соответствии с ее описанием в файле /**etc**/**gettydefs**. Затем он запрашивает *имя* пользователя и заменяет себя (при помощи сисвызова **exec**) процессом **login**, передавая ему в качестве одного из аргументов полученное имя пользователя.

Затем **login** запрашивает пароль, настраивает окружение, и.т.п., то есть именно он производит все операции, приведенные выше на схеме. В конце концов он заменяет себя интерпретатором команд.

Такое разделение делается, в частности, для того, чтобы считанный пароль в случае опечатки не хранился бы в памяти процесса **getty**, а уничтожался бы при очистке памяти завершившегося процесса **login**. Таким образом пароль в истинном, незашифрованном виде хранится в системе минимальное время, что затрудняет его подсматривание средствами электронного или программного шпионажа. Кроме того, это позволяет изменять систему проверки паролей не изменяя программу инициализации терминала **getty**.

Имя, под которым пользователь вошел в систему на данном терминале, можно узнать вызовом стандартной функции

char ***getlogin**();

Эта функция не проверяет *uid* процесса, а просто извлекает запись про данный терминал из файла /**etc**/**utmp**.

Наконец отметим, что владелец *файла* устанавливается при создании этого файла (вызовами **creat** или **mknod**), и полагается равным эффективному идентификатору создающего процесса.

```
di_uid = u_uid;      di_gid = u_gid;
```

6.8.4. Напишите программу, узнающую у системы и распечатывающую: номер процесса, номер и имя своего владельца, номер группы, название и тип терминала на котором она работает (из переменной окружения **TERM**).

6.9. Блокировка доступа к файлам.

В базах данных нередко встречается ситуация одновременного доступа к одним и тем же данным. Допустим, что в некотором файле хранятся данные, которые могут читаться и записываться произвольным числом процессов.

- Допустим, что процесс A изменяет некоторую область файла, в то время как процесс B пытается прочесть ту же область. Итогом такого соревнования может быть то, что процесс B прочтет неверные данные.
- Допустим, что процесс A изменяет некоторую область файла, в то время как процесс C также изменяет ту же самую область. В итоге эта область может содержать неверные данные (часть - от процесса A, часть - от C).

Ясно, что требуется механизм синхронизации процессов, позволяющий не пускать другой процесс (процессы) читать и/или записывать данные в указанной области. Механизмов синхронизации в **UNIX** существует множество: от семафоров до блокировок областей файла. О последних мы и будем тут говорить.

Прежде всего отметим, что блокировки файла носят в **UNIX** *необязательный* характер. То есть, программа не использующая вызовов синхронизации, будет иметь доступ к данным без каких либо ограничений. Увы. Таким образом, программы, собирающиеся корректно пользоваться общими данными, должны *все* использовать - и при том один и тот же - механизм синхронизации: заключить между собой "джентльменское соглашение".

6.9.1. Блокировка устанавливается при помощи вызова

```
flock_t lock;

fcntl(fd, operation, &lock);
```

Здесь *operation* может быть одним из трех:

F_SETLK

Устанавливает или снимает замок, описываемый структурой *lock*. Структура *flock_t* имеет такие поля:

```
short  l_type;
short  l_whence;
off_t  l_start;
size_t l_len;

long   l_sysid;
pid_t  l_pid;
```

l_type тип блокировки:

```
F_RDLCK - на чтение;
F_WRLCK - на запись;
F_UNLCK - снять все замки.
```

l_whence, *l_start*, *l_len*

описывают сегмент файла, на который ставится замок: от точки **lseek**(fd,*l_start*,*l_whence*); длиной *l_len* байт. Здесь *l_whence* может быть: **SEEK_SET**, **SEEK_CUR**, **SEEK_END**. *l_len* равное нулю означает "до конца файла". Так если все три параметра равны 0, то будет заблокирован весь файл.

F_SETLKW

Устанавливает или снимает замок, описываемый структурой *lock*. При этом, если замок на область, пересекающуюся с указанной уже кем-то установлен, то сперва дождаться снятия этого замка.

Пытаемся поставить замок на	Нет чужих замков	Уже есть замок на READ	уже есть замок на WRITE
READ	читать	читать	ждать;запереть;читать
WRITE	записать	ждать;запереть;записать	ждать;запереть;записать
UNLOCK	отпереть	отпереть	отпереть

- Если кто-то читает сегмент файла, то другие тоже могут его читать свободно, ибо чтение не изменяет файла.
- Если же кто-то записывает файл - то все остальные должны дождаться окончания записи и разблокировки.
- Если кто-то читает сегмент, а другой процесс собрался изменить (записать) этот сегмент, то этот другой процесс обязан дождаться окончания чтения первым.
- В момент, обозначенный как *отпереть* - будятся процессы, ждущие разблокировки, и ровно один из них получает доступ (может установить свою блокировку). Порядок - кто из них будет первым - вообще говоря не определен.

F_GETLK

Запрашиваем возможность установить замок, описанный в *lock*.

- Если мы можем установить такой замок (не заперто никем), то в структуре *lock* поле *l_type* становится равным **F_UNLCK** и поле *l_whence* равным **SEEK_SET**.
- Если замок уже кем-то установлен (и вызов **F_SETLKW** заблокировал бы наш процесс, привел бы к ожиданию), мы получаем информацию о чужом замке в структуру *lock*. При этом в поле *l_pid* заносится идентификатор процесса, создавшего этот замок, а в поле *l_sysid* - идентификатор машины (поскольку блокировка файлов поддерживается через сетевые файловые системы).

Замки автоматически снимаются при закрытии дескриптора файла. Замки **не** наследуются порожденным процессом при вызове **fork**.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>

char DataFile [] = "data.xxx";
char info     [] = "abcdefghijklmnopqrstuvwxyz";
#define OFFSET 5
#define SIZE   12

#define PAUSE 2

int trial = 1;
int fd, pid;
char buffer[120], myname[20];
void writeAccess(), readAccess();

void fcleanup(int nsig){
        unlink(DataFile);
        printf("cleanup:%s\n", myname);
        if(nsig) exit(0);
}

int main(){
        int i;

        fd = creat(DataFile, 0644);
        write(fd, info, strlen(info));
        close(fd);

        signal(SIGINT, fcleanup);

        sprintf(myname, fork() ? "B-%06d" : "A-%06d", pid = getpid());

        srand(time(NULL)+pid);
        printf("%s:started\n", myname);

        fd = open(DataFile, O_RDWR|O_EXCL);
        printf("%s:opened %s\n", myname, DataFile);

        for(i=0; i < 30; i++){
                if(rand()%2)    readAccess();
                else            writeAccess();
        }

        close(fd);

        printf("%s:finished\n", myname);

        wait(NULL);
        fcleanup(0);
        return 0;
}
```

```
void writeAccess(){
        flock_t lock;

        printf("Write:%s #%d\n", myname, trial);

        lock.l_type   = F_WRLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start  = (off_t)  OFFSET;
        lock.l_len    = (size_t) SIZE;

        if(fcntl(fd, F_SETLKW, &lock) <0)
                perror("F_SETLKW");
        printf("\twrite:%s locked\n", myname);

        sprintf(buffer, "%s #%02d", myname, trial);
        printf ("\twrite:%s \"%s\"\n", myname, buffer);

        lseek (fd, (off_t) OFFSET, SEEK_SET);
        write (fd, buffer, SIZE);

        sleep (PAUSE);

        lock.l_type   = F_UNLCK;
        if(fcntl(fd, F_SETLKW, &lock) <0)
                perror("F_SETLKW");

        printf("\twrite:%s unlocked\n", myname);

        trial++;
}

void readAccess(){
        flock_t lock;

        printf("Read:%s #%d\n", myname, trial);

        lock.l_type   = F_RDLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start  = (off_t)  OFFSET;
        lock.l_len    = (size_t) SIZE;

        if(fcntl(fd, F_SETLKW, &lock) <0)
                perror("F_SETLKW");
        printf("\tread:%s locked\n", myname);

        lseek(fd, (off_t) OFFSET, SEEK_SET);
        read (fd, buffer, SIZE);

        printf("\tcontents:%s \"%*.*s\"\n", myname, SIZE, SIZE, buffer);
        sleep (PAUSE);

        lock.l_type   = F_UNLCK;
        if(fcntl(fd, F_SETLKW, &lock) <0)
                perror("F_SETLKW");

        printf("\tread:%s unlocked\n", myname);

        trial++;
}
```

Исследуя выдачу этой программы, вы можете обнаружить, что READ-области могут перекрываться; но что никогда не перекрываются области READ и WRITE ни в какой комбинации. Если идет чтение процессом A - то запись процессом B дождется разблокировки A (чтение - не будет дожидаться). Если идет запись процессом A - то и чтение процессом B и запись процессом B дождутся разблокировки A.

6.9.2.

UNIX SVR4 имеет еще один интерфейс для блокировки файлов: функцию **lockf**.

```
#include <unistd.h>

int lockf(int fd, int operation, size_t size);
```

Операция *operation*:

F_ULOCK
Разблокировать указанный сегмент файла (это может снимать один или несколько замков).

F_LOCK

F_TLOCK
Установить замок. При этом, если уже имеется чужой замок на запрашиваемую область, **F_LOCK** блокирует процесс, **F_TLOCK** - просто выдает ошибку (функция возвращает -1, **errno** устанавливается в **EAGAIN**).

- Ожидание отпирания/запирания замка может быть прервано сигналом.
- Замок устанавливается следующим образом: от текущей позиции указателя чтения-записи в файле *fd* (что не похоже на **fcntl**, где позиция задается явно как параметр в структуре); длиной *size*. Отрицательное значение *size* означает отсчет от текущей позиции к началу файла. Нулевое значение - означает "от текущей позиции до конца файла". При этом "конец файла" понимается именно как конец, а не как текущий размер файла. Если файл изменит размер, запертая область все равно будет простираться до конца файла (уже нового).
- Замки, установленные процессом, автоматически отпираются при завершении процесса.

F_TEST
Проверить наличие замка. Функция возвращает 0, если замка нет; -1 в противном случае (заперто).

Если устанавливается замок, перекрывающийся с уже установленным, то замки объединяются.

```
было:      ____________#######_____######____________

запрошено:_______________##########________________

стало:     ____________##################____________
```

Если снимается замок с области, покрывающей только часть заблокированной прежде, остаток области остается как отдельный замок.

```
было:      ____________##################____________

запрошено:_______________XXXXXXXXXX________________

стало:     ____________###___________####____________
```

6.10. Файлы устройств.

Пространство дисковой памяти может состоять из нескольких *файловых систем* (в дальнейшем FS), т.е. логических и/или физических дисков. Каждая файловая система имеет древовидную логическую структуру (каталоги, подкаталоги и файлы) и имеет *свой* корневой каталог. Файлы в каждой FS имеют свои собственные I-узлы и собственную их нумерацию с 1. В начале каждой FS зарезервированы:

- блок для загрузчика - программы, вызываемой аппаратно при включении машины (загрузчик записывает с диска в память машины программу /**boot**, которая в свою очередь загружает в память ядро /**unix**);
- *суперблок* - блок заголовка файловой системы, хранящий размер файловой системы (в блоках), размер блока (512, 1024, ...), количество I-узлов, начало списка свободных блоков, и другие сведения об FS;
- некоторая непрерывная область диска для хранения I-узлов - "I-файл".

Файловые системы объединяются в единую древовидную иерархию операцией *монтирования* - подключения корня файловой системы к какому-то из каталогов-"листьев" дерева другой FS.

Файлы в объединенной иерархии адресуются при помощи двух способов:

- имен, задающих путь в дереве каталогов:

```
/usr/abs/bin/hackIt
bin/hackIt
./../../bin/vi
```

(этот способ предназначен для *программ*, пользующихся файлами, а также пользователей);

- внутренних адресов, используемых программами ядра и некоторыми системными программами.

Поскольку в каждой FS имеется *собственная* нумерация I-узлов, то файл в объединенной иерархии должен адресоваться ДВУМЯ параметрами:

- номером (кодом) устройства, содержащего файловую систему, в которой находится искомый файл: **dev_t** *i_dev*;
- номером I-узла файла в этой файловой системе: **ino_t** *i_number*;

Преобразование *имени файла* в объединенной файловой иерархии в такую *адресную пару* выполняет в ядре уже упоминавшаяся выше функция **namei** (при помощи просмотра каталогов):

```
struct inode *ip = namei(...);
```

Создаваемая ею копия I-узла в памяти ядра содержит поля *i_dev* и *i_number* (которые на самом диске не хранятся!).

Рассмотрим некоторые алгоритмы работы ядра с файлами. Ниже они приведены чисто *схематично* и в сильном упрощении. Форматы вызова (и оформление) функций не соответствуют форматам, используемым на самом деле в ядре; верны лишь *названия* функций. Опущены проверки на корректность, подсчет ссылок на структуры **file** и **inode**, блокировка I-узлов и кэш-буферов от одновременного доступа, и многое другое.

Пусть мы хотим открыть файл для чтения и прочитать из него некоторую информацию. Вызовы открытия и закрытия файла имеют схему (часть ее будет объяснена позже):

```
#include <sys/types.h>
#include <sys/inode.h>
#include <sys/file.h>
int fd_read = open(имяФайла, O_RDONLY){

  int fd; struct inode *ip; struct file *fp; dev_t dev;

  u_error = 0;    /* errno в программе */
// Найти файл по имени. Создается копия I-узла в памяти:
  ip = namei(имяФайла, LOOKUP);
// namei может выдать ошибку, если нет такого файла
  if(u_error) return(-1);  // ошибка

// Выделяется структура "открытый файл":
  fp = falloc(ip, FREAD);
  // fp->f_flag = FREAD; открыт на чтение
  // fp->f_offset = 0;   RWptr
  // fp->f_inode  = ip;  ссылка на I-узел

// Выделить новый дескриптор
  for(fd=0; fd < NOFILE; fd++)
     if(u_ofile[fd] == NULL ) // свободен
         goto done;
  u_error = EMFILE; return (-1);
done:
  u_ofile[fd] = fp;

// Если это устройство - инициализировать его.
// Это функция openi(ip, fp->f_flag);
  dev = ip->i_rdev;
  if((ip->i_mode & IFMT) == IFCHR)
    (*cdevsw[major(dev)].d_open)(minor(dev),fp->f_flag);
  else if((ip->i_mode & IFMT) == IFBLK)
    (*bdevsw[major(dev)].d_open)(minor(dev),fp->f_flag);
  return fd;  // через u_rval1
}

close(fd){
  struct file  *fp = u_ofile[fd];
  struct inode *ip = fp->f_inode;
  dev_t dev = ip->i_rdev;

  if((ip->i_mode & IFMT) == IFCHR)
```

```
    (*cdevsw[major(dev)].d_close)(minor(dev),fp->f_flag);
  else if((ip->i_mode & IFMT) == IFBLK)
    (*bdevsw[major(dev)].d_close)(minor(dev),fp->f_flag);

  u_ofile[fd] = NULL;
  // и удалить ненужные структуры из ядра.
}
```

Теперь рассмотрим функцию преобразования логических блоков файла в номера физических блоков в файловой системе. Для этого преобразования в I-узле файла содержится таблица адресов блоков. Она устроена довольно сложно - ее начало находится в узле, а продолжение - в нескольких блоках в самой файловой системе (устройство это можно увидеть в примере "Фрагментированность файловой системы" в приложении). Мы для простоты будем предполагать, что это просто линейный массив *i_addr*[], в котором *n*-ому логическому блоку файла отвечает *bno*-тый физический блок файловой системы:

```
bno = ip->i_addr[n];
```

Если файл является интерфейсом *устройства*, то этот файл не хранит информации в логической файловой системе. Поэтому у устройств нет таблицы адресов блоков. Вместо этого, поле *i_addr*[0] используется для хранения *кода устройства*, к которому приводит этот специальный файл. Это поле носит название *i_rdev*, т.е. как бы сделано

```
#define i_rdev i_addr[0]
```

(на самом деле используется union). Устройства бывают *байто-ориентированные*, обмен с которыми производится по одному байту (как с терминалом или с коммуникационным портом); и *блочно-ориентированные*, обмен с которыми возможен только большими порциями - блоками (пример - диск). То, что файл является устройством, помечено в поле *тип файла*

```
ip->i_mode & IFMT
```

одним из значений: **IFCHR** - байтовое; или **IFBLK** - блочное. Алгоритм вычисления номера блока:

```
ushort u_pboff;  // смещение от начала блока
ushort u_pbsize; // сколько байт надо использовать
// ushort  - это unsigned short, смотри <sys/types.h>
// daddr_t - это long (disk address)

daddr_t bmap(struct inode *ip,
             off_t offset, unsigned count){
  int sz, rem;

  // вычислить логический номер блока по позиции RWptr.
  // BSIZE - это размер блока файловой системы,
  // эта константа определена в <sys/param.h>
  daddr_t bno = offset / BSIZE;
  // если BSIZE == 1 Кб, то можно offset >> 10

      u_pboff = offset % BSIZE;
      // это можно записать как offset & 01777

      sz = BSIZE - u_pboff;
      // столько байт надо взять из этого блока,
      // начиная с позиции u_pboff.

      if(count < sz) sz = count;
      u_pbsize = sz;
```

Если файл представляет собой устройство, то трансляция логических блоков в физические не производится - устройство представляет собой "сырой" диск без файлов и каталогов, т.е. обращение происходит сразу по физическому номеру блока:

```
      if((ip->i_mode & IFMT) == IFBLK) // block device
         return bno;       // raw disk
      // иначе провести пересчет:

      rem = ip->i_size /*длина файла*/ - offset;
      // это остаток файла.
      if( rem < 0 )  rem = 0;
      // файл короче, чем заказано нами:
      if( rem < sz ) sz = rem;
      if((u_pbsize = sz) == 0) return (-1); // EOF
```

```
        // и, собственно, замена логич. номера на физич.
        return ip->i_addr[bno];
}
```

Теперь рассмотрим алгоритм **read**. Параметры, начинающиеся с *u_...*, на самом деле передаются как статические через вспомогательные переменные в **u-area** процесса.

```
read(int fd, char *u_base, unsigned u_count){
    unsigned srccount = u_count;
    struct   file  *fp = u_ofile[fd];
    struct   inode *ip = fp->f_inode;
    struct   buf   *bp;
    daddr_t         bno; // очередной блок файла

    // dev - устройство,
    // интерфейсом которого является файл-устройство,
    // или на котором расположен обычный файл.
    dev_t dev = (ip->i_mode & (IFCHR|IFBLK)) ?
          ip->i_rdev : ip->i_dev;

    switch( ip->i_mode & IFMT ){

    case IFCHR:  // байто-ориентированное устройство
      (*cdevsw[major(dev)].d_read)(minor(dev));
      // прочие параметры передаются через u-area
      break;

    case IFREG:  // обычный файл
    case IFDIR:  // каталог
    case IFBLK:  // блочно-ориентированное устройство
      do{
         bno = bmap(ip, fp->f_offset /*RWptr*/, u_count);
         if(u_pbsize==0 || (long)bno < 0) break; // EOF
         bp  = bread(dev, bno);  // block read

         iomove(bp->b_addr + u_pboff, u_pbsize, B_READ);
```

Функция **iomove** копирует данные

```
bp->b_addr[ u_pboff..u_pboff+u_pbsize-1 ]
```

из адресного пространства ядра (из буфера в ядре) в адресное пространство процесса по адресам

```
u_base[ 0..u_pbsize-1 ]
```

то есть пересылает *u_pbsize* байт между ядром и процессом (*u_base* попадает в **iomove** через статическую переменную). При записи вызовом **write**(), **iomove** с флагом **B_WRITE** производит обратное копирование - из памяти процесса в память ядра. Продолжим:

```
         // продвинуть счетчики и указатели:
         u_count      -= u_pbsize;
         u_base       += u_pbsize;
         fp->f_offset += u_pbsize;  // RWptr
      } while( u_count != 0 );
      break;
    ...
    return( srccount - u_count );
} // end read
```

Теперь обсудим некоторые места этого алгоритма. Сначала посмотрим, как происходит обращение к байтовому устройству. Вместо адресов блоков мы получаем код устройства *i_rdev*. Коды устройств в **UNIX** (тип **dev_t**) представляют собой пару двух чисел, называемых *мажор* и *минор*, хранимых в старшем и младшем байтах кода устройства:

```
#define major(dev)  ((dev >> 8) & 0x7F)
#define minor(dev)  ( dev       & 0xFF)
```

Мажор обозначает *тип устройства* (диск, терминал, и.т.п.) и приводит к одному из драйверов (если у нас есть 8 терминалов, то их обслуживает один и тот же драйвер); а *минор* обозначает *номер устройства* данного типа (... каждый из терминалов имеет миноры 0..7). Миноры обычно служат индексами в некоторой таблице структур внутри выбранного драйвера. Мажор же служит индексом в переключательной таблице устройств. При этом блочно-ориентированные устройства выбираются в одной таблице - **bdevsw**[], а

байто-ориентированные - в другой - **cdevsw**[] (см. <*sys/conf.h*>; имена таблиц означают *block/character device switch*). Каждая строка таблицы содержит адреса функций, выполняющих некоторые предопределенные операции способом, зависимым от устройства. Сами эти функции реализованы в драйверах устройств. Аргументом для этих функций обычно служит *минор* устройства, к которому производится обращение. Функция в драйвере использует этот минор как *индекс* для выбора конкретного экземпляра устройства данного типа; как индекс в массиве управляющих структур (содержащих текущее состояние, режимы работы, адреса функций прерываний, адреса очередей данных и.т.п. каждого конкретного устройства) для данного типа устройств. Эти управляющие структуры *различны* для разных типов устройств (и их драйверов).

Каждая строка переключательной таблицы содержит адреса функций, выполняющих операции **open**, **close**, **read**, **write**, **ioctl**, **select**. **open** служит для инициализации устройства при первом его открытии (++*ip*->*i_count*==1) - например, для включения мотора; **close** - для выключения при последнем закрытии (--*ip*->*i_count*==0). У блочных устройств поля для **read** и **write** объединены в функцию **strategy**, вызываемую с параметром **B_READ** или **B_WRITE**. Вызов **ioctl** предназначен для управления параметрами работы устройства. Операция **select** - для опроса: есть ли поступившие в устройство данные (например, есть ли в clist-е ввода с клавиатуры байты? см. главу "Экранные библиотеки"). Вызов **select** применим только к некоторым байтоориентированным устройствам и сетевым портам (**socket**-ам). Если данное устройство не умеет выполнять такую операцию, то есть запрос к этой операции должен вернуть в программу ошибку (например, операция **read** неприменима к принтеру), то в переключательной таблице содержится специальное имя функции **nodev**; если же операция допустима, но является фиктивной (как **write** для **/dev/null**) - имя **nulldev**. Обе эти функции-заглушки представляют собой "пустышки": {}.

Теперь обратимся к блочно-ориентированным устройствам. **UNIX** использует внутри ядра дополнительную *буферизацию* при обменах с такими устройствами†. Использованная нами выше функция *bp*=**bread**(*dev*,*bno*); производит чтение физического блока номер *bno* с устройства *dev*. Эта операция обращается к драйверу конкретного устройства и вызывает чтение блока в некоторую область памяти в ядре ОС: в один из *кэш-буферов* (cache, "запасать"). Заголовки кэш-буферов (struct **buf**) организованы в список и имеют поля (см. файл <*sys/buf.h*>):

b_dev код устройства, с которого прочитан блок;

b_blkno
: номер физического блока, хранящегося в буфере в данный момент;

b_flags
: флаги блока (см. ниже);

b_addr
: адрес участка памяти (как правило в самом ядре), в котором собственно и хранится содержимое блока.

Буферизация блоков позволяет системе экономить число обращений к диску. При обращении к **bread**() сначала происходит поиск блока (*dev*,*bno*) в таблице кэш-буферов. Если блок уже был ранее прочитан в кэш, то обращения к диску не происходит, поскольку копия содержимого дискового блока уже есть в памяти ядра. Если же блока еще нет в кэш-буферах, то в ядре выделяется чистый буфер, в заголовке ему прописываются нужные значения полей *b_dev* и *b_blkno*, и блок считывается в буфер с диска вызовом функции

```
bp->b_flags |= B_READ;  // род работы: прочитать
(*bdevsw[major(dev)].d_startegy)(bp);
// bno и минор - берутся из полей *bp
```

из драйвера конкретного устройства.

Когда мы что-то изменяем в файле вызовом **write**(), то изменения на самом деле происходят в кэш-буферах в памяти ядра, а не сразу на диске. При записи в блок буфер помечается как *измененный*:

```
b_flags |= B_DELWRI;  // отложенная запись
```

и на диск немедленно не записывается. Измененные буфера физически записываются на диск в таких случаях:

- Был сделан системный вызов **sync**();
- Ядру не хватает кэш-буферов (их число ограничено). Тогда самый старый буфер (к которому дольше всего не было обращений) записывается на диск и после этого используется для другого блока.
- Файловая система была отмонтирована вызовом **umount**;

---

† Следует отличать эту системную буферизацию от буферизации при помощи библиотеки **stdio**. Библиотека создает буфер в самом *процессе*, тогда как системные вызовы имеют буфера внутри *ядра*.

Понятно, что *не измененные* блоки обратно на диск из буферов не записываются (т.к. на диске и так содержатся те же самые данные). Даже если файл уже закрыт **close**, его блоки могут быть еще не записаны на диск - запись произойдет лишь при вызове **sync**. Это означает, что измененные блоки записываются на диск "массированно" - по многу блоков, но не очень часто, что позволяет оптимизировать и саму запись на диск: сортировкой блоков можно достичь минимизации перемещения магнитных головок над диском.

Отслеживание самых "старых" буферов происходит за счет реорганизации списка заголовков кэш-буферов. В большом упрощении это можно представить так: как только к блоку происходит обращение, соответствующий заголовок переставляется в начало списка. В итоге самый "пассивный" блок оказывается в хвосте - он то и переиспользуется при нужде.

"Подвисание" файлов в памяти ядра значительно ускоряет работу программ, т.к. работа с памятью гораздо быстрее, чем с диском. Если блок надо считать/записать, а он уже есть в кэше, то реального обращения к диску не происходит. Зато, если случится сбой питания (или кто-то неаккуратно выключит машину), а некоторые буфера еще не были сброшены на диск - то часть изменений в файлах будет потеряна. Для принудительной записи всех измененных кэш-буферов на диск существует сисвызов "синхронизации" содержимого дисков и памяти

```
sync();  // synchronize
```

Вызов **sync** делается раз в 30 секунд специальным служебным процессом **/etc/update**, запускаемым при загрузке системы. Для работы с файлами, которые должны гарантированно быть корректными на диске, используется открытие файла

```
fd = open( имя, O_RDWR | O_SYNC);
```

которое означает, что при каждом **write** блок из кэш-буфера *немедленно* записывается на диск. Это делает работу надежнее, но существенно медленнее.

Специальные файлы устройств не могут быть созданы вызовом **creat**, создающим только обычные файлы. Файлы устройств создаются вызовом **mknod**:

```
#include <sys/sysmacros.h>
dev_t dev = makedev(major, minor);
                /* (major << 8) | minor */
mknod( имяФайла, кодыДоступа|тип, dev);
```

где *dev* - пара (мажор,минор) создаваемого устройства; *кодыДоступа* - коды доступа к файлу (0777)£; *тип* - это одна из констант **S_IFIFO**, **S_IFCHR**, **S_IFBLK** из include-файла <*sys/stat.h*>.

mknod доступен для выполнения только суперпользователю (за исключением случая **S_IFIFO**). Если бы это было не так, то можно было бы создать файл устройства, связанный с существующим диском, и читать информацию с него напрямую, в обход механизмов логической файловой системы и защиты файлов кодами доступа.

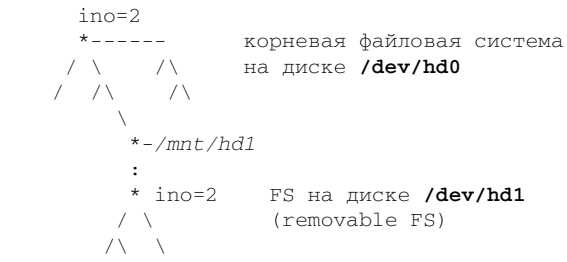
Можно создать файл устройства с мажором и/или минором, не отвечающим никакому реальному устройству (нет такого драйвера или минор слишком велик). Открытие таких устройств выдает код ошибки **ENODEV**.

Из нашей программы мы можем вызовом **stat**() узнать код устройства, на котором расположен файл. Он будет содержаться в поле **dev_t** *st_dev*; а если файл является специальным файлом (интерфейсом драйвера устройства), то код самого этого устройства можно узнать из поля **dev_t** *st_rdev*; Рассмотрим пример, который выясняет, относятся ли два имени к одному и тому же файлу:

```
#include <sys/types.h>
#include <sys/stat.h>
void main(ac, av) char *av[]; {
  struct stat st1, st2; int eq;
  if(ac != 3) exit(13);
  stat(av[1], &st1); stat(av[2], &st2);
  if(eq =
    (st1.st_ino == st2.st_ino && /* номера I-узлов */
     st1.st_dev == st2.st_dev))  /* коды устройств */
printf("%s и %s - два имени одного файла\n",av[1],av[2]);
  exit( !eq );
}
```

Наконец, вернемся к склейке нескольких файловых систем в одну объединенную иерархию:

£ Обычно к блочным устройствам (дискам) доступ разрешается только суперпользователю, в противном случае можно прочитать с "сырого" диска (в обход механизмов файловой системы) физические блоки любого файла и весь механизм защиты окажется неработающим.



Для того, чтобы поместить корневой каталог файловой системы, находящейся на диске */dev/hd1*, вместо каталога */mnt/hd1* уже "собранной" файловой системы, мы должны издать сисвызов

```
mount("/dev/hd1", "/mnt/hd1", 0);
```

Для отключения смонтированной файловой системы мы должны вызвать

```
umount("/dev/hd1");
```

(каталог, к которому она смонтирована, уже числится в таблице ядра, поэтому его задавать не надо). При монтировании все содержимое каталога */mnt/hd1* станет недоступным, зато при обращении к имени */mnt/hd1* мы на самом деле доберемся до (безымянного) корневого каталога на диске */dev/hd1*. Такой каталог носит название *mount point* и может быть выявлен по тому признаку, что "." и ".." в нем лежат на разных устройствах:

```
struct stat st1, st2;
stat("/mnt/hd1/.", &st1); stat("/mnt/hd1/..", &st2);
if( st1.st_dev != st2.st_dev) ... ; /*mount point*/
```

Для *st1* поле *st_dev* означает код устройства */dev/hd1*, а для *st2* - устройства, содержащего корневую файловую систему. Операции монтирования и отмонтирования файловых систем доступны только суперпользователю.

И напоследок - сравнение структур I-узла.

на диске	в памяти	в вызове **stat**
<sys/ino.h>	*<sys/inode.h>*	*<sys/stat.h>*
struct **dinode**	struct **inode**	struct **stat**
// коды доступа и тип файла		
ushort *di_mode*	*i_mode*	*st_mode*
// число имен файла		
short *di_nlink*	*i_nlink*	*st_nlink*
// номер I-узла		
ushort ---	*i_number*	*st_ino*
// идентификатор владельца		
ushort *di_uid*	*i_uid*	*st_uid*
// идентификатор группы владельца		
ushort *di_gid*	*i_gid*	*st_gid*
// размер файла в байтах		
off_t *di_size*	*i_size*	*st_size*
// время создания		
time_t *di_ctime*	*i_ctime*	*st_ctime*
// время последнего изменения (**write**)		
time_t *di_mtime*	*i_mtime*	*st_mtime*
// время последнего доступа (**read**/**write**)		
time_t *di_atime*	*i_atime*	*st_atime*
// устройство, на котором расположен файл		
dev_t ---	*i_dev*	*st_dev*
// устройство, к которому приводит спец.файл		
dev_t ---	*i_rdev*	*st_rdev*
// адреса блоков		
char *di_addr*[39]	*i_addr*[]	
// счетчик ссылок на структуру в ядре		
cnt_t	*i_count*	
//	и кое-что еще	

Минусы означают, что данное поле не хранится на диске, а вычисляется ядром. В современных версиях **UNIX** могут быть легкие отличия от вышенаписанной таблицы.

6.10.1. Напишите программу **pwd**, определяющую полное имя текущего рабочего каталога. **#define U42** определяет файловую систему с длинными именами, отсутствие этого флага - с короткими (14 символов).

```
/* Команда pwd.
 * Текст getwd() взят из исходных текстов библиотеки языка Си.
 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#define  ediag(e,r)      (e)
/*
 * getwd() возвращает полное имя текущего рабочего каталога.
 * При ошибке возвращается NULL, а в pathname копируется сообщение
 * об ошибке.
 */
#ifndef MAXPATHLEN
#define MAXPATHLEN      128
#endif

#define CURDIR          "."   /* имя текущего каталога      */
#define PARENTDIR       ".."  /* имя родительского каталога */
#define PATHSEP         "/"   /* разделитель компонент пути */
#define ROOTDIR         "/"   /* корневой каталог           */
#define GETWDERR(s)     strcpy(pathname, (s));
#define CP(to,from)     strncpy(to,from.d_name,DIRSIZ),to[DIRSIZ]='\0'

char *strcpy(char *, char *); char *strncpy(char *, char *, int);
char *getwd(char *pathname);
static char *prepend(char *dirname, char *pathname);

static int pathsize;                    /* длина имени */

#ifndef U42
char *getwd(char *pathname)
{
        char pathbuf[MAXPATHLEN];       /* temporary pathname buffer    */
        char *pnptr = &pathbuf[(sizeof pathbuf)-1]; /* pathname pointer */
        dev_t rdev;                     /* root device number           */
        int fil = (-1);                 /* directory file descriptor    */
        ino_t rino;                     /* root inode number            */
        struct direct dir;              /* directory entry struct       */
        struct stat d ,dd;              /* file status struct           */
                                        /* d - "."    dd - ".." | dname */
        char dname[DIRSIZ+1];           /* an directory entry           */

        pathsize = 0;
        *pnptr = '\0';
        if (stat(ROOTDIR, &d) < 0) {
                GETWDERR(ediag("getwd: can't stat /",
                               "getwd: нельзя выполнить stat /"));
                return (NULL);
        }
        rdev = d.st_dev; /* код устройства, на котором размещен корень */
        rino = d.st_ino; /* номер I-узла, представляющего корневой каталог */
```

```
for (;;) {
        if (stat(CURDIR, &d) < 0) {
CantStat:
                GETWDERR(ediag("getwd: can't stat .",
                                "getwd: нельзя выполнить stat ."));
                goto fail;
        }
        if (d.st_ino == rino && d.st_dev == rdev)
                break;          /* достигли корневого каталога */
        if ((fil = open(PARENTDIR, O_RDONLY)) < 0) {
                GETWDERR(ediag("getwd: can't open ..",
                                "getwd: нельзя открыть .."));
                goto fail;
        }
        if (chdir(PARENTDIR) < 0) {
                GETWDERR(ediag("getwd: can't chdir to ..",
                                "getwd: нельзя перейти в .."));
                goto fail;
        }
        if (fstat(fil, &dd) < 0)
                goto CantStat;
        if (d.st_dev == dd.st_dev) {  /* то же устройство */
                if (d.st_ino == dd.st_ino) {
                        /* достигли корня ".." == "." */
                        close(fil); break;
                }
                do {
                        if (read(fil, (char *) &dir,
                            sizeof(dir)) < sizeof(dir)
                        ){
                ReadErr:
                          close(fil);
                          GETWDERR(ediag("getwd: read error in ..",
                                         "getwd: ошибка чтения .."));
                                goto fail;
                        }
                } while (dir.d_ino != d.st_ino);
                CP(dname,dir);

        } else  /* ".." находится на другом диске: mount point */

                do {
                        if (read(fil, (char *) &dir,
                            sizeof(dir)) < sizeof(dir))
                                goto ReadErr;
                        if( dir.d_ino == 0 )    /* файл стерт */
                                continue;
                        CP(dname,dir);
                        if (stat(dname, &dd) < 0) {
                                sprintf (pathname, "getwd: %s %s",
                                         ediag ("can't stat",
                                    "нельзя выполнить stat"), dname);
                                goto fail;
                        }
                } while(dd.st_ino != d.st_ino ||
                        dd.st_dev != d.st_dev);
        close(fil);
        pnptr = prepend(PATHSEP, prepend(dname, pnptr));
}
```

```
        if (*pnptr == '\0')             /* текущий каталог == корневому */
                strcpy(pathname, ROOTDIR);
        else {
                strcpy(pathname, pnptr);
                if (chdir(pnptr) < 0) {
                        GETWDERR(ediag("getwd: can't change back to .",
                                       "getwd: нельзя вернуться в ."));
                        return (NULL);
                }
        }
        return (pathname);

fail:
        close(fil);
        chdir(prepend(CURDIR, pnptr));
        return (NULL);
}

#else /* U42 */
extern char   *strcpy ();
extern DIR    *opendir();

char   *getwd (char *pathname)
{
    char    pathbuf[MAXPATHLEN];/* temporary pathname buffer */
    char   *pnptr = &pathbuf[(sizeof pathbuf) - 1];/* pathname pointer */
    char   *prepend ();         /* prepend dirname to pathname */
    dev_t rdev;                 /* root device number */
    DIR * dirp;                 /* directory stream */
    ino_t rino;                 /* root inode number */
    struct dirent  *dir;        /* directory entry struct */
    struct stat d,
                dd;             /* file status struct */

    pathsize = 0;
    *pnptr = '\0';
    stat (ROOTDIR, &d);
    rdev = d.st_dev;
    rino = d.st_ino;

    for (;;) {
        stat (CURDIR, &d);

        if (d.st_ino == rino && d.st_dev == rdev)
            break;              /* reached root directory */

        if ((dirp = opendir (PARENTDIR)) == NULL) {
            GETWDERR ("getwd: can't open ..");
            goto fail;
        }
        if (chdir (PARENTDIR) < 0) {
            closedir (dirp);
            GETWDERR ("getwd: can't chdir to ..");
            goto fail;
        }
```

```
            fstat (dirp -> dd_fd, &dd);
            if (d.st_dev == dd.st_dev) {
                if (d.st_ino == dd.st_ino) {
                /* reached root directory */
                    closedir (dirp);
                    break;
                }
                do {
                    if ((dir = readdir (dirp)) == NULL) {
                        closedir (dirp);
                        GETWDERR ("getwd: read error in ..");
                        goto fail;
                    }
                } while (dir -> d_ino != d.st_ino);
            }

            else
                do {
                    if ((dir = readdir (dirp)) == NULL) {
                        closedir (dirp);
                        GETWDERR ("getwd: read error in ..");
                        goto fail;
                    }
                    stat (dir -> d_name, &dd);
                } while (dd.st_ino != d.st_ino || dd.st_dev != d.st_dev);
            closedir (dirp);
            pnptr = prepend (PATHSEP, prepend (dir -> d_name, pnptr));
        }

        if (*pnptr == '\0')         /* current dir == root dir */
            strcpy (pathname, ROOTDIR);
        else {
            strcpy (pathname, pnptr);
            if (chdir (pnptr) < 0) {
                GETWDERR ("getwd: can't change back to .");
                return (NULL);
            }
        }
        return (pathname);

    fail:
        chdir (prepend (CURDIR, pnptr));
        return (NULL);
    }
    #endif
```

```
/*
 * prepend() tacks a directory name onto the front of a pathname.
 */
static char *prepend (
        register char *dirname,         /* что добавлять    */
        register char *pathname         /* к чему добавлять */
) {
        register int i;         /* длина имени каталога */

        for (i = 0; *dirname != '\0'; i++, dirname++)
                continue;
        if ((pathsize += i) < MAXPATHLEN)
                while (i-- > 0)
                        *--pathname = *--dirname;
        return (pathname);
}

#ifndef CWDONLY
void main(){
        char buffer[MAXPATHLEN+1];
        char *cwd = getwd(buffer);
        printf( "%s%s\n", cwd ? "": "ERROR:", buffer);
}
#endif
```

6.10.2. Напишите функцию **canon**(), канонизирующую имя файла, т.е. превращающую его в *полное* имя (от корневого каталога), не содержащее компонент "." и "..", а также лишних символов слэш '/'. Пусть, к примеру, текущий рабочий каталог есть */usr/abs/C-book*. Тогда функция преобразует

```
  .                   -> /usr/abs/C-book
  ..                  -> /usr/abs
  ../..               -> /usr
  ////..              -> /
  /aa                 -> /aa
  /aa/../bb           -> /bb
  cc//dd/../ee        -> /usr/abs/C-book/cc/ee
  ../a/b/./d          -> /usr/abs/a/b/d
```

Ответ:

```
#include <stdio.h>
/* слэш, разделитель компонент пути */
#define SLASH        '/'
extern char *strchr (char *, char),
            *strrchr(char *, char);
struct savech{ char *s, c; };
#define SAVE(sv, str) (sv).s = (str); (sv).c = *(str)
#define RESTORE(sv) if((sv).s)   *(sv).s = (sv).c
/* Это структура для использования в таком контексте:
void main(){
  char *d = "hello"; struct savech ss;
  SAVE(ss, d+3); *(d+3) = '\0'; printf("%s\n", d);
  RESTORE(ss);                  printf("%s\n", d);
}
*/

/* ОТСЕЧЬ ПОСЛЕДНЮЮ КОМПОНЕНТУ ПУТИ */
struct savech parentdir(char *path){
   char *last  = strrchr( path, SLASH );
   char *first = strchr ( path, SLASH );
   struct savech sp; sp.s = NULL; sp.c = '\0';

if( last    == NULL  ) return sp; /* не полное имя    */
if( last[1] == '\0'  ) return sp; /* корневой каталог */
if( last    == first ) /* единственный слэш: /DIR     */
    last++;
   sp.s = last; sp.c = *last; *last = '\0';
```

```
   return sp;
}
#define isfullpath(s)   (*s == SLASH)
/* КАНОНИЗИРОВАТЬ ИМЯ ФАЙЛА */
void canon(
     char *where, /* куда поместить ответ */
     char *cwd,   /* полное имя текущего каталога */
     char *path   /* исходное имя для канонизации */
){   char *s, *slash;
   /* Сформировать имя каталога - точки отсчета */
   if( isfullpath(path)){
       s = strchr(path, SLASH);   /* @ */
       strncpy(where, path, s - path + 1);
       where[s - path + 1] = '\0';
       /* или даже просто strcpy(where, "/"); */
       path = s+1; /* остаток пути без '/' в начале */
   } else strcpy(where, cwd);

   /* Покомпонентный просмотр пути */
   do{  if(slash = strchr(path, SLASH)) *slash = '\0';
   /* теперь path содержит очередную компоненту пути */
        if(*path == '\0' || !strcmp(path, ".")) ;
     /* то просто проигнорировать "." и лишние "///" */
        else  if( !strcmp(path, ".."))
              (void) parentdir(where);
        else{ int len = strlen(where);
           /* добавить в конец разделяющий слэш */
              if( where[len-1] != SLASH ){
                  where[len] = SLASH;
                  where[len+1] = '\0';
              }
              strcat( where+len, path );
              /* +len чисто для ускорения поиска
               * конца строки внутри strcat(); */
        }
        if(slash){ *slash = SLASH; /* восстановить */
                   path   = slash + 1;
        }
   } while (slash != NULL);
}
char cwd[256], input[256], output[256];
void main(){
/* Узнать полное имя текущего каталога.
 * getcwd() - стандартная функция, вызывающая
 * через popen() команду pwd (и потому медленная).
 */
   getcwd(cwd, sizeof cwd);
   while( gets(input)){
     canon(output, cwd, input);
     printf("%-20s -> %s\n", input, output);
   }
}
```

В этом примере (изначально писавшемся для MS DOS) есть "странное" место, помеченное /*@*/. Дело в том, что в DOS функция **isfullpath** была способна распознавать имена файлов вроде *C:\aaa\bbb*, которые не обязательно *начинаются* со слэша.

6.11. Мультиплексирование ввода-вывода.

Данная глава посвящена системному вызову **select**, который, однако, мы предоставляем вам исследовать самостоятельно. Его роль такова: он позволяет опрашивать *несколько* дескрипторов открытых файлов (или устройств) и как только в файле появляется новая информация - сообщать об этом нашей программе. Обычно это бывает связано с дескрипторами, ведущими к сетевым устройствам.

6.11.1.

```
/* Пример использования вызова select() для мультиплексирования
 * нескольких каналов ввода. Этот вызов можно также использовать
 * для получения таймаута.
 *     Вызов: войти на терминалах tty01 tty02 и набрать на каждом
 *            sleep 30000
 *     затем  на tty00 сказать           select /dev/tty01 /dev/tty02
 *     и вводить что-либо на терминалах  tty01 и tty02
 * Сборка:      cc select.c -o select -lsocket
 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>  /* fd_set, FD_SET, e.t.c. */
#include <sys/param.h>  /* NOFILE */
#include <sys/select.h>
#include <sys/time.h>
#include <sys/filio.h>  /* для FIONREAD */
#define max(a,b)        ((a) > (b) ? (a) : (b))

char buf[512];          /* буфер чтения                         */
int fdin, fdout;        /* дескрипторы каналов stdin, stdout */
int nready;             /* число готовых каналов                */
int nopen;              /* число открытых каналов               */
int maxfd = 0;          /* максимальный дескриптор              */
int nfds;               /* сколько первых дескрипторов проверять */
int f;                  /* текущий дескриптор                   */
fd_set set, rset;       /* маски                                */

/* таблица открытых нами файлов */
struct _fds {
        int fd;         /* дескриптор */
        char name[30];  /* имя файла  */
} fds[ NOFILE ] = { /* NOFILE - макс. число открытых файлов на процесс */
        { 0, "stdin" }, { 1, "stdout" }, { 2, "stderr" }
        /* все остальное - нули */
};
struct timeval timeout, rtimeout;

/* выдать имя файла по дескриптору */
char *N( int fd ){
        register i;
        for(i=0; i < NOFILE; i++)
                if(fds[i].fd == fd ) return fds[i].name;
        return "???";
}
```

```
void main( int ac, char **av ){
        nopen = 3;              /* stdin, stdout, stderr */
        for( f = 3; f < NOFILE; f++ ) fds[f].fd = (-1);
        fdin = fileno(stdin);   fdout = fileno(stdout);
        setbuf(stdout, NULL);   /* отмена буферизации */
        FD_ZERO(&set);          /* очистка маски */

        for(f=1; f < ac; f++ )
                if((fds[nopen].fd = open(av[f], O_RDONLY)) < 0 ){
                   fprintf(stderr, "Can't read %s\n", av[f] );
                   continue;
                } else {
                   FD_SET(fds[nopen].fd, &set );   /* учесть в маске */
                      maxfd = max(maxfd, fds[nopen].fd );
                   strncpy(fds[nopen].name, av[f], sizeof(fds[0].name) - 1);
                   nopen++;
                }

        if( nopen == 3 ){
                fprintf(stderr, "Nothing is opened\n");
                exit(1);
        }

        FD_SET(fdin, &set); /* учесть stdin */
           maxfd = max(maxfd, fdin );
        nopen -= 2;         /* stdout и stderr не участвуют в select */
        timeout.tv_sec = 10;    /* секунд */
        timeout.tv_usec = 0;    /* миллисекунд */

    /* nfds - это КОЛИЧЕСТВО первых дескрипторов, которые надо
     * просматривать. Здесь можно использовать
     *          nfds = NOFILE;  (кол-во ВСЕХ дескрипторов   )
     * или      nfds = maxfd+1; (кол-во = номер последнего+1)
     * ( +1 т.к. нумерация fd идет с номера 0, а количество - с 1).
     */
        nfds = maxfd + 1;
        while( nopen ){

           rset = set; rtimeout = timeout; /* копируем, т.к. изменятся */
    /* опрашивать можно FIFO-файлы, терминалы, pty, socket-ы, stream-ы */

           nready = select( nfds, &rset, NULL, NULL, &rtimeout );

    /* Если вместо &rtimeout написать NULL, то ожидание будет
     * бесконечным (пока не собьют сигналом)
     */
           if( nready <= 0 ){  /* ничего не поступило */
                fprintf(stderr, "Timed out, nopen=%d\n", nopen);
                continue;
           }
```

```
                /* опрос готовых дескрипторов */
                for(f=0; f < nfds; f++ )
                     if( FD_ISSET(f, &rset)){  /* дескриптор f готов */
                             int n;

                             /* Вызов FIONREAD позволяет запросить
                              * число байт готовых к передаче
                              * через дескриптор.
                              */
                             if(ioctl(f, FIONREAD, &n) < 0)
                                     perror("FIONREAD");
                             else printf("%s have %d bytes.\n", N(f), n);

                             if((n = read(f, buf, sizeof buf)) <= 0 ){
                     eof:
                                 FD_CLR(f, &set); /* исключить */
                                 close(f); nopen--;
                                 fprintf(stderr, "EOF in %s\n", N(f));

                             } else {

                                 fprintf(stderr, "\n%d bytes from %s:\n", n, N(f));
                                 write(fdout, buf, n);
                                 if( n == 4 && !strncmp(buf, "end\n", 4))
                                 /* ncmp, т.к. buf может не оканчиваться \0 */
                                     goto eof;
                             }
                     }
             }
             exit(0);
     }
```

6.11.2. В качестве самостоятельной работы предлагаем вам пример программы, ведущей протокол сеанса работы. Информацию о псевдотерминалах изучите самостоятельно.

```
/*
 *              script.c
 *      Программа получения трассировки работы других программ.
 *      Используется системный вызов опроса готовности каналов
 *      ввода/вывода select() и псевдотерминал (пара ttyp+ptyp).
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>
#include <sys/param.h>  /* NOFILE */
#include <sys/times.h>
#include <sys/wait.h>
#include <errno.h>

#ifdef TERMIOS
# include <termios.h>
# define TERMIO struct termios
# define GTTY(fd, tadr)  tcgetattr(fd, tadr)
# define STTY(fd, tadr)  tcsetattr(fd, TCSADRAIN, tadr)
#else
# include <termio.h>
# define TERMIO struct termio
# define GTTY(fd, tadr) ioctl(fd, TCGETA,  tadr)
# define STTY(fd, tadr) ioctl(fd, TCSETAW, tadr)
#endif
```

```
#ifdef __SVR4
# include <stropts.h>   /* STREAMS i/o */
extern char *ptsname();
#endif

#if defined(ISC2_2)
# include <sys/bsdtypes.h>
#else
# include <sys/select.h>
#endif

#ifndef BSIZE
# define BSIZE 512
#endif

#define LOGFILE         "/usr/spool/scriptlog"
#define max(a,b)        ((a) > (b) ? (a) : (b))
extern int errno;
TERMIO told, tnew, ttypmodes;
FILE *fpscript = NULL;  /* файл с трассировкой (если надо) */
int go = 0;

int scriptflg  = 0;
int halfflag   = 0;      /* HALF DUPLEX */
int autoecho   = 0;
char *protocol = "typescript";

#define STDIN   0 /* fileno(stdin)   */
#define STDOUT  1 /* fileno(stdout)  */
#define STDERR  2 /* fileno(stderr)  */

/* какие каналы связаны с терминалом? */
int tty_stdin, tty_stdout, tty_stderr;
int TTYFD;

void wm_checkttys(){
        TERMIO t;
        tty_stdin  = ( GTTY(STDIN,  &t) >= 0 );
        tty_stdout = ( GTTY(STDOUT, &t) >= 0 );
        tty_stderr = ( GTTY(STDERR, &t) >= 0 );

        if     ( tty_stdin  )  TTYFD = STDIN;
        else if( tty_stdout )  TTYFD = STDOUT;
        else if( tty_stderr )  TTYFD = STDERR;
        else {
                fprintf(stderr, "Cannot access tty\n");
                exit(7);
        }
}

/* Описатель трассируемого процесса */
struct ptypair {
        char line[25];          /* терминальная линия: /dev/ttyp? */
        int pfd;                /* дескриптор master pty          */
        long in_bytes;          /* прочтено байт с клавиатуры     */
        long out_bytes;         /* послано байт на экран          */
        int pid;                /* идентификатор процесса         */
        time_t t_start, t_stop; /* время запуска и окончания      */
        char *command;          /* запущенная команда             */
} PP;
```

```
/* Эта функция вызывается при окончании трассируемого процесса -
 * по сигналу SIGCLD
 */
char Reason[128];
void ondeath(sig){
        int pid;
        extern void wm_done();
        int status;
        int fd;

        /* выявить причину окончания процесса */
        while((pid = wait(&status)) > 0 ){
          if( WIFEXITED(status))
              sprintf( Reason, "Pid %d died with retcode %d",
                                pid, WEXITSTATUS(status));
            else if( WIFSIGNALED(status)) {
              sprintf( Reason, "Pid %d killed by signal #%d",
                                pid, WTERMSIG(status));
#ifdef WCOREDUMP
              if(WCOREDUMP(status)) strcat( Reason, " Core dumped" );
#endif
            } else if( WIFSTOPPED(status))
              sprintf( Reason, "Pid %d suspended by signal #%d",
                                pid, WSTOPSIG(status));
        }
        wm_done(0);
}

void wm_init(){
        wm_checkttys();

        GTTY(TTYFD, &told);

        /* Сконструировать "сырой" режим для нашего _базового_ терминала */
        tnew = told;

        tnew.c_cc[VINTR]  = '\0';
        tnew.c_cc[VQUIT]  = '\0';
        tnew.c_cc[VERASE] = '\0';
        tnew.c_cc[VKILL]  = '\0';
#ifdef VSUSP
        tnew.c_cc[VSUSP]  = '\0';
#endif

        /* CBREAK */
        tnew.c_cc[VMIN]   = 1;
        tnew.c_cc[VTIME]  = 0;

        tnew.c_cflag &= ~(PARENB|CSIZE);
        tnew.c_cflag |=   CS8;
        tnew.c_iflag &= ~(ISTRIP|ICRNL);
        tnew.c_lflag &= ~(ICANON|ECHO|ECHOK|ECHOE|XCASE);

        tnew.c_oflag &= ~OLCUC;
        /* но оставить c_oflag ONLCR и TAB3, если они были */

        /* моды для псевдотерминала */
        ttypmodes = told;
        /* не выполнять преобразования на выводе:
         * ONLCR:       \n --> \r\n
         * TAB3:        \t --> пробелы
         */
        ttypmodes.c_oflag &= ~(ONLCR|TAB3);

        (void) signal(SIGCLD, ondeath);
}
```

```
void wm_fixtty(){
        STTY(TTYFD, &tnew);
}
void wm_resettty(){
        STTY(TTYFD, &told);
}

/* Подобрать свободный псевдотерминал для трассируемого процесса */
struct ptypair wm_ptypair(){
        struct ptypair p;

#ifdef __SVR4
        p.pfd = (-1); p.pid = 0;
        p.in_bytes = p.out_bytes = 0;

        /* Открыть master side пары pty (еще есть slave) */
        if((p.pfd = open( "/dev/ptmx", O_RDWR)) < 0 ){
            /* Это клонируемый STREAMS driver.
             * Поскольку он клонируемый, то есть создающий новое псевдоустройство
             * при каждом открытии, то на master-стороне может быть только
             * единственный процесс!
             */
            perror( "Open /dev/ptmx" );
            goto err;
        }

# ifdef notdef
        /* Сделать права доступа к slave-стороне моими. */
        if( grantpt (p.pfd) < 0 ){
                perror( "grantpt");
                exit(errno);
        }
# endif
        /* Разблокировать slave-сторону псевдотерминала:
           позволить первый open() для нее */
        if( unlockpt(p.pfd) < 0 ){
                perror( "unlockpt");
                exit(errno);
        }

        /* Получить и записать имя нового slave-устройства-файла. */
        strcpy( p.line, ptsname(p.pfd));

#else
        register i;
        char c;
        struct stat st;

        p.pfd = (-1); p.pid = 0;
        p.in_bytes = p.out_bytes = 0;

        strcpy( p.line, "/dev/ptyXX" );
```

```
        for( c = 'p'; c <= 's'; c++ ){
                p.line[ strlen("/dev/pty") ] = c;
                p.line[ strlen("/dev/ptyp")] = '0';
                if( stat(p.line, &st) < 0 )
                        goto err;
                for(i=0; i < 16; i++){
                        p.line[ strlen("/dev/ptyp") ] =
                                "0123456789abcdef" [i] ;
                        if((p.pfd = open( p.line, O_RDWR )) >= 0 ){
                                p.line[ strlen("/dev/") ] = 't';
                                return p;
                        }
                }
        }
#endif
err:    return p;
}

/* Ведение статистики по вызовам script */
void write_stat( in_bytes, out_bytes, time_here , name, line, at )
        long in_bytes, out_bytes;
        time_t time_here;
        char *name;
        char *line;
        char *at;
{
        FILE *fplog;
        struct flock lock;

        if((fplog = fopen( LOGFILE, "a" )) == NULL )
                return;

        lock.l_type   = F_WRLCK;
        lock.l_whence = 0;
        lock.l_start  = 0;
        lock.l_len    = 0;  /* заблокировать весь файл */
        fcntl  ( fileno(fplog), F_SETLKW, &lock );

        fprintf( fplog, "%s (%s) %ld bytes_in %ld bytes_out %ld secs %s %s %s",
                         PP.command, Reason, in_bytes, out_bytes,
                         time_here, name, line, at );
        fflush ( fplog );

        lock.l_type = F_UNLCK;
        lock.l_whence = 0;
        lock.l_start  = 0;
        lock.l_len    = 0;  /* разблокировать весь файл */
        fcntl  ( fileno(fplog), F_SETLK, &lock );

        fclose ( fplog );
}
```

```
void wm_done(sig){
        char *getlogin(), *getenv(), *logname = getlogin();
        time( &PP.t_stop );  /* запомнить время окончания */

        wm_resettty();   /* восстановить режим базового терминала */
        if( fpscript )
                fclose(fpscript);
        if( PP.pid > 0 ) kill( SIGHUP, PP.pid ); /* "обрыв связи" */

        if( go ) write_stat( PP.in_bytes, PP.out_bytes,
                             PP.t_stop - PP.t_start,
                             logname ? logname : getenv("LOGNAME"),
                             PP.line, ctime(&PP.t_stop) );
        printf( "\n" );
        exit(0);
}

/* Запуск трассируемого процесса на псевдотерминале */
void wm_startshell (ac, av)
        char **av;
{
        int child, fd, sig;

        if( ac == 0 ){
                static char *avshell[] = { "/bin/sh", "-i", NULL };
                av = avshell;
        }
        if((child = fork()) < 0 ){
                perror("fork");
                wm_done(errno);
        }
        if( child == 0 ){       /* SON */
                if( tty_stdin )
                        setpgrp(); /* отказ от управляющего терминала */

                /* получить новый управляющий терминал */
                if((fd = open( PP.line, O_RDWR )) < 0 ){
                        exit(errno);
                }

                /* закрыть лишние каналы */
                if( fpscript )
                        fclose(fpscript);
                close( PP.pfd );

#ifdef __SVR4
                /* Push pty compatibility modules onto stream */
                ioctl(fd, I_PUSH, "ptem");      /* pseudo tty module */
                ioctl(fd, I_PUSH, "ldterm");    /* line discipline module */
                ioctl(fd, I_PUSH, "ttcompat"); /* BSD ioctls module */
#endif
```

```
                /* перенаправить каналы, связанные с терминалом */
                if( fd != STDIN  && tty_stdin  ) dup2(fd, STDIN);
                if( fd != STDOUT && tty_stdout ) dup2(fd, STDOUT);
                if( fd != STDERR && tty_stderr ) dup2(fd, STDERR);
                if( fd > STDERR )
                    (void) close(fd);

                /* установить моды терминала */
                STTY(TTYFD, &ttypmodes);

                /* восстановить реакции на сигналы */
                for(sig=1; sig < NSIG; sig++)
                        signal( sig, SIG_DFL );

                execvp(av[0], av);
                system( "echo OBLOM > HELP.ME");
                perror("execl");
                exit(errno);

        } else {                /* FATHER */
                PP.pid = child;
                PP.command = av[0];
                time( &PP.t_start ); PP.t_stop = PP.t_start;

                signal( SIGHUP,  wm_done );
                signal( SIGINT,  wm_done );
                signal( SIGQUIT, wm_done );
                signal( SIGTERM, wm_done );
                signal( SIGILL,  wm_done );
                signal( SIGBUS,  wm_done );
                signal( SIGSEGV, wm_done );
        }
}

char buf[ BSIZE ];   /* буфер для передачи данных */

/*                                /dev/pty?    /dev/ttyp?
     экран          *---------*          *---------*
     /|||           |         |  PP.pfd  |         |
    |||||<-STDOUT--|  мой    |<---------| псевдо  |<-STDOUT---|
     \|||           |терминал|          |терминал|<-STDERR---|трассируемый
                    |(базовый)          |         |           |процесс
    -------         |         |  STDIN   |         |           |
    |.....|-STDIN-->|         |--------->|         |--STDIN--->|
    |_____|         |         |          |         |
    клавиатура      *---------*          *---------*
                                      master     slave

*/
```

```
/* Опрос дескрипторов */
void wm_select(){
        int nready;
        int nfds;
        int maxfd;
        int nopen;  /* число опрашиваемых дескрипторов */
        register f;

        fd_set set, rset;       /* маски */
        struct timeval timeout, rtimeout;

        FD_ZERO(&set); nopen = 0;        /* очистка маски */

        FD_SET (PP.pfd, &set); nopen++;  /* учесть в маске */
        FD_SET (STDIN,  &set); nopen++;
        maxfd = max(PP.pfd, STDIN);

        timeout.tv_sec = 3600;      /* секунд */
        timeout.tv_usec = 0;        /* миллисекунд */

        nfds =  maxfd + 1;
        while( nopen ){
           rset = set;
           rtimeout = timeout;

           /* опросить дескрипторы */
           if((nready = select( nfds, &rset, NULL, NULL, &rtimeout )) <= 0)
                continue;

           for(f=0; f < nfds; f++ )
                if( FD_ISSET(f, &rset)){  /* дескриптор f готов */
                        int n;

                        if((n = read(f, buf, sizeof buf)) <= 0 ){
                            FD_CLR(f, &set); nopen--; /* исключить */
                            close(f);

                        } else {
                            int fdout;

                            /* учет и контроль */
                            if( f == PP.pfd ){
                                fdout = STDOUT;
                                PP.out_bytes += n;
                                if( fpscript )
                                    fwrite(buf, 1, n, fpscript);

                            } else if( f == STDIN  ) {
                                fdout = PP.pfd;
                                PP.in_bytes += n;
                                if( halfflag && fpscript )
                                    fwrite(buf, 1, n, fpscript);
                                if( autoecho )
                                    write(STDOUT, buf, n);
                            }
                            write(fdout, buf, n);
                        }
                }
        }
}
```

```
int main(ac, av) char **av;
{
        while( ac > 1 && *av[1] == '-' ){
                switch(av[1][1]){
                case 's':
                        scriptflg++;
                        break;
                case 'f':
                        av++; ac--;
                        protocol = av[1];
                        scriptflg++;
                        break;
                case 'h':
                        halfflag++;
                        break;
                case 'a':
                        autoecho++;
                        break;
                default:
                        fprintf(stderr, "Bad key %s\n", av[1]);
                        break;
                }
                ac--; av++;
        }
        if( scriptflg ){
                fpscript = fopen( protocol, "w" );
        }
        ac--; av++;

        wm_init();
        PP = wm_ptypair();
        if( PP.pfd < 0 ){
                fprintf(stderr, "Cannot get pty. Please wait and try again.\n");
                return 1;
        }
        wm_fixtty();
        wm_startshell(ac, av);
        go++;
        wm_select();
        wm_done(0);
        /* NOTREACHED */
        return 0;
}
```

6.12. Простой интерпретатор команд.

Данный раздел просто приводит исходный текст простого интерпретатора команд. Функция **match** описана в главе "Текстовая обработка".

```
/* Примитивный интерпретатор команд. Распознает построчно
 * команды вида: CMD ARG1 ... ARGn <FILE >FILE >>FILE >&FILE >>&FILE
 * Сборка: cc -U42 -DCWDONLY sh.c match.c pwd.c -o sh
 */

#include <sys/types.h>/* определение типов, используемых системой */
#include <stdio.h>      /* описание библиотеки ввода/вывода  */
#include <signal.h>     /* описание сигналов                 */
#include <fcntl.h>      /* определение O_RDONLY              */
#include <errno.h>      /* коды системных ошибок             */
#include <ctype.h>      /* макросы для работы с символами    */
#include <dirent.h>     /* эмуляция файловой системы BSD 4.2 */
#include <pwd.h>        /* работа с /etc/passwd              */
#include <sys/wait.h>   /* описание формата wait()           */
```

```
char cmd[256];          /* буфер для считывания команды */
#define   MAXARGS 256   /* макс. количество аргументов  */
char *arg[MAXARGS];     /* аргументы команды */
char *fin, *fout;       /* имена для перенаправления ввода/вывода */
int  rout;              /* флаги перенаправления вывода */

char *firstfound;       /* имя найденной, но невыполняемой программы */
#define LIM ':'         /* разделитель имен каталогов в path */
extern char *malloc(), *getenv(), *strcpy(), *getwd();
extern char *strchr(), *execat();
extern void callshell(), printenv(), setenv(), dowait(), setcwd();
extern struct passwd   *getpwuid();
        /* Предопределенные переменные */
extern char **environ;  /* окружение: изначально смотрит на тот же
                         * массив, что и ev из main() */
extern int errno;       /* код ошибки системного вызова   */

char *strdup(s)char *s;
{ char *p; return(p=malloc(strlen(s)+1), strcpy(p,s)); }
        /* strcpy() возвращает свой первый аргумент */
char *str3spl(s, p, q) char *s, *p, *q;
{ char *n = malloc(strlen(s)+strlen(p)+strlen(q)+1);
  strcpy(n, s); strcat(n, p); strcat(n, q); return n;
}

int cmps(s1, s2) char **s1, **s2;
{ return strcmp(*s1, *s2); }

/* Перенаправить вывод */
#define APPEND 0x01
#define ERRTOO 0x02
int output (name, append, err_too, created) char *name; int *created;
{
    int     fd;
    *created = 0;     /* Создан ли файл ? */

    if( append ){                   /* >>file */
        /* Файл name существует? Пробуем открыть на запись */
        if((fd = open (name, O_WRONLY)) < 0) {
            if (errno == ENOENT) /* Файл еще не существовал */
                goto CREATE;
            else
                return 0;    /* Не имеем права писать в этот файл */
        }
        /* иначе fd == открытый файл, *created == 0 */
    }else{
CREATE: /* Пытаемся создать (либо опустошить) файл "name" */
        if((fd = creat (name, 0666)) < 0 )
                    return 0;     /* Не могу создать файл  */
        else        *created = 1; /* Был создан новый файл */
    }
    if (append)
        lseek (fd, 01, 2);      /* на конец файла */
 /* перенаправить стандартный вывод */
    dup2(fd, 1);
    if( err_too ) dup2(fd, 2);  /* err_too=1 для >& */
    close(fd); return 1;
}
```

```
/* Перенаправить ввод */
int input (name) char *name;
{
    int     fd;
    if((fd = open (name, O_RDONLY)) < 0 ) return 0;/* Не могу читать */
 /* перенаправить стандартный ввод */
    dup2(fd, 0); close(fd); return 1;
}

/* запуск команды */
int cmdExec(progr, av, envp, inp, outp, outflg)
        char *progr;       /* имя программы */
        char **av;         /* список аргументов */
        char **envp;       /* окружение */
        char *inp, *outp;  /* файлы ввода-вывода (перенаправления) */
        int outflg;        /* режимы перенаправления вывода */
{
        void (*del)(), (*quit)();
        int pid;
        int cr = 0;

        del = signal(SIGINT, SIG_IGN); quit = signal(SIGQUIT, SIG_IGN);
        if( ! (pid = fork())){     /* ветвление */
           /* порожденный процесс (сын) */
           signal(SIGINT, SIG_DFL); /* восстановить реакции */
           signal(SIGQUIT,SIG_DFL); /* по умолчанию         */
           /* getpid() выдает номер (идентификатор) данного процесса */
           printf( "Процесс pid=%d запущен\n", pid = getpid());

           /* Перенаправить ввод-вывод */
           if( inp ) if(!input( inp )){
             fprintf(stderr, "Не могу <%s\n", inp ); goto Err;
           }
           if( outp )
               if(!output (outp, outflg & APPEND, outflg & ERRTOO, &cr)){
                   fprintf(stderr, "Не могу >%s\n", outp ); goto Err;
               }
 /* Заменить программу: при успехе
  * данная программа завершается, а вместо нее вызывается
  * функция main(ac, av, envp) программы, хранящейся в файле progr.
  * ac вычисляет система.
  */
           execvpe(progr, av, envp);

Err:
  /* при неудаче печатаем причину и завершаем порожденный процесс */
           perror(firstfound ? firstfound: progr);
           /* Мы не делаем free(firstfound),firstfound = NULL
            * потому что данный процесс завершается (и тем ВСЯ его
            * память освобождается) :
            */
           if( cr && outp )  /* был создан новый файл     */
               unlink(outp); /* но теперь он нам не нужен */

           exit(errno);
        }
        /* процесс - отец */

        /* Сейчас сигналы игнорируются, wait не может быть оборван
         * прерыванием с клавиатуры */
        dowait();       /* ожидать окончания сына */
        /* восстановить реакции на сигналы от клавиатуры */
        signal(SIGINT, del); signal(SIGQUIT, quit);
        return pid;     /* вернуть идентификатор сына */
}
```

```
/* Запуск программы с поиском по переменной среды PATH */
int execvpe(progr, av, envp) char *progr, **av, **envp;
{
        char *path, *cp;
        int try = 1;
        register eacces = 0;
        char fullpath[256];     /* полное имя программы */

        firstfound = NULL;
        if((path = getenv("PATH")) == NULL )
            path = ".:/bin:/usr/bin:/etc";
        /* имя: короткое или путь уже задан ? */
        cp = strchr(progr, '/') ? "" : path;
        do{     /* пробуем разные варианты */
                cp = execat(cp, progr, fullpath);
        retry:
            fprintf(stderr, "пробуем \"%s\"\n", fullpath );
                execve(fullpath, av, envp);
                /* если программа запустилась, то на этом месте данный
                 * процесс заменился новой программой. Иначе - ошибка. */
                switch( errno ){  /* какова причина неудачи ? */
                case ENOEXEC:  /* это командный файл */
                        callshell(fullpath, av, envp);
                        return (-1);
                case ETXTBSY:   /* файл записывается */
                        if( ++try > 5 ) return (-1);
                        sleep(try); goto retry;
                case EACCES:    /* не имеете права */
                        if(firstfound == NULL)
                           firstfound = strdup(fullpath);
                        eacces++; break;
                case ENOMEM:    /* программа не лезет в память */
                case E2BIG:
                        return (-1);
                }
        }while( cp );
        if( eacces ) errno = EACCES;
        return (-1);
}

/* Склейка очередной компоненты path и имени программы name */
static char *execat(path, name, buf)
        register char *path, *name;
        char *buf;      /* где будет результат */
{
        register char *s = buf;
        while(*path && *path != LIM )
                *s++ = *path++;         /* имя каталога */
        if( s != buf ) *s++ = '/';
        while( *name )
                *s++ = *name++;         /* имя программы */
        *s = '\0';
        return ( *path ? ++path /* пропустив LIM */ : NULL );
}
```

```
/* Запуск командного файла при помощи вызова интерпретатора */
void callshell(progr, av, envp) char *progr, **av, **envp;
{
        register i; char *sh; char *newav[MAXARGS+2];
        int fd; char first = 0;

        if((fd = open(progr, O_RDONLY)) < 0 )
                sh = "/bin/sh";
        else{
                read(fd, &first, 1); close(fd);
                sh = (first == '#') ? "/bin/csh" : "/bin/sh";
        }
        newav[0] = "Shellscript"; newav[1] = progr;
        for(i=1; av[i]; i++)
                newav[i+1] = av[i];
        newav[i+1] = NULL;
        printf( "Вызываем %s\n", sh );
        execve(sh, newav, envp);
}

/* Ожидать окончания всех процессов, выдать причины смерти. */
void dowait(){
        int ws; int pid;

        while((pid = wait( &ws)) > 0 ){
          if( WIFEXITED(ws)){
            printf( "Процесс %d умер с кодом %d\n",
                            pid,            WEXITSTATUS(ws));
          }else if( WIFSIGNALED(ws)){
            printf( "Процесс %d убит сигналом %d\n",
                            pid,             WTERMSIG(ws));
            if(WCOREDUMP(ws)) printf( "Образовался core\n" );
            /* core - образ памяти процесса для отладчика adb */
          }else if( WIFSTOPPED(ws)){
            printf( "Процесс %d остановлен сигналом %d\n",
                            pid,            WSTOPSIG(ws));
          }
        }
}
```

```
/* Расширение шаблонов имен. Это упрощенная версия, которая
 * расширяет имена только в текущем каталоге.
 */
void glob(dir, args, indx, str /* что расширять */, quote )
     char *args[], *dir; int *indx; char *str;
     char quote; /* кавычки, в которые заключена строка str */
{
        static char globchars[] = "*?[";
        char *p; char **start = &args[ *indx ];
        short nglobbed = 0;

        register struct dirent *dirbuf;
        DIR *fd; extern DIR *opendir();

        /* Затычка для отмены глоббинга: */
        if( *str == '\\' ){ str++;    goto noGlob; }

        /* Обработка переменных $NAME    */
        if( *str == '$' && quote != '\'' ){
                char *s = getenv(str+1);
                if( s ) str = s;
        }
        /* Анализ: требуется ли глоббинг */
        if( quote ) goto noGlob;
        for( p=str; *p; p++ )  /* Есть ли символы шаблона? */
                if( strchr(globchars, *p))
                        goto doGlobbing;
noGlob:
        args[ (*indx)++ ] = strdup(str);
        return;

doGlobbing:
       if((fd = opendir (dir)) == NULL){
              fprintf(stderr, "Can't read %s\n", dir); return;
       }
       while ((dirbuf = readdir (fd)) != NULL ) {
              if (dirbuf->d_ino == 0) continue;
              if (strcmp (dirbuf->d_name, ".") == 0 ||
                  strcmp (dirbuf->d_name, "..") == 0) continue;
              if( match( dirbuf->d_name, str)){
                  args[ (*indx)++ ] = strdup(dirbuf->d_name);
                  nglobbed++;
              }
       }
       closedir(fd);
       if( !nglobbed){
           printf( "%s: no match\n", str);
           goto noGlob;
       }else{   /* отсортировать */
           qsort(start, nglobbed, sizeof (char *), cmps);
       }
}

/* Разбор командной строки */
int parse(s) register char *s;
{
        int i; register char *p;
        char tmp[80];   /* очередной аргумент */
        char c;

        /* очистка старых аргументов */
        for(i=0; arg[i]; i++) free(arg[i]), arg[i] = NULL;
        if( fin  ) free(fin ), fin  = NULL;
        if( fout ) free(fout), fout = NULL;
        rout = 0;
```

```
        /* разбор строки */
        for( i=0 ;; ){
                char quote = '\0';

                /* пропуск пробелов - разделителей слов */
                while((c = *s) && isspace(c)) s++;
                if( !c ) break;
                /* очередное слово */
                p = tmp;
                if(*s == '\'' || *s == '"' ){
                /* аргумент в кавычках */
                        quote = *s++;  /* символ кавычки */
                        while((c = *s) != '\0' && c != quote){
                           if( c == '\\' ){ /* заэкранировано */
                               c = *++s;
                               if( !c ) break;
                           }
                           *p++ = c; ++s;
                        }
                        if(c == '\0')
                  fprintf(stderr, "Нет закрывающей кавычки %c\n", quote);
                        else s++; /* проигнорировать кавычку на конце */

                } else
                        while((c = *s) && !isspace(c)){
                           if(c == '\\') /* заэкранировано */
                                if( !(c = *++s))
                                     break /* while */;
                           *p++ = c; s++;
                        }
                *p = '\0';
                /* Проверить, не есть ли это перенаправление
                 * ввода/вывода. В отличие от sh и csh
                 * здесь надо писать >ФАЙЛ <ФАЙЛ
                 * >< вплотную к имени файла.
                 */
                p = tmp;        /* очередное слово */
                if( *p == '>'){ /* перенаправлен вывод */
                    p++;
                    if( fout ) free(fout), rout = 0; /* уже было */
                    if( *p == '>' ){ rout |= APPEND; p++; }
                    if( *p == '&' ){ rout |= ERRTOO; p++; }
                    if( !*p ){
                            fprintf(stderr, "Нет имени для >\n");
                            fout = NULL; rout = 0;
                    } else  fout = strdup(p);
                } else if( *p == '<' ){  /* перенаправлен ввод */
                    p++;
                    if( fin ) free(fin);        /* уже было */
                    if( !*p ){
                            fprintf(stderr, "Нет имени для <\n");
                            fin = NULL;
                    } else  fin = strdup(p);
                } else /* добавить имена к аргументам */
                            glob( ".", arg, &i, p, quote );
        }
        arg[i] = NULL; return i;
}
```

```
/* Установить имя пользователя */
void setuser(){
        int uid = getuid();     /* номер пользователя, запустившего Шелл */
        char *user = "mr. Nobody";      /* имя пользователя     */
        char *home = "/tmp";            /* его домашний каталог */
        struct passwd *pp = getpwuid( uid );
        if( pp != NULL ){
                if(pp->pw_name && *pp->pw_name ) user = pp->pw_name;
                if(               *pp->pw_dir  ) home = pp->pw_dir;
        }
        setenv("USER", user); setenv("HOME", home);
}

void setcwd(){ /* Установить имя текущего каталога */
        char cwd[512];
        getwd(cwd); setenv( "CWD", cwd );
}

void main(ac, av, ev) char *av[], *ev[]; {
        int argc;               /* количество аргументов */
        char *prompt;           /* приглашение           */

        setuser(); setcwd();
        signal(SIGINT, SIG_IGN);
        setbuf(stdout, NULL);  /* отменить буферизацию */
        for(;;){
             prompt = getenv( "prompt" ); /* setenv prompt -->\  */
             printf( prompt ? prompt : "@ ");/* приглашение */
             if( gets(cmd) == NULL /* at EOF  */ ) exit(0);
             argc = parse(cmd);
             if( !argc) continue;
             if( !strcmp(arg[0], "exit" )) exit(0);

             if( !strcmp(arg[0], "cd" )){
                 char *d = (argc==1) ? getenv("HOME"):arg[1];
                 if(chdir(d) < 0)
                    printf( "Не могу войти в %s\n", d );
                 else setcwd();
                 continue;
             }

             if( !strcmp(arg[0], "echo" )){
                 register i; FILE *fp;
                 if( fout ){
                    if((fp = fopen(fout, rout & APPEND ? "a":"w"))
                           == NULL) continue;
                 }  else fp = stdout;
                 for(i=1; i < argc; i++ )
                    fprintf( fp, "%s%s", arg[i], i == argc-1 ? "\n":"   ");
                 if( fp != stdout ) fclose(fp);
                 continue;
             }

             if( !strcmp(arg[0], "setenv" )){
                      if( argc == 1 ) printenv();
                 else if( argc == 2 ) setenv( arg[1], "" );
                 else                 setenv( arg[1], arg[2]);
                 continue;
             }
             cmdExec(arg[0], (char **) arg, environ, fin, fout, rout);
        }
}
```

```
/* ------------------------------------------------------------*/
/* Отсортировать и напечатать окружение */
void printenv(){
        char *e[40]; register i = 0; char *p, **q = e;

        do{
             p = e[i] = environ[i]; i++;
        } while( p );

#ifdef SORT
        qsort( e, --i /* сколько */, sizeof(char *), cmps);
#endif
        while( *q )
             printf( "%s\n", *q++ );
}

/* Сравнение имени переменной окружения с name */
static char *envcmp(name, evstr) char *name, *evstr;
{
        char *p; int code;
        if((p = strchr(evstr, '=')) == NULL ) return NULL; /* error ! */
        *p = '\0';      /* временно */
        code = strcmp(name, evstr);
        *p = '=';       /* восстановили */
        return code==0 ? p+1 : NULL;
}

/* Установить переменную окружения */
void setenv( name, value ) char *name, *value;
{
        static malloced = 0;    /* 1, если environ перемещен */
        char *s, **p, **newenv;
        int len, change_at = (-1), i;

        /* Есть ли переменная name в environ-е ? */
        for(p = environ; *p; p++ )
                if(s = envcmp(name, *p)){       /* уже есть */
                        if((len = strlen(s)) >= strlen(value)){
                            /* достаточно места */
                            strcpy(s, value); return;
                        }
                        /* Если это новый environ ... */
                        if( malloced ){
                            free( *p ); *p = str3spl(name, "=", value);
                            return;
                        }
                        /* иначе создаем копию environ-а */
                        change_at = p - environ;        /* индекс */
                        break;
                }

        /* Создаем копию environ-а. Если change_at == (-1), то
         * резервируем новую ячейку для еще не определенной переменной */
        for(p=environ, len=0; *p; p++, len++ );
        /* вычислили количество переменных */
        if( change_at < 0 ) len++;
        if((newenv = (char **) malloc( sizeof(char *) * (len+1)))
                == (char **) NULL) return;
        for(i=0; i < len+1; i++ ) newenv[i] = NULL;     /* зачистка */
        /* Копируем старый environ в новый */
        if( !malloced ) /* исходный environ в стеке (дан системой) */
             for(i=0; environ[i]; i++ ) newenv[i] = strdup(environ[i]);
        else for(i=0; environ[i]; i++ ) newenv[i] =        environ[i];
        /* Во втором случае строки уже были спасены, копируем ссылки */
```

```
        /* Изменяем, если надо: */
        if( change_at >= 0 ){
                free( newenv[change_at] );
                newenv[change_at] = str3spl(name, "=", value);
        } else {        /* добавить в конец новую переменную */
                newenv[len-1] = str3spl(name, "=", value);
        }
        /* подменить environ */
        if( malloced ) free( environ );
        environ = newenv; malloced++;
        qsort( environ, len, sizeof(char *), cmps);
}

/* Допишите команды:
   unsetenv имя_переменной - удаляет переменную среды;
   exit N                  - завершает интерпретатор с
                             кодом возврата N (это целое число);
*/
```

7. Текстовая обработка.

Под "текстовой обработкой" (в противовес "вычислительным задачам") здесь понимается огромный класс задач обработки информации нечислового характера, например редактирование текста, форматирование документов, поиск и сортировка, базы данных, лексический и синтаксический анализ, печать на принтере, преобразование формата таблиц, и.т.п.

7.1. Напишите программу, "угадывающую" слово из заранее заданного списка по первым нескольким буквам. Выдайте сообщение "неоднозначно", если есть несколько похожих слов. Усложните программу так, чтобы список слов считывался в программу при ее запуске из файла *list.txt*

7.2. Напишите программу, которая удваивает пробелы в тексте с одиночными пробелами.

7.3. Напишите программу, которая копирует ввод на вывод, заменяя каждую последовательность из идущих подряд нескольких пробелов и/или табуляций на *один* пробел. Схема ее решения сходна с решением следующей задачи.

7.4. Напишите программу подсчета слов в файле. Слово определите как последовательность символов, не включающую символы пробела, табуляции или новой строки. "Канонический" вариант решения, приведенный у Кернигана и Ритчи, таков:

```
#include <ctype.h>
#include <stdio.h>
const int YES=1, NO=0;
main(){
  register int inWord = NO; /* состояние */
  int words = 0, c;
  while((c = getchar()) != EOF)
    if(isspace(c) || c == '\n') inWord = NO;
    else if(inWord == NO){
         inWord = YES; ++words;
    }
  printf("%d слов\n", words);
}
```

Обратите внимание на конструкцию **const**. Это объявление имен как констант. Эта конструкция близка к
#define YES 1
но позволяет компилятору

- более строго проверять тип, т.к. это *типизированная* константа;
- создавать более экономный код;
- запрещает изменять это значение.

Рассмотрим пример

```
main(){  /* cc 00.c -o 00 -lm */
  double sqrt(double);
  const double sq12 = sqrt(12.0);
#define        SQRT2  sqrt(2.0)
  double x;
  x = sq12 * sq12 * SQRT2 * SQRT2;     /* @1 */
  sq12 = 3.4641;                       /* @2 */
  printf("%g %g\n", sq12, x);
}
```

Использование #define превратит строку **@1** в

```
  x = sq12 * sq12 * sqrt(2.0) * sqrt(2.0);
```

то есть создаст код с двумя вызовами функции **sqrt**. Конструкция же **const** заносит вычисленное выражение в ячейку памяти и далее просто использует ее значение. При этом компилятор не позволяет впоследствии изменять это значение, поэтому строка **@2** ошибочна.

Теперь предложим еще одну программу подсчета слов, где слово определяется макросом **isWord**, перечисляющим буквы допустимые в слове. Программа основана на переключательной таблице функций (этот подход применим во многих случаях):

```
#include <ctype.h>
#include <stdio.h>
int wordLength, inWord, words; /* = 0 */
char aWord[128], *wrd;
```

```
void space  (c){}
void letter (c){ wordLength++; *wrd++ = c; }
void begWord(c){ wordLength=0; inWord=1;
     wrd=aWord;  words++; letter(c);       }
void endWord(c){ inWord=0; *wrd = '\0';
        printf("Слово '%s' длины %d\n",
                        aWord,    wordLength); }
void (*sw[2][2])() = {
/* !isWord */ { space,   endWord },
/*  isWord */ { begWord, letter  }
            /* !inWord   inWord  */
};
#define isWord(c) (isalnum(c) || c=='-' || c=='_')

main(){ register c;
  while((c = getchar()) != EOF)
    (*sw[isWord(c)][inWord])(c);
  printf("%d слов\n", words);
}
```

7.5. Напишите программу, выдающую гистограмму длин строк файла (т.е. таблицу: строк длины 0 столько-то, длины 1 - столько-то, и.т.п., причем таблицу можно изобразить графически).

7.6. Напишите программу, которая считывает слово из файла *in* и записывает это слово в конец файла *out*.

7.7. Напишите программу, которая будет печатать слова из файла ввода, причем по одному на строку.

7.8. Напишите программу, печатающую гистограмму длин слов из файла ввода.

7.9. Напишите программу, читающую слова из файла и размещающую их в виде двунаправленного списка слов, отсортированного по алфавиту. Указания: используйте динамическую память (**malloc**) и указатели; напишите функцию включения нового слова в список на нужное место.

В конце работы распечатайте список дважды: в прямом и в обратном порядке.

Усложнение: не хранить в списке дубликаты; вместо этого вместе со словом хранить счетчик количества его вхождений в текст.

7.10. Напишите программу, которая печатает слова из своего файла ввода, расположенные в порядке убывания частоты их появления. Перед каждым словом напечатайте число частоты его появления.

7.11. Напишите программу, читающую файл построчно и печатающую слова в каждой строке в обратном порядке.

7.12. Напишите программу копирования ввода на вывод таким образом, чтобы из каждой группы последовательно одинаковых строк выводилась только одна строка. Это аналог программы **uniq** в системе **UNIX**. Ответ:

```
#include <stdio.h>      /* char *gets(); */
char buf1[4096], buf2[4096];
char *this = buf1, *prev = buf2;
main(){
  long nline =0L; char *tmp;
  while( gets(this)){
   if(nline){ /* сравнить новую и предыдущую строки */
       if( strcmp(this, prev)) /* различны ? */
           puts(prev);
   }
   /* обмен буферов: */ tmp=prev; prev=this; this=tmp;
   nline++;     /* номер строки */
  }/* endwhile */
  if( nline ) puts(prev);
  /* последняя строка всегда выдается */
}
```

7.13. Составьте программу, которая будет удалять в конце (и в начале) каждой строки файла пробелы и табуляции, а также удалять строки, целиком состоящие из пробелов и табуляций.

7.14. Для экономии места в файле, редакторы текстов при записи отредактированного файла сжимают подряд идущие пробелы в табуляцию. Часто это неудобно для программ обработки текстов (поскольку требует особой обработки табуляций - это ОДИН символ, который на экране и в тексте занимает НЕСКОЛЬКО позиций!), поэтому при чтении файла мы должны расширять табуляции в нужное количество пробелов, например так:

```
/* заменять табуляции на пробелы */
void untab(s) register char *s;
{
  char newstr[256];  /* новая строка */
  char *src = s;
  int n;             /* счетчик      */
  register dstx;     /* координата x в новой строке */

  for(dstx = 0; *s != '\0'; s++)
      if( *s == '\t'){
          for(n = 8 - dstx % 8 ; n > 0 ; n--)
              newstr[dstx++] = ' ';
      }else   newstr[dstx++] = *s;
  newstr[dstx] = '\0';
  strcpy(src, newstr); /* строку на старое место */
}
```

7.15. Напишите обратную функцию, сжимающую подряд идущие пробелы в табуляции.

```
void tabify(){
   int chr;
   int icol, ocol;         /* input/output columns */

   for(icol = ocol = 0; ; ){

           if((chr = getchar()) == EOF)
                   break;

           switch(chr){

           case ' ':
                   icol++;
                   break;

           case '\n':
           case '\r':
                   ocol = icol = 0;
                   putchar(chr);
                   break;

           case '\t':
                   icol += 8;
                   icol &= ~07;    /* icol -= icol % 8; */
                   break;
```

```
            default:
                    while(((ocol + 8) & ~07) <= icol){

#ifdef NOTDEF
                            if(ocol + 1 == icol)
                                    break;
                                    /* взять ' ' вместо '\t' */
#endif
                            putchar('\t');
                            ocol += 8;
                            ocol &= ~07;
                    }
                    while(ocol < icol){
                            putchar(' ');
                            ocol++;
                    }
                    putchar(chr);
                    icol++;
                    ocol++;
                    break;
            }
    }
}
```

7.16. Составьте программу, укорачивающую строки исходного файла до заданной величины и помещающую результат в указанный файл. Учтите, что табуляция разворачивается в несколько пробелов!

7.17. Разработайте программу, укорачивающую строки входного файла до 60 символов. Однако теперь запрещается обрубать слова.

7.18. Разработайте программу, заполняющую промежутки между словами строки дополнительными пробелами таким образом, чтобы длина строки была равна 60 символам.

7.19. Напишите программу, переносящую слишком длинные строки. Слова разбивать нельзя (неумешающееся слово следует перенести целиком). Ширину строки считать равной 60.

7.20. Составьте программу, центрирующую строки файла относительно середины экрана, т.е. добавляющую в начало строки такое количество пробелов, чтобы середина строки печаталась в 40-ой позиции (считаем, что обычный экран имеет ширину 80 символов).

7.21. Напишите программу, отсекающую *n* пробелов в начале каждой строки (или *n* первых любых символов). Учтите, что в файле могут быть строки короче *n* (например пустые строки).

```
#include <stdio.h>
/* ... текст функции untab(); ... */
void process(char name[], int n, int spacesOnly){
     char line[256]; int length, shift, nline = 0;
     char newname[128]; FILE *fpin, *fpout;
     if((fpin = fopen(name, "r")) == NULL){
         fprintf(stderr, "Не могу читать %s\n", name);
         return;
     }
     sprintf(newname, "_%s", name); /* например */
     if((fpout = fopen(newname, "w")) == NULL){
         fprintf(stderr, "Не могу создать %s\n",
               newname); fclose(fpin); return;
     }
 while(fgets(line, sizeof line, fpin)){ ++nline;
   if((length = strlen(line)) &&
       line[length-1] == '\n')
       line[--length] =  '\0'; /* обрубить '\n' */
   untab(line); /* развернуть табуляции */
   for(shift=0; line[shift] != '\0' && shift < n ;
                                     ++shift)
       if(spacesOnly && line[shift] != ' ') break;
```

```
   if(*line && shift != n ) /* Предупреждение */
       fprintf(stderr,
         "Начало строки #%d слишком коротко\n", nline);
   fprintf(fpout, "%s\n", line+shift);
   /* нельзя было fputs(line+n, fpout);
    * т.к. эта позиция может быть ЗА концом строки
    */
 }
 fclose(fpin); fclose(fpout);
}
void main(int argc, char **argv){
     if( argc != 3 ) exit(1);
     process(argv[2], atoi(argv[1]) /* 8 */, 1);
     exit(0);
}
```

7.22. Напишите программу, разбивающую файл на два по вертикали: в первый файл попадает левая половина исходного файла, во второй - правая. Ширину колонки задавайте из аргументов **main**(). Если же аргумент не указан - 40 позиций.

7.23. Напишите программу сортировки строк в алфавитном порядке. Учтите, что функция **strcmp**() сравнивает строки в порядке кодировки, принятой на данной конкретной машине. Русские буквы, как правило, идут не в алфавитном порядке! Следует написать функцию для алфавитного сравнения отдельных символов и, пользуясь ею, переписать функцию **strcmp**().

7.24. Отсортируйте массив строк по лексикографическому убыванию, игнорируя различия между строчными и прописными буквами.

7.25. Составьте программу дихотомического поиска в отсортированном массиве строк (методом деления пополам).

```
/* Поиск в таблице методом половинного деления: dihotomia */
#include <stdio.h>

struct elem {
        char *name;     /* ключ поиска */
        int   value;
} table[] = {
        /* имена строго по алфавиту */
        { "andrew",  17 },
        { "bill",    23 },
        { "george",  55 },
        { "jack",    54 },
        { "jaw",     43 },
        { "john",    33 },
        { "mike",    99 },
        { "paul",    21 },
        { "sue",     66 },   /* SIZE - 2 */

        { NULL,      -1 },   /* SIZE - 1 */
        /* NULL введен только для распечатки таблицы */
};

#define SIZE (sizeof(table) / sizeof(struct elem))

/* Дихотомический поиск по таблице */
struct elem *find(s, table, size)
        char *s;              /* что найти ? */
        struct elem table[];  /* в чем     ? */
        int size;             /* среди первых size элементов */
{
        register top, bottom, middle;
        register code;

        top    = 0;          /* начало */
        bottom = size - 1;   /* конец:  индекс строки "sue" */

        while( top <= bottom ){
                middle = (top + bottom) / 2;    /* середина */

                /* сравнить строки */
                code = strcmp( s, table[middle].name ) ;

                if( code > 0 ){
                        top = middle + 1;
                }else if( code < 0 ){
                        bottom = middle - 1;
                }else return &table[ middle ];

        }
        return (struct elem *) NULL; /* не нашел */
}
```

```
/* распечатка таблицы */
void printtable(tbl) register struct elem *tbl; {
        for( ; tbl->name != NULL ; tbl++ ){
               printf( "%-15s %d\n", tbl->name, tbl->value );
        }
}

int main(){
        char buf[80];
        struct elem *ptr;

        printtable(table);
        for(;;){
                printf( "-> " );
                if( gets( buf ) == NULL) break; /* EOF */
                if( ! strcmp( buf, "q" ))
                        exit(0);                /* quit: выход */
                ptr = find( buf, table, SIZE-1 );
                if( ptr )
                        printf( "%d\n", ptr->value );
                else {
                        printf( "--- Не найдено ---\n" );
                        printtable(table);
                }
        }
        return 0;
}
```

7.26. Напишем функцию, которая преобразует строку так, что при ее печати буквы в ней будут *подчеркнуты*, а цифры - выделены **жирно**. Формат текста с выделениями, который создается этим примером, является общепринятым в **UNIX** и распознается некоторыми программами: например, программа просмотра файлов **less** (**more**) выделяет такие буквы на экране специальными шрифтами или инверсией фона.

```
#define LEN 9  /* потом напишите 256 */
char input[] = "(xxx+yyy)/123.75=?";
char output[LEN];
void main( void ){
    int len=LEN, i; void bi_conv(); char c;
    bi_conv(input, output, &len);
    if(len > LEN){
       printf("Увеличь LEN до %d\n", len);
       len = LEN;  /* доступный максимум */
    }
    for(i=0; i < len && (c = output[i]); ++i)
        putchar(c);
    putchar('\n');
}

/* Заметьте, что include-файлы не обязательно
 * должны включаться в самом начале программы! */
#include <stdio.h>
#include <ctype.h>

#define PUT(c) { count++;              \
    if(put < *len){ *p++ = (c); ++put;}}
#define GET() (*s ? *s++ : EOF)

void bi_conv(
        /*IN*/    char *s,
        /*OUT*/   char *p,
        /*INOUT*/ int *len ){
    int count, put, c;
    for(count=put=0; (c=GET()) != EOF; ){
    /* жирный:       C\bC    */
    /* подчеркнутый: _\bC    */
             if(isalpha(c)){ PUT('_'); PUT('\b'); }
```

```
        else if(isdigit(c)){ PUT( c ); PUT('\b'); }
        PUT(c);
    }
    PUT('\0');  /* закрыть строку */
    *len = count;
#undef PUT
#undef GET
}
```

Напишите программу для подобной обработки *файла*. Заметим, что для этого не нужны промежуточные строки *input* и *output* и построчное чтение файла; все, что надо сделать, это определить

```
#define PUT(c) if(c)putchar(c)
#define GET()  getchar()
```

Напишите подобную функцию, удваивающую буквы в сстгрроокее.

7.27. Напишите программу, удаляющую из файла выделения. Для этого надо просто удалять последовательности вида *C***b**

```
#include <stdio.h>
#define NOPUT   (-1) /* не символ ASCII */
/* Названия шрифтов - в перечислимом типе */
typedef enum { NORMAL=1, ITALICS, BOLD, RED=BOLD } font;
int ontty; font textfont; /* текущее выделение */
#define setfont(f)      textfont=(f)
#define getfont()       (textfont)
#define SetTtyFont(f)   if(ontty) tfont(f)

/* Установить выделение на экране терминала */
void tfont(font f){ /* только для ANSI терминала */
  static font ttyfont = NORMAL;
  if(ttyfont == f) return;
  printf("\033[0m"); /* set NORMAL font */
  switch(ttyfont = f){
  case NORMAL:  /* уже сделано выше */ break;
  case BOLD:    printf("\033[1m");     break;
  case ITALICS: /* use reverse video */
                printf("\033[7m");     break;
  }
}
void put(int c){ /* Вывод символа текущим цветом */
  if(c == NOPUT) return; /* '\b' */
  SetTtyFont(getfont()); putchar(c);
  setfont(NORMAL); /* Ожидать новой C\b посл-ти */
}
void
main(){ register int c, cprev = NOPUT;
  /* Стандартный вывод - это терминал ? */
  ontty = isatty(fileno(stdout));
  setfont(NORMAL);
  while((c = getchar()) != EOF){
    if(c == '\b'){  /* выделение */
       if((c = getchar()) == EOF) break;
       if(c == cprev)             setfont(BOLD);
       else if(cprev == '_')      setfont(ITALICS);
       else /* наложение A\bB */ setfont(RED);
    } else put(cprev);
    cprev = c;
  }
  put(cprev); /* последняя буква файла */
  SetTtyFont(NORMAL);
}
```

7.28. Напишите программу печати на принтере листинга Си-программ. Ключевые слова языка выделяйте двойной надпечаткой. Для выдачи на терминал напишите программу, подчеркивающую ключевые слова (подчеркивание - в следующей строке). Упрощение: выделяйте не ключевые слова, а большие буквы. Указание: для двойной печати используйте управляющий символ '**\r**' - возврат к началу той же строки; затем

строка печатается повторно, при этом символы, которые не должны печататься жирно, следует заменить на пробелы (или на табуляцию, если этот символ сам есть '**\t**').

7.29. Напишите программу, печатающую тексты Си-программ на принтере. Выделяйте ключевые слова языка жирным шрифтом, строки "*строка*", символы '*с*' и комментарии - курсивом. Шрифты для **EPSON-FX** совместимых принтеров (например **EP-2424**) переключаются такими управляющими последовательностями (**ESC** означает символ '**\033**'):

	ВКЛЮЧЕНИЕ	ВЫКЛЮЧЕНИЕ
жирный шрифт (*bold*)	**ESC G**	**ESC H**
утолщенный шрифт (*emphasized*)	**ESC E**	**ESC F**
курсив (*italics*)	**ESC 4**	**ESC 5**
подчеркивание (*underline*)	**ESC - 1**	**ESC - 0**
повышенное качество печати (*near letter quality*)	**ESC x 1** *nlq*	**ESC x 0** *draft*
верхние индексы (*superscript*)	**ESC S 0**	**ESC T**
нижние индексы (*subscript*)	**ESC S 1**	**ESC T**
сжатый шрифт (17 букв/дюйм) (*condensed*)	**'\017'**	**'\022'**
двойная ширина букв (*expanded*)	**ESC W 1**	**ESC W 0**
пропорциональная печать (*proportional spacing*)	**ESC p 1**	**ESC p 0**

Можно включить одновременно несколько из перечисленных выше режимов. В каждой из следующих двух групп надо выбрать одно из трех:

pitch (плотность печати)	
pica (10 букв/дюйм)	**ESC P**
elite (12 букв/дюйм)	**ESC M**
micron (15 букв/дюйм)	**ESC g**

font (шрифт)	
черновик (*draft* (*Roman*))	**ESC k '\0'**
текст (*text* (*Sans Serif*))	**ESC k '\1'**
курьер (*courier*)	**ESC k '\2'**

Всюду выше **0** означает либо '**0**' либо '**\0**'; **1** означает либо '**1**' либо '**\1**'. Пример:

```
printf( "This is \033Gboldface\033H word\n");
```

7.30. Составьте программу вывода набора файлов на печать, начинающую каждый очередной файл с новой страницы и печатающую перед каждым файлом заголовок и номер текущей страницы. Используйте символ '**\f**' (*form feed*) для перевода листа принтера.

7.31. Напишите программу печати текста в две колонки. Используйте буфер для формирования листа: файл читается построчно (слишком длинные строки обрубать), сначала заполняется левая половина листа (буфера), затем правая. Когда лист полностью заполнен или файл кончился - выдать лист построчно, расписать буфер пробелами (очистить лист) и повторить заполнение очередного листа. Указание: размеры листа должны передаваться как аргументы **main**(), для буфера используйте двумерный массив букв, память для него заказывайте динамически. Усложнение: не обрубайте, а переносите слишком длинные строки (строка может потребовать даже переноса с листа на лист).

```
/* ПРОГРАММА ПЕЧАТИ В ДВЕ ПОЛОСЫ: pr.c */
#include <stdio.h>
#include <string.h>
#define YES 1
#define NO  0
#define FORMFEED '\f'
#define LINEFEED '\n'

extern char *malloc(unsigned);
extern char *strchr(char *, char);
void untab(register char *s);
void resetsheet( void );
void addsheet( char *s, FILE *fpout );
void flushsheet( FILE *fpout );
void printline( int y, char *s, char *attr,
                FILE *fpout );
```

```
void doattr( register char *abuf,
             register char *vbuf );
void printcopy( FILE *fpin, FILE *fpout );
void main(void);
char *strdup (const char *s){
  char *p = malloc(strlen(s)+1); strcpy(p,s); return p;

  /* return strcpy((char *) malloc(strlen(s)+1), s); */
}

/* ... текст функции untab() ... */

int Sline;       /* строка на листе            */
int Shalf;       /* половина листа             */
int npage;       /* номер страницы             */
int startpage = 1;
            /* печать начиная с 1ой страницы   */
int fline;       /* номер строки файла         */
int topline = 0; /* смещение до начала листа   */
int halfwidth;   /* ширина полулиста           */
int twocolumns = YES;   /* в две колонки ?     */
int lshift, rshift = 1; /* поля слева и справа */
typedef unsigned short ushort;
int COLS  = 128; /* ширина листа (букв)        */
int LINES =  66; /* длина листа (строк)        */
ushort *mem;     /* буфер листа                */
#define AT(x,y)  mem[ (x) + (y) * COLS ]

/* Выделить буфер под лист и зачистить его */
void resetsheet ( void ){
   register x;
   if( mem == NULL ){ /* выделить память */
       if ((mem = (ushort *)
         malloc (COLS * LINES * sizeof(ushort)))
         == NULL ){
         fprintf(stderr, "Out of memory.\n"); exit(1);
       }
   }
   /* очистить */
   for( x= COLS * LINES - 1 ; x >= 0 ; x-- )
           mem[x] = ' ' & 0xFF;
   halfwidth = (twocolumns ? COLS/2 : COLS )
               - (lshift + rshift );
   Sline = topline; Shalf = 0;
}

#define NEXT_HALF \
   if( twocolumns == YES && Shalf == 0 ){         \
       /* закрыть данную половину листа */       \
       Shalf = 1; /* перейти к новой половине */ \
       Sline = topline;                           \
   } else                                         \
       flushsheet(fpout)   /* напечатать лист */

/* Записать строку в лист */
void addsheet ( char *s, FILE *fpout )
{
   register x, y;
   register i;
   char *rest = NULL;
   int wrap = NO;
/* YES когда идет перенос слишком длинной строки */

   /* в какое место поместить строку? */
   x = (Shalf == 0 ? 0 : COLS/2) + lshift;
   y = Sline;
```

```
    i = 0;      /* позиция в строке s */
    while (*s) {
        if( *s == '\f' ){
            /* вынужденный form feed */
            rest = strdup( s+1 ); /* остаток строки */
            NEXT_HALF;
            if( *rest ) addsheet(rest, fpout);
            free( rest );
            return;
        }
        if( i >= halfwidth ){
        /* перенести длинную строку */
            wrap = YES;
            rest = strdup(s);
            break;
        }
        /* Обработка выделений текста */
        if( s[1] == '\b' ){
            while( s[1] == '\b' ){
               AT(x, y) = (s[0] << 8) | (s[2] & 0xFF);
               /* overstrike */
               s += 2;
            }
            s++; x++; i++;
        } else {
            AT (x, y) = *s++ & 0xFF;
            x++; i++;
        }
    }
    /* Увеличить строку/половину_листа */
    Sline++;
    if (Sline == LINES) { /* полулист заполнен */
        NEXT_HALF;      }
    if( wrap && rest )  { /* дописать остаток строки */
        addsheet(rest, fpout); free(rest);
    }
}
int again;      /* нужна ли повторная надпечатка? */
/* Напечатать заполненный лист */
void flushsheet ( FILE *fpout ){
    register x, y, xlast;
    char *s, *p;
    static char outbuf[BUFSIZ], attr[BUFSIZ];
    /* attr - буфер под атрибуты выделений */
    ushort c;

    if( npage >= startpage )
      for (y = 0; y < LINES; y++) {
         /* обрезать концевые пробелы */
         for (xlast = (-1), x = COLS - 1; x >= 0; x--)
            if (AT (x, y) != ' ') { xlast = x; break; }
         again = NO; s = outbuf;  p = attr;
         for (x = 0; x <= xlast; x++){
            c = AT(x, y);
            *s++ = c & 0xFF;

            /* имеет атрибуты ? */
            c >>= 8; c &= 0xFF;
            *p++ = c ? c : ' ';
            if( c ) again = YES;
         }
         *s = '\0'; *p = '\0';
         printline(y, outbuf, attr, fpout);
      }
    npage++;        /* next page */
    resetsheet();   /* зачистить новый лист */
```

```
}

/* Напечатать одну строку листа */
void printline ( int y, char *s, char *attr,
                 FILE *fpout ){
   register x;
   if( again ){
       doattr(attr, s); fprintf(fpout, "%s\r", attr );
   }
   fprintf(fpout, "%s", s);
   /* перевод листа или строки */
   fputc( y == LINES-1 ? FORMFEED : LINEFEED, fpout );
}

/* Проверить - нет ли атрибутов выделений */
void doattr ( register char *abuf,
              register char *vbuf ){
   for(; *abuf; abuf++, vbuf++ )
      if( !strchr(" _-!|\177", *abuf))
          *abuf = *vbuf;
}

/* Копирование файла на принтер */
void printcopy ( FILE *fpin, FILE *fpout )
{
   char inbuf[BUFSIZ];

   npage = 1; /* первая страница имеет номер 1 */
   fline = 0; /* текущая строка файла - 0      */

   resetsheet();  /* зачистить буфер листа */
   while( fgets(inbuf, sizeof inbuf - 1, fpin )
           != NULL ){
           register l = strlen( inbuf );
           if( l && inbuf[l-1] == '\n' )
                    inbuf[--l] =  '\0' ;
           fline++;
           untab   ( inbuf );
           addsheet( inbuf, fpout );
   }
   if( !(Sline == topline && Shalf == 0))
   /* если страница не была только что зачищена ... */
         flushsheet(fpout);
   fprintf(stderr, "%d строк, %d листов.\n",
                    fline,    npage-1);
}
/* Вызов: pr < файл > /dev/lp */
void main (){ printcopy(stdin, stdout); }
```

Файл-принтер имеет в **UNIX** имя **/dev/lp** или подобное ему, а в **MS DOS** - имя **prn**.

7.32. Напишите программу, которая построчно считывает небольшой файл в память и печатает строки в обратном порядке. Указание: используйте динамическую память - функции **malloc**() и **strcpy**().

Объясним, почему желательно пользоваться динамической памятью. Пусть мы знаем, что строки имеют максимальную длину 80 символов и максимальное количество строк равно 50. Мы могли бы хранить текст в двумерном массиве:

```
char text[50][80];
```

занимающем 50*80 = 4000 байт памяти. Пусть теперь оказалось, что строки файла в действительности имеют длину по 10 букв. Мы

```
используем           50 * (10 + 1) =  550 байт
не используем 4000 - 50 * (10 + 1) = 3450 байт
```

(+1 нужен для символа '\0' на конце строки).

Пусть мы теперь пишем

```
char *text[50]; int i=0;
```

и при чтении очередной строки сохраняем ее так:

```
char buffer[81], *malloc(),  *gets();
while( gets(buffer) != NULL ){
  text[i] = (char *) malloc(strlen(buffer)+1);
  /* +1 для хранения \0, который не учтен strlen-ом */
  strcpy(text[i++], buffer);
}
```

то есть заказываем ровно столько памяти, сколько надо для хранения строки и ни байтом больше. Здесь мы (если sizeof(char *)==4) используем

```
              50 * 4 +  50 * (10 + 1 + 4) = 950 байт
массив указателей + заказанная malloc память
```

(+4 - служебная информация **malloc**), но зато у нас не остается неиспользуемой памяти. Преимуществом выделения памяти в виде массива является то, что эта память выделится ГАРАНТИРОВАННО, тогда как **malloc**()-у может не хватить памяти (если мы ее прежде очень много захватывали и не освобождали **free**()). Если **malloc** не может выделить участок памяти требуемого размера, он возвращает значение NULL:

```
if((text[i] = malloc(....)) == NULL)
{ fprintf(stderr, "Мало памяти\n"); break; }
```

Распечатка строк:

```
for(--i; i >= 0; i-- ){
        printf("%s\n", text[i]);
        free( text[i] );
}
```

Функция **free**(*ptr*) "освобождает"† отведенную ранее **malloc**()ом или **calloc**()ом область памяти по адресу *ptr* так, что при новых вызовах **malloc**() эта область может быть переиспользована. Данные в освобожденной памяти ПОРТЯТСЯ после **free**(). Ошибочно (и опасно) освобождать память, которая НЕ БЫЛА отведена **malloc**()-ом!

Организация текста в виде массива ссылок на строки или *списка* ссылок на строки, а не в виде двумерного текстового поля, выгодна еще тем, что такие строки проще переставлять, сортировать, вставлять строку в текст, удалять строку из текста. При этом переставляются лишь *указатели* в линейном массиве, а сами строки никуда не копируются. В двумерном же байтовом массиве нам пришлось бы для тех же перестановок копировать целые массивы байт - строки этой текстовой матрицы.

7.33. Напишите программу, печатающую строки файла в обратном порядке. Не считывать файл целиком в память! Следует использовать метод "обратного чтения" либо метод "быстрого доступа" к строкам файла, описанный в главе "Работа с файлами".

---

† На самом деле все освобожденные куски включаются в список свободной памяти, и склеиваются вместе, если два освобожденных куска оказались рядом. При новых вызовах **malloc** сначала просматривается список свободной памяти - нет ли там области достаточного размера? Этот алгоритм описан у Кернигана и Ритчи.

```
/* Инвертирование порядка строк в файле.
 * Используется та идея, что файл-результат имеет тот же
 * размер, что и исходный
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

#define BUFS 4096       /* максимальная длина строки */

void main(int argc, char **argv )
{
        FILE *fp;
        struct stat st;
        long len;
        char buffer[ BUFS+1 ];
        FILE *fpnew;      /* инверсный файл */
        int lgt;

        if( argc != 2 ){
                printf("Error: must be filename\n");
                exit(1);
        }
        if( (fp= fopen( argv[1], "r" )) == NULL ){
                printf( "Can not open %s\n", argv[1] );
                exit(2);
        }
        stat( argv[1], &st );   /* fstat(fileno(fp), &st); */
        len = st.st_size;       /* длина файла в байтах */

        if( (fpnew = fopen( "inv.out", "w" ))== NULL ){
                printf("Can not create file\n");
                exit(3);
        }

        while( fgets( buffer, sizeof buffer, fp ) != NULL ){
                lgt = strlen( buffer );
                fseek(fpnew, len - lgt , 0);
                /* Помните, что смещение у lseek и fseek -
                 * это число типа long, а не int.
                 * Поэтому лучше всегда писать
                 *      lseek(fd, (long) off, whence);
                 */
                len -= lgt;
                fprintf( fpnew, "%s", buffer );
                /* или лучше  fputs(buffer, fpnew); */
        }
        fclose( fp ); fclose( fpnew );
}
```

7.34. Напишите программу, которая читает файл, состоящий из "блоков" текста, разделенных пустыми строками. Размер "блока" ограничен. Программа готовит файл для печати на принтер так, чтобы ни один блок не разбивался на части:

```
-----------             -----------
|###### A |             |###### A | лист1
|####   A | превращать  |####   A |
|#####  A |     в       |#####  A |
|         |             |         |
|###### B |             |         |
-----------             -----------
|####   B |             |###### B | лист2
|         |             |####   B |
   ...                  |         |
```

то есть если блок не умещается на остатке листа, он должен быть перенесен на следующий лист. Блоки следует разделять одной пустой строкой (но первая строка листа не должна быть пустой!). Если блок

длиннее страницы - не переносите его.

```
/* Решение задачи о переносе блоков текста,
 * если они не умещаются на остатке листа */

#include <stdio.h>
#include <ctype.h>
extern  void *malloc(unsigned);
extern  int atoi(char *);
FILE    *fpin = stdin, *fpout = stdout;

/* Спасти строку в динамически выделенной памяти */
char *strdup (const char *s) {
    char   *ptr = (char *) malloc (strlen (s) + 1);
    if( ptr ) strcpy (ptr, s); return ptr;
}
int     page_length = 66;  /* длина страницы */
int     current_line;      /* текущая строка на странице (с нуля) */
int numbered = 0;          /* нумеровать строки листа ? */
#define MAXLINES 256       /* макс. длина блока  */
int     stored = 0;        /* запомнено строк    */
char   *lines[MAXLINES];   /* запомненные строки */

/* Запомнить строку блока в буфер строк */
void remember (char *s) {
    if (stored >= MAXLINES) {
        fprintf (stderr, "Слишком длинный блок.\n"); return;
    } else if((lines[stored++] = strdup (s)) == NULL ){
        fprintf (stderr, "Мало памяти (Out of memory).\n"); exit(13);
    }
}

/* Переход на следующую страницу */
void newpage () {
    current_line = 0; putc('\f', fpout);
}

/* Перевод строки или листа */
void newline (void) {
    if (current_line == page_length - 1)
        newpage ();             /* начать новый лист */
    else {
        current_line++;
        if( numbered ) fprintf(fpout, "%02d\n", current_line);
        else           putc ('\n', fpout);
    }
}

/* Переход на следующую страницу вставкой пустых строк */
void nextpage () {
    while (current_line != 0)
           newline ();
}

/* Выдать спасенный блок */
void throwout () {
    register    i;
    for (i = 0; i < stored; i++) {
        if( numbered )
             fprintf(fpout, "%02d %s", current_line, lines[i]);
        else fputs (lines[i], fpout);
        newline (); free (lines[i]);
    }
    stored = 0;
}
```

```
/* Выдать блок, перенося на следующий лист если надо */
void flush () {
    int     rest_of_page = page_length - current_line;
    /* осталось пустых строк на странице */

    if ((stored > page_length && rest_of_page < page_length / 4) ||
         rest_of_page < stored)
             nextpage ();
    throwout ();
    if (current_line)           /* не первая строка листа */
        newline ();             /* разделитель блоков */
}

/* Обработать входной файл */
void process () {
    char buffer[512];  int l;

    while (fgets (buffer, sizeof buffer, fpin) != NULL) {
        if ((l = strlen (buffer)) && buffer[l - 1] == '\n')
                                    buffer[  --l] =  '\0';
        if (l) remember (buffer);
    /* а по пустой строке - выдать блок */
        else if (stored) flush ();
    }
    if (stored) flush ();
    nextpage();
}

void main (int argc, char *argv[]) {
    argc--; argv++;
    while (*argv) {
        if (**argv == '-') {
            char   *key = *argv + 1, *arg;
            switch (*key) {
                case 'l':
                    if (! key[1]) {
                        if( argv[1] ){
                          arg = argv[1]; argv++; argc--;
                        } else arg = "";
                    } else arg = key+1;
                    if( isdigit(*arg) ){
                       page_length = atoi(arg);
       fprintf (stderr, "Длина страницы: %d строк\n", page_length);
                    } else fprintf(stderr, "-l ЧИСЛО\n");
                    break;
                case 'n':
                    numbered++; break;
                default:
                    fprintf (stderr, "Неизвестный ключ %s\n", key);
                    break;
            }
        }
        argv++; argc--;
    }
    process ();
    exit(0);
}
```

7.35. Составьте программу вывода строк файла в инверсном отображении, причем порядок символов в строках также следует инвертировать. Например,

```
abcdef ... oklmn                    987654321
   .....            превращать в        .....
123456789                           nmlko ... fedcba
```

Программа должна быть составлена двумя способами: при помощи обратного чтения файла и рекурсивным вызовом самой функции инвертирования. Указание: при обратном чтении надо читать файл большими

кусками (блоками).

7.36. Напишите программу, читающую файл построчно и размещающую строки в отсортированное двоичное дерево. По концу файла - распечатайте это дерево. Указание: используйте динамическую память и рекурсию.

```
/* Двоичная сортировка строк при помощи дерева */
#include <stdio.h>

char buf[240];          /* буфер ввода */
int lines;              /* номер строки файла */

typedef struct node{
        struct _data{   /* ДАННЫЕ */
           char *key;   /* ключ  - строка  */
           int  line;   /* номер строки    */
        } data;
                        /* СЛУЖЕБНАЯ ИНФОРМАЦИЯ */
        struct node *l, /* левое поддерево */
                    *r; /* правое поддерево */
} Node;
Node *root = NULL;      /* корень дерева (ссылка на верхний узел) */

/* Отведение памяти и инициализация нового узла */
Node *newNode(s)
  char *s;              /* строка */
{
        Node *tmp;
        extern char *malloc();    /* выделитель памяти */

        tmp = (Node *) malloc(sizeof(Node));
        if( tmp == NULL ){
                fprintf( stderr, "Нет памяти.\n");
                exit(1);
        }
        tmp -> l = tmp -> r = NULL;       /* нет поддеревьев */
        tmp -> data.line = lines;         /* номер строки файла */

        tmp -> data.key = malloc( strlen(s) + 1 );
             /* +1  - под байт '\0' в конце строки */
        strcpy(tmp -> data.key, s);       /* копируем ключ в узел */

        return tmp;
}

int i; /* Вынесено в статическую память, чтобы при каждом
        * рекурсивном вызове не создавалась новая auto-переменная,
        * а использовалась одна и та же статическая */
```

```
/* Рекурсивная печать дерева */
void printtree(root, tree, level, c)
  Node *root;                    /* корень дерева */
  Node *tree;                    /* дерево        */
  int level;                     /* уровень       */
  char c;                        /* имя поддерева */
{
        if( root == NULL ){ printf("Дерево пусто.\n"); return; }
        if( tree == NULL )  return;

        /* если есть - распечатать левое поддерево */
        printtree (root, tree -> l, level + 1, '/');  /* 'L' */

        /* распечатать ключ узла */
        for( i=0; i < level; i++ )
                printf("   ");
        printf("%c%3d--\"%s\"\n",
                 c, tree-> data.line, tree -> data.key);

        /* если есть - распечатать правое поддерево */
        printtree(root, tree -> r, level + 1, '\\');  /* 'R' */
}

void prTree(tree) Node *tree;
{
        printtree(tree, tree, 0, '*');
}

/* Добавить узел с ключом key в дерево tree  */
void addnode(tree, key)
  Node **tree;  /* в какое дерево добавлять: адрес переменной,
                 * содержащей ссылку на корневой узел */
  char *key;    /* ключ узла */
{
#define TREE (*tree)

        if( TREE == NULL ){  /* дерево пока пусто */
                TREE = newNode( key );
                return;
        }
        /* иначе есть хоть один узел   */
        if  ( strcmp (key, TREE -> data.key) < 0 )
        {
                /*  добавить в левое поддерево    */
                if ( TREE -> l == NULL ){
                        /* нет левого дерева  */
                        TREE -> l = newNode(key);
                        return;
                }
                else addnode( & TREE ->l , key);
        }

        else{
                /* добавить в правое дерево  */
                if ( TREE -> r == NULL ){
                        /* нет правого поддерева */
                        TREE -> r = newNode(key);
                        return;
                }
                else addnode ( & TREE ->r, key) ;
        }
}
```

```
/* Процедура удаления из дерева по ключу. */
typedef struct node *NodePtr;
static NodePtr delNode;    /* удаляемая вершина */

void delete(key, tree)
  char *key;          /* ключ удаляемого элемента */
  NodePtr *tree;      /* из какого дерева удалять */
{
     extern void doDelete();
     if(*tree == NULL){
          printf( "%s не найдено\n", key ); return;
     }
     /* поиск ключа */
     else if(strcmp(key, (*tree)->data.key) < 0)
          delete( key, &(*tree)->l );
     else if(strcmp(key, (*tree)->data.key) > 0)
          delete( key, &(*tree)->r );
     else{  /* ключ найден */
          delNode = *tree;  /* указатель на удаляемый узел */
          if(delNode->r == NULL)      *tree = delNode->l;
          else if(delNode->l == NULL) *tree = delNode->r;
          else doDelete( & delNode->l );
          free(delNode);
     }
}

static void doDelete(rt) NodePtr *rt;
{
        if( (*rt)->r != NULL )    /* спуск по правой ветви */
            doDelete( &(*rt)->r );
        else{
            /* перенос данных в другой узел */
            delNode->data = (*rt)->data;

            delNode = *rt;         /* для free() */
            *rt = (*rt)->l;
        }
}

void main(){
        extern char *gets(); char *s;
        while (gets(buf) != NULL){      /* пока не конец файла */
                lines++;
                addnode( & root, buf );
        }
        prTree(root);

        /* удалим строку */
        freopen("/dev/tty", "r", stdin);
        do{
            printf( "что удалить ? " );
            if((s = gets(buf)) == NULL) break;
            delete(buf, &root);
            prTree( root );
        } while( s && root );

        printf("Bye-bye.\n");
        exit(0);
}
```

7.37. Напишите программу, которая читает со стандартного ввода 10 чисел *либо* слов, а затем распечатывает их. Для хранения введенных данных используйте объединение.

```
#include <stdio.h>
#include <ctype.h>
#define INT 'i'
```

```
#define STR 's'
struct data {
  char tag;  /* тэг, пометка. Код типа данных. */
  union {
        int i;
        char *s;
  } value;
} a[10];
int counter = 0; /* счетчик */
void main(){
  char word[128]; int i; char *malloc(unsigned);

  /* Чтение: */
  for(counter=0; counter < 10; counter++){
    if( gets(word) == NULL ) break;
    if( isdigit((unsigned char) *word)){
        a[counter].value.i = atoi(word);
        a[counter].tag     = INT;
    } else {
        a[counter].value.s = malloc(strlen(word)+1);
        strcpy(a[counter].value.s, word);
        a[counter].tag     = STR;
    }
  }
  /* Распечатка: */
  for(i=0; i < counter; i++)
    switch(a[i].tag){
    case INT: printf("число %d\n", a[i].value.i);
              break;
    case STR: printf("слово %s\n", a[i].value.s);
              free(a[i].value.s);
              break;
   }
}
```

7.38. Рассмотрим задачу написания функции, которая обрабатывает переменное число аргументов, например функцию-генератор меню. В такую функцию надо подавать строки меню и адреса функций, вызываемых при выборе каждой из строк. Собственно проблема, которую мы тут обсуждаем - как передавать переменное число аргументов в подобные функции? Мы приведем три программы использующие три различных подхода. Предпочтение не отдано ни одному из них - каждый из них может оказаться эффективнее других в определенных ситуациях. Думайте сами!

7.38.1. Массив

```
/* Передача аргументов в функцию как МАССИВА.
 * Следует явно указать число аргументов в массиве.
 */

#include <stdio.h>              /* printf(), NULL */
#include <string.h>             /* strdup() */
#include <stdlib.h>             /* malloc() */

#define A_INT           1
#define A_STR           2
#define A_NULL          0

typedef struct arg {
        int type;
        union jack {
                char *s;
                int   d;
        } data;
        struct arg *next;
} Arg;
```

```
void doit(Arg args[], int n){
        int i;

        for(i=0; i < n; i++)
                switch(args[i].type){
                case A_INT:
                        printf("%d", args[i].data.d);
                        break;
                case A_STR:
                        printf("%s", args[i].data.s);
                        break;
                default:
                        fprintf(stderr, "Unknown type!\n");
                        break;
                }
}

/* При инициализации union надо использовать тип
 * первого из перечисленных значений.
 */
Arg sample[] = {
        { A_INT, (char *) 123 },
        { A_STR, (char *) " hello, " },
        { A_INT, (char *) 456 },
        { A_STR, (char *) " world\n" }
};

int main(int ac, char *av[]){
        doit(sample, sizeof sample / sizeof sample[0]);
        return 0;
}
```

7.38.2. Список

```
/* Передача аргументов в функцию как СПИСКА.
 * Достоинство: список можно модифицировать
 * во время выполнения программы: добавлять и
 * удалять элементы. Недостаток тот же: список надо
 * построить динамически во время выполнения,
 * заранее этого сделать нельзя.
 * Недостатком данной программы является также то,
 * что список не уничтожается после использования.
 * В C++ эта проблема решается при помощи использования
 * автоматически вызываемых деструкторов.
 */

#include <stdio.h>              /* printf(), NULL */
#include <string.h>             /* strdup() */
#include <stdlib.h>             /* malloc() */

#define A_INT           1
#define A_STR           2
#define A_NULL          0

typedef struct arg {
        int type;
        union jack {
                char *s;
                int   d;
        } data;
        struct arg *next;
} Arg;
```

```
void doit(Arg *arglist){
        for( ; arglist; arglist=arglist->next)
                switch(arglist->type){
                case A_INT:
                        printf("%d", arglist->data.d);
                        break;
                case A_STR:
                        printf("%s", arglist->data.s);
                        break;
                default:
                        fprintf(stderr, "Unknown type!\n");
                        break;
                }
}

Arg *new_int(int n, Arg *next){
        Arg *ptr = (Arg *) malloc(sizeof(Arg));
        ptr->type   = A_INT;
        ptr->data.d = n;
        ptr->next   = next;
        return ptr;
}

Arg *new_str(char *s, Arg *next){
        Arg *ptr = (Arg *) malloc(sizeof(Arg));
        ptr->type   = A_STR;
        ptr->data.s = strdup(s);
        ptr->next   = next;
        return ptr;
}

int main(int ac, char *av[]){
        doit(
                new_int(123,
                new_str(" hello, ",
                new_int(456,
                new_str(" world\n",
                        NULL))))
        );
        return 0;
}
```

7.38.3. Функция с переменным числом параметров

```
/* Передача аргументов в функцию как СПИСКА АРГУМЕНТОВ
 * ФУНКЦИИ с признаком конца списка.
 */

#include <stdio.h>              /* printf(), NULL */
#include <stdarg.h>             /* va_...   */

#define A_INT           1
#define A_STR           2
#define A_NULL          0
```

```
void doit(...){         /* переменное число аргументов */
        va_list args;

        /* второй параметр - аргумент, предшествующий ...
         * Если такого нет - ставим запятую и пустое место!
         */
        va_start(args, );

        for(;;){
                switch(va_arg(args, int)){
                case A_INT:
                        printf("%d", va_arg(args, int));
                        break;
                case A_STR:
                        printf("%s", va_arg(args, char *));
                        break;
                case A_NULL:
                        goto breakloop;
                default:
                        fprintf(stderr, "Unknown type!\n");
                        break;
                }
        }
breakloop:
        va_end(args);
}

int main(int ac, char *av[]){
        doit(
                A_INT, 123,
                A_STR, " hello, ",
                A_INT, 456,
                A_STR, " world\n",
                A_NULL
        );
        return 0;
}
```

7.39. Напишите несколько функций для работы с упрощенной базой данных. Запись в базе данных содержит ключ - целое, и строку фиксированной длины:

```
struct data {
       int  b_key;              /* ключ */
       char b_data[ DATALEN ]; /* информация */
};
```

Напишите:

- добавление записи
- уничтожение по ключу
- поиск по ключу (и печать строки)
- обновление по ключу.

Файл организован как несортированный массив записей без дубликатов (т.е. ключи не могут повторяться). Поиск производить линейно. Используйте функции **fread**, **fwrite**, **fseek**. Последняя функция позволяет вам позиционироваться к *n*-ой записи файла:

```
fseek( fp, (long) n * sizeof(struct data), 0 );
```

Перепишите эту программу, объявив ключ как строку, например

```
char b_key[ KEYLEN ];
```

Если строка-ключ короче KEYLEN символов, она должна оканчиваться '\0', иначе - используются все KEYLEN букв и '\0' на конце отсутствует (так же устроено поле *d_name* в каталогах файловой системы). Усовершенствуйте алгоритм доступа, используя хеширование по ключу (*hash* - перемешивание, см. пример в приложении). Вынесите ключи в отдельный файл. Этот файл ключей состоит из структур

```
struct record_header {
  int  b_key   ;    /* ключ */
  long b_offset;    /* адрес записи в файле данных */
  int  b_length;    /* длина записи (необязательно) */
};
```

то есть организован аналогично нашей первой базе данных. Сначала вы ищете нужный ключ в файле ключей. Поле *b_offset* у найденного ключа задает адрес данного в другом файле. Чтобы прочитать его, надо сделать **fseek** на расстояние *b_offset* в файле данных и прочесть *b_length* байт.

7.40. Организуйте базу данных в файле как *список* записей. В каждой записи вместо ключа должен храниться *номер* очередной записи (ссылка). Напишите функции: поиска данных в списке (по значению), добавления данных в список в алфавитном порядке, (они просто приписываются к концу файла, но в нужных местах переставляются ссылки), распечатки списка в порядке ссылок, удалению элементов из списка (из самого файла они не удаляются!). Ссылка (номер) первой записи (головы списка) хранится в первых двух байтах файла, рассматриваемых как **short**.

Введите оптимизацию: напишите функцию для сортировки файла (превращению перемешанного списка в линейный) и вычеркивания из него удаленных записей. При этом файл будет перезаписан. Если файл отсортирован, то поиск в нем можно производить более эффективно, чем прослеживание цепочки ссылок: просто линейным просмотром. Третий байт файла используйте как признак: 1 - файл был отсортирован, 0 - после сортировки в него было что-то добавлено и линейный порядок нарушен.

7.41. Напишите функцию **match**(*строка*,*шаблон*); для проверки соответствия строки упрощенному регулярному выражению в стиле Шелл. Метасимволы шаблона:

```
* - любое число любых символов (0 и более);
? - один любой символ.
              Усложнение:
[буквы]  - любая из перечисленных букв.
[!буквы] - любая из букв, кроме перечисленных.
[h-z]    - любая из букв от h до z включительно.
```

Указание: для проверки "остатка" строки используйте рекурсивный вызов этой же функции.

Используя эту функцию, напишите программу, которая выделяет из файла СЛОВА, удовлетворяющие заданному шаблону (например, "[*Ии]*о*т*"). Имеется в виду, что каждую строку надо сначала разбить на слова, а потом проверить каждое слово.

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#define  U(c) ((c) & 0377)  /* подавление расширения знака */
#define  QUOT    '\\'       /* экранирующий символ         */
#ifndef  MATCH_ERR
# define MATCH_ERR printf("Нет ]\n")
#endif

/* s - сопоставляемая строка
 * p - шаблон. Символ \ отменяет спецзначение метасимвола.
 */
int match (register char *s, register char *p)
{
    register int    scc; /* текущий символ строки                */
    int     c, cc, lc;   /* lc - предыдущий символ в [...] списке */
    int     ok, notflag;

    for (;;) {
        scc = U(*s++);          /* очередной символ строки  */
        switch (c = U (*p++)) { /* очередной символ шаблона */

            case QUOT:          /* a*\*b */
                c = U (*p++);
                if( c == 0 ) return(0); /* ошибка: pattern\ */
                else goto def;

            case '[':   /* любой символ из списка */
                ok = notflag = 0;
                lc = 077777;      /* достаточно большое число */
                if(*p == '!'){ notflag=1; p++; }

                while (cc = U (*p++)) {
                    if (cc == ']') {    /* конец перечисления */
                        if (ok)
                            break;      /* сопоставилось      */
                        return (0);     /* не сопоставилось   */
                    }
                    if (cc == '-') {    /* интервал символов  */
                        if (notflag){
                            /* не из диапазона - OK */
                            if (!syinsy (lc, scc, U (*p++)))
                                ok++;
                            /* из диапазона - неудача */
                            else return (0);
                        } else {
                            /* символ из диапазона - OK */
                            if (syinsy (lc, scc, U (*p++)))
                                ok++;
                        }
                    }
                    else {
                        if (cc == QUOT){      /* [\[\]] */
                            cc = U(*p++);
                            if(!cc) return(0);/* ошибка */
                        }
                        if (notflag){
                            if (scc && scc != (lc = cc))
                                ok++;          /* не входит в список */
                            else return (0);
                        } else {
                            if (scc == (lc = cc)) /* входит в список */
                                ok++;
                        }
                    }
                }
```

```
                    if (cc == 0){    /* конец строки */
                        MATCH_ERR;
                        return (0);         /* ошибка */
                    }
                    continue;

                case '*':   /* любое число любых символов */
                    if (!*p)
                        return (1);
                    for (s--; *s; s++)
                        if (match (s, p))
                            return (1);
                    return (0);

                case 0:
                    return (scc == 0);

                default: def:
                    if (c != scc)
                        return (0);
                    continue;

                case '?':   /* один любой символ */
                    if (scc == 0)
                        return (0);
                    continue;
            }
        }
    }

    /* Проверить, что smy лежит между smax и smin
     */
    int syinsy (unsigned smin, unsigned smy, unsigned smax)
    {
        char left   [2];
        char right  [2];
        char middle [2];

        left  [0]   = smin;  left  [1]   = '\0';
        right [0]   = smax;  right [1]   = '\0';
        middle[0]   = smy;   middle[1]   = '\0';

        return (strcoll(left, middle) <= 0 && strcoll(middle, right) <= 0);
    }
```

Обратите внимание на то, что в **UNIX** расширением шаблонов имен файлов, вроде *.c, занимается не операционная система (как в **MS DOS**), а программа-интерпретатор команд пользователя (shell: **/bin/sh**, **/bin/csh**, **/bin/ksh**). Это позволяет обрабатывать (в принципе) разные стили шаблонов имен.

7.42. Изучите раздел руководства **man regexp** и include-файл */usr/include/regexp.h*, содержащий исходные тексты функций **compile** и **step** для регулярного выражения в стиле программ **ed**, **lex**, **grep**:

одна буква *C*
: или заэкранированный спецсимвол **\.** \[***** **\$** \^ \\ означают сами себя;

. означает один любой символ кроме \n;

[*abc*] или [*a*-*b*] означает любой символ из перечисленных (из интервала);

[*abc*-] минус в конце означает сам символ **-**;

[]*abc*] внутри [] скобка] на первом месте означает сама себя;

[^*a*-*z*] крышка ^ означает отрицание, т.е. любой символ *кроме* перечисленных;

[*a*-*z*^] крышка не на первом месте означает сама себя;

[*.] спецсимволы внутри [] не несут специального значения, а представляют сами себя;

*C** любое (0 и более) число символов *C*;

.* любое число любых символов;

*выражение******
любое число (0 и более) повторений выражения, например [0-9]* означает число (последовательность цифр) или пустое место. Ищется самое длинное прижатое *влево* подвыражение;

*выражение***\{***n*,*m***\}**
повторение выражения от *n* до *m* раз (включительно), где числа не превосходят 255;

*выражение***\{***n*,**\}**
повторение по крайней мере *n* раз, например [0-9]\{1,\} означает число;

*выражение***\{***n***\}**
повторение ровно *n* раз;

*выражение***$**
строка, чей *конец* удовлетворяет выражению, например .*define.*\\$

^*выражение*
строка, чье *начало* удовлетворяет выражению;

\n символ перевода строки;

\(.....**\)**
сегмент. Сопоставившаяся с ним подстрока будет запомнена;

N где *N* цифра. Данный участок образца должен совпадать с *N*-ым сегментом (нумерация с 1).

Напишите функцию **matchReg**, использующую этот стиль регулярных выражений. Сохраняйте шаблон, при вызове **matchReg** сравнивайте старый шаблон с новым. Перекомпиляцию следует производить только если шаблон изменился:

```
#include <stdio.h>
#include <ctype.h>
#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(ptr)     return
#define ERROR(code)  \
{fprintf(stderr,"%s:ERR%d\n",instring,code);exit(177);}

#               include <regexp.h>

#define EOL             '\0'    /* end of line */
#define ESIZE           512

int matchReg(char *str, char *pattern){
  static char oldPattern[256];
  static char compiledExpr[ESIZE];
  if( strcmp(pattern, oldPattern)){  /* различны */
    /* compile regular expression */
      compile(pattern,
              compiledExpr, &compiledExpr[ESIZE], EOL);
      strcpy(oldPattern, pattern);   /* запомнить   */
  }
  return step(str, compiledExpr);    /* сопоставить */
}
/* Пример вызова: reg '^int' 'int$' char | less */
/* reg 'putchar.*(.*)' < reg.c | more */

void main(int ac, char **av){
  char inputline[BUFSIZ]; register i;

  while(gets(inputline)){
    for(i=1; i < ac; i++)
      if(matchReg(inputline, av[i])){

  char *p; extern char *loc1, *loc2;
/*printf("%s\n", inputline);*/
/* Напечатать строку,
```

```
 * выделяя сопоставившуюся часть жирно */
  for(p=inputline; p != loc1; p++) putchar(*p);
  for(           ; p != loc2; p++)
     if(isspace((unsigned char) *p))
          putchar(*p);
     else printf("%c\b%c", *p, *p);
  for(           ; *p;        p++) putchar(*p);
  putchar('\n');
  break;
      }
    }
}
```

7.43. Используя <*regexp.h*> напишите программу, производящую контекстную замену во всех строках файла. Если строка не удовлетворяет регулярному выражению - она остается неизменной. Примеры вызова:

```
$  regsub '\([0-9]\{1,\}\)'   '(\1)'
$  regsub 'f(\(.*\),\(.*\))'  'f(\2,\1)'  < file
```

Вторая команда должна заменять все вхождения f(a,b) на f(b,a). Выражение, обозначенное в образце как **\(**...**\)**, подставляется на место соответствующей конструкции *N* во втором аргументе, где *N* - цифра, номер сегмента. Чтобы поместить в выход сам символ \, его надо удваивать: \\.

```
/* Контекстная замена */
#include <stdio.h>
#include <ctype.h>

#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(ptr)     return
#define ERROR(code)     regerr(code)
void regerr();
#               include <regexp.h>
#define EOL             '\0'    /* end of line */
#define ESIZE           512
short all = 0;
/* ключ -a означает, что в строке надо заменить ВСЕ вхождения образца (global, all):
 *          regsub -a   int INT
 *          "aa int bbb int cccc" -> "aa INT bbb INT cccc"
 *
 * step() находит САМУЮ ДЛИННУЮ подстроку, удовлетворяющую выражению,
 * поэтому  regsub 'f(\(.*\),\(.*\))'  'f(\2,\1)'
 * заменит  "aa f(1,2) bb f(3,4) cc" -> "aa f(4,1,2) bb f(3) cc'
 *               |___________|_|             |_|___________|
 */
char compiled[ESIZE], line[512];
```

```
void main(int ac, char *av[]){
   register char *s, *p; register n;  extern int nbra;
   extern char *braslist[], *braelist[], *loc1, *loc2;

   if( ac > 1 && !strcmp(av[1], "-a")){ ac--; av++; all++; }
   if(ac != 3){
        fprintf(stderr, "Usage: %s [-a] pattern subst\n", av[0]);
        exit(1);
   }
   compile(av[1], compiled, compiled + sizeof compiled, EOL);

   while( gets(line) != NULL ){
       if( !step(s = line, compiled)){
           printf("%s\n", line); continue;
       }
       do{
         /* Печатаем начало строки */
         for( ; s != loc1; s++) putchar(*s);

         /* Делаем замену */
         for(s=av[2]; *s; s++)
           if(*s == '\\'){
             if(isdigit(s[1])){ /* сегмент */
                int num = *++s - '1';
                if(num < 0 || num >= nbra){
                   fprintf(stderr, "Bad block number %d\n", num+1);
                   exit(2);
                }
                for(p=braslist[num]; p != braelist[num]; ++p)
                    putchar(*p);
             } else if(s[1] == '&'){
                ++s;  /* вся сопоставленная строка */
                for(p=loc1; p != loc2; ++p)
                    putchar(*p);
             } else putchar(*++s);
           } else putchar(*s);

       } while(all && step(s = loc2, compiled));

       /* Остаток строки */
       for(s=loc2; *s; s++) putchar(*s);
       putchar('\n');
   } /* endwhile */
}

void regerr(int code){ char *msg;
   switch(code){
   case 11: msg = "Range endpoint too large.";     break;
   case 16: msg = "Bad number.";                   break;
   case 25: msg = "\\digit out of range.";         break;
   case 36: msg = "Illegal or missing delimiter."; break;
   case 41: msg = "No remembered search string.";  break;
   case 42: msg = "\\(~\\) imbalance.";            break;
   case 43: msg = "Too many \\(.";                 break;
   case 44: msg = "More than 2 numbers given in \\{~\\\"}."; break;
   case 45: msg = "} expected after \\.";          break;
   case 46: msg = "First number exceeds second in \\{~\\}."; break;
   case 49: msg = "[ ] imbalance.";                break;
   case 50: msg = "Regular expression overflow.";  break;
   default: msg = "Unknown error";                 break;
   } fputs(msg, stderr); fputc('\n', stderr); exit(code);
}
```

```
void prfields(){
        int i;
        for(i=0; i < nbra; i++)
                prfield(i);
}
void prfield(int n){
        char *fbeg = braslist[n], *fend = braelist[n];
        printf("\\%d='", n+1);
        for(; fbeg != fend; fbeg++)
                putchar(*fbeg);
        printf("'\n");
}
```

7.44. Составьте функцию поиска подстроки в строке. Используя ее, напишите программу поиска подстроки в текстовом файле. Программа должна выводить строки (либо номера строк) файла, в которых встретилась данная подстрока. Подстрока задается в качестве аргумента функции **main**().

```
/* Алгоритм быстрого поиска подстроки.
 * Дж. Мур, Р. Бойер, 1976 Texas
 * Смотри: Communications of the ACM 20, 10 (Oct., 1977), 762-772
 *
 * Этот алгоритм выгоден при многократном поиске образца в
 * большом количестве строк, причем если они равной длины -
 * можно сэкономить еще и на операции strlen(str).
 * Алгоритм характерен тем, что при неудаче производит сдвиг не на
 * один, а сразу на несколько символов вправо.
 * В лучшем случае алгоритм делает slen/plen сравнений.
 */

char *pattern;          /* образец (что искать) */
static int plen;        /* длина образца */
static int d[256];      /* таблица сдвигов; в алфавите ASCII -
                         * 256 букв. */

/* расстояние от конца образца до позиции i в нем */
#define DISTANCE(i)     ((plen-1) - (i))
```

```
/* Поиск:
 * выдать индекс вхождения pattern в str,
 * либо -1, если не входит
 */
int indexBM( str )  char *str;      /* в чем искать */
{
    int slen = strlen(str); /* длина строки */
    register int pindx;  /* индекс сравниваемой буквы в образце */
    register int cmppos; /* индекс сравниваемой буквы в строке  */
    register int endpos; /* позиция в строке, к которой "приставляется"
                          * последняя буква образца */

    /* пока образец помещается в остаток строки */
    for( endpos = plen-1; endpos < slen ; ){

            /* Для отладки: pr(str, pattern, endpos - (plen-1), 0); /**/

            /* просмотр образца от конца к началу */
            for( cmppos = endpos, pindx = (plen - 1);
                                  pindx >= 0 ;
                                  cmppos--, pindx-- )

               if( str[cmppos] != pattern[pindx] ){
                  /* Сдвиг, который ставит самый правый в образце
                   * символ str[endpos] как раз под endpos-тую
                   * позицию строки. Если же такой символ в образце не
                   * содержится (или содержится только на конце),
                   * то начало образца устанавливается в endpos+1 ую
                   * позицию
                   */
                  endpos += d[ str[endpos] & 0377 ];
                  break;    /* & 0377 подавляет расширение знака. Еще  */
               }            /* можно сделать все char -> unsigned char */

            if( pindx < 0 ) return ( endpos - (plen-1));
            /* Нашел: весь образец вложился */
    }
    return( -1 );       /* Не найдено */
}

/* Разметка таблицы сдвигов */
void compilePatternBM( ptrn ) char *ptrn; {
        register int c;

        pattern = ptrn; plen = strlen(ptrn);

        /* c - номер буквы алфавита */
        for(c = 0; c < 256; c++)
                d[c] = plen;
                /* сдвиг на длину всего образца */

        /* c - позиция в образце */
        for(c = 0; c < plen - 1; c++)
                d[ pattern[c] & 0377 ] = DISTANCE(c);
        /* Сдвиг равен расстоянию от самого правого
         * (кроме последней буквы образца)
         * вхождения буквы в образец до конца образца.
         * Заметим, что если буква входит в образец несколько раз,
         * то цикл учитывает последнее (самое правое) вхождение.
         */
}
```

```
/* Печать найденных строк */
void pr(s, p, n, nl) char *s, *p;
{
        register i;

        printf("%4d\t%s\n", nl, s );
        printf("    \t");
        for(i = 0; i < n; i++ )
                putchar( s[i] == '\t' ? '\t' : ' ' );
        printf( "%s\n", p );
}

/* Аналог программы fgrep */
#include <stdio.h>
char str[ 1024 ];        /* буфер для прочитанной строки */

void main(ac, av) char **av;
{
        int nline = 0;  /* номер строки файла */
        int ind;
        int retcode = 1;

        if(ac != 2){
                fprintf(stderr, "Usage: %s 'pattern'\n", av[0] );
                exit(33);
        }
        compilePatternBM( av[1] );
        while( gets(str) != NULL ){
                nline++;
                if((ind = indexBM(str)) >= 0 ){
                        retcode = 0;    /* O'KAY */
                        pr(str, pattern, ind, nline);
                }
        }
        exit(retcode);
}

/* Пример работы алгоритма:
        peter piper picked a peck of pickled peppers.
        peck
        peter piper picked a peck of pickled peppers.
          peck
        peter piper picked a peck of pickled peppers.
              peck
        peter piper picked a peck of pickled peppers.
                peck
        peter piper picked a peck of pickled peppers.
                     peck
        peter piper picked a peck of pickled peppers.
                         peck
        peter piper picked a peck of pickled peppers.
                               peck
        peter piper picked a peck of pickled peppers.
                                 peck
*/
```

7.45. Напишите аналогичную программу, выдающую все строки, удовлетворяющие упрощенному регулярному выражению, задаваемому как аргумент для **main**(). Используйте функцию **match**, написанную нами ранее. Вы написали аналог программы **grep** из **UNIX** (но с другим типом регулярного выражения, нежели в оригинале).

7.46. Составьте функцию **expand**(*s1*, *s2*), которая расширяет сокращенные обозначения вида *a-z* строки *s1* в эквивалентный полный список *abcd...xyz* в строке *s2*. Допускаются сокращения для строчных и прописных букв и цифр. Учтите случаи типа *a–b–c*, *a–z0–9* и *–a–g* (соглашение состоит в том, что символ "–", стоящий в начале или в конце, воспринимается буквально).

7.47. Напишите программу, читающую файл и заменяющую строки вида

```
|<1 и более пробелов и табуляций><текст>
```

на пары строк

```
|.pp
|<текст>
```

(здесь | обозначает левый край файла, а <> - метасимволы). Это - простейший препроцессор, готовящий текст в формате **nroff** (это форматтер текстов в **UNIX**). Усложнения:

- строки, начинающиеся с точки или с апострофа, заменять на

```
\&<текст, начинающийся с точки или '>
```

- строки, начинающиеся с цифры, заменять на

```
.ip <число>
<текст>
```

- символ \ заменять на последовательность **\e**.
- удалять пробелы перед символами **.,;:!?)** и вставлять после них пробел (знак препинания должен быть приклеен к концу слова, иначе он может быть перенесен на следующую строку. Вы когда-нибудь видели строку, *начинающуюся* с запятой?).
- склеивать перенесенные слова, поскольку **nroff** делает переносы сам:

```
....xxxx начало-  =>  ....xxxx началоконец
конец yyyy.....       yyyy.............
```

Вызывайте этот препроцессор разметки текста так:

```
$   prep файлы... | nroff -me > text.lp
```

7.48. Составьте программу преобразования прописных букв из файла ввода в строчные, используя при этом функцию, в которой необходимо организовать анализ символа (действительно ли это буква). Строчные буквы выдавать без изменения. Указание: используйте макросы из *<ctype.h>*.

Ответ:

```
#include <ctype.h>
#include <stdio.h>
main(){
  int c;
  while( (c = getchar()) != EOF )
    putchar( isalpha( c ) ?
            (isupper( c ) ? tolower( c ) : c) : c);
}
```

```
              либо ...
putchar( isalpha(c) && isupper(c) ? tolower(c) : c );
              либо даже
putchar( isupper(c) ? tolower(c) : c );
```

В последнем случае под **isupper** и **islower** должны пониматься только буквы (увы, не во всех реализациях это так!).

7.49. Обратите внимание, что если мы выделяем класс символов при помощи сравнения, например:

```
char ch;
if( 0300 <= ch && ch < 0340 ) ...;
```

(в кодировке **КОИ-8** это маленькие русские буквы), то мы можем натолкнуться на следующий сюрприз: перед сравнением с целым значение *ch* приводится к типу **int** (приведение также делается при использовании **char** в качестве аргумента функции). При этом, если у *ch* был установлен старший бит (0200), произойдет расширение его во весь старший байт (расширение знакового бита). Результатом будет *отрицательное* целое число! Опыт:

```
char c = '\201';    /* = 129 */
printf( "%d\n", c );
```

печатается -127. Таким образом, наше сравнение не сработает, т.к. оказывается что *ch* < 0. Следует подавлять расширение знака:

```
if( 0300 <= (ch & 0377) && (ch & 0377) < 0340) ...;
```

(0377 - маска из 8 бит, она же 0xFF, весь байт), либо объявить

```
unsigned char ch;
```

что означает, что при приведении к **int** знаковый бит не расширяется.

7.50. Рассмотрим еще один пример:

```
main(){
   char ch;
   /* 0377 - код последнего символа алфавита ASCII */
   for (ch = 0100; ch <= 0377; ch++ )
        printf( "%03o %s\n",
          ch & 0377,
          ch >= 0300 && ch < 0340 ? "yes" : "no" );
}
```

Какие неприятности ждут нас здесь?

- во-первых, когда бит 0200 у *ch* установлен, в сравнении *ch* выступает как *отрицательное* целое число (т.к. приведение к **int** делается расширением знакового бита), то есть у нас всегда печатается "no". Это мы можем исправить, написав **unsigned char** *ch*, либо используя *ch* в виде

  `(ch & 0377)` или `((unsigned) ch)`

- во-вторых, рассмотрим сам цикл. Пусть сейчас *ch* =='\377'. Условие *ch* <= 0377 истинно. Выполняется оператор *ch*++. Но *ch* - это байт, поэтому операции над ним производятся по модулю 0400 (0377 - это максимальное значение, которое можно хранить в байте - все биты единицы). То есть теперь значением *ch* станет 0. Но 0 < 0377 и условие цикла верно! Цикл продолжается; т.е. происходит зацикливание. Избежать этого можно только описав **int** *ch*; чтобы 0377+1 было равно 0400, а не 0 (или **unsigned int**, лишь бы длины переменной хватало, чтобы вместить число больше 0377).

7.51. Составьте программу, преобразующую текст, состоящий только из строчных букв в текст, состоящий из прописных и строчных букв. Первая буква и буква после каждой точки - прописные, остальные - строчные.

```
слово один. слово два. -->
Слово один. Слово два.
```

Эта программа может оказаться полезной для преобразования текста, набранного в одном регистре, в текст, содержащий буквы обоих регистров.

7.52. Напишите программу, исправляющую опечатки в словах (*spell check*): программе задан список слов; она проверяет - является ли введенное вами слово словом из списка. Если нет - пытается найти наиболее похожее слово из списка, причем если есть несколько похожих - выдает все варианты. Отлавливайте случаи:

- две соседние буквы переставлены местами: *ножинцы=>ножницы*;
- удвоенная буква (буквы): *ккарандаш=>карандаш*;
- потеряна буква: *бот=>болт*;
- измененная буква: *бинт=>бант*;
- лишняя буква: *морда=>мода*;
- буквы не в том регистре - сравните с каждым словом из списка, приводя все буквы к маленьким: *сОВОк=>совок*;

Надо проверять каждую букву слова. Возможно вам будет удобно использовать рекурсию. Подсказка: для некоторых проверок вам может помочь функция **match**:

```
слово_таблицы = "дом";
if(strlen(входное_слово) <= strlen(слово_таблицы)+1 &&
match(входное_слово, "*д*о*м*") ... /* похоже */
      *о*м*          ?дом         дом?
      *д*м*          д?ом
      *д*о*          до?м
```

Приведем вариант решения этой задачи:

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

typedef unsigned char uchar;
#define ANYCHAR '*'
/* символ, сопоставляющийся с одной любой буквой */

static uchar version[120];      /* буфер для генерации вариантов */
static uchar vv;                /* буква, сопоставившаяся с ANYCHAR */

/* привести все буквы к одному регистру */
static uchar icase(uchar c){
        return isupper(c) ? tolower(c) : c;
}

/* сравнение строк с игнорированием регистра */
static int eqi(uchar *s1, uchar *s2 )
{
        while( *s1 && *s2 ){
                if( icase( *s1 ) != icase( *s2 ))
                        break;
                s1++; s2++;
        }
        return ( ! *s1 && ! *s2 ) ? 1 : 0 ;
                                /* OK : FAIL */
}

/* сравнение строк с игнорированием ANYCHAR */
static strok(register uchar *word, register uchar *pat)
{
        while( *word && *pat ){
                if( *word == ANYCHAR){
                        /* Неважно, что есть *pat, но запомним */
                        vv= *pat;
                } else {
                        if( icase(*pat) != icase(*word) )
                            break;
                }
                word++; pat++;
        }
        /* если слова кончились одновременно ... */
        return ( !*word && !*pat) ? 1 : 0;
                                /* OK : FAIL */
}

/* ЛИШНЯЯ БУКВА */
static int superfluous( uchar *word /* слово для коррекции */
                      , uchar *s    /* эталон */
){
        register int i,j,k;
        int reply;
        register len = strlen(word);

        for(i=0 ; i < len ; i++){
                /* генерим слова , получающиеся удалением одной буквы */
                k=0;
                for(j=0 ; j < i ; j++)
                        version[k++]=word[j];
                for(j=i+1 ; j < len ; j++)
                        version[k++]=word[j];
                version[k]='\0';
                if( eqi( version, s )) return 1; /* OK */
        }
        return 0;       /* FAIL */
}
```

```
/* ПОТЕРЯНА БУКВА */
static int hole;   /* место, где вставлена ANYCHAR */
static int lost(uchar *word, uchar *s)
{
        register int i,j,k;
        register len = strlen(word);

        hole= (-1);
        for(i=0 ; i < len+1 ; i++){
                k=0;
                for(j=0 ; j < i ; j++)
                        version[k++]=word[j];
                version[k++]=ANYCHAR;
                for(j=i ; j < len ; j++)
                        version[k++]=word[j];
                version[k]='\0';
                if( strok( version, s )){
                        hole=i;
                        return 1;       /* OK */
                }
        }
        return 0;       /* FAIL */
}

/* ИЗМЕНИЛАСЬ ОДНА БУКВА (включает случай ошибки регистра) */
static int changed(uchar *word, uchar *s)
{
        register int i,j,k;
        register len = strlen(word);

        hole = (-1);
        for(i=0 ; i < len ; i++){
                k=0;
                for( j=0 ; j < i ; j++)
                        version[k++]=word[j];
                version[k++]=ANYCHAR;
                for( j=i+1 ; j < len ; j++)
                        version[k++]=word[j];
                version[k]='\0';
                if( strok( version,s)){
                        hole=i;
                        return 1;       /* OK */
                }
        }
        return 0;       /* FAIL */
}
```

```
/* УДВОЕННАЯ БУКВА */
static int duplicates(uchar *word, uchar *s, int leng)
{
        register int i,j,k;
        uchar tmp[80];

        if( eqi( word, s )) return 1;       /* OK */

        for(i=0;i < leng - 1; i++)
        /* ищем парные буквы */
                if( word[i]==word[i+1]){
                        k=0;
                        for(j=0 ; j < i ; j++)
                                tmp[k++]=word[j];
                        for(j=i+1 ; j < leng ; j++)
                                tmp[k++]=word[j];
                        tmp[k]='\0';
                        if( duplicates( tmp, s, leng-1) == 1)
                                return 1;       /* OK */
                }
        return 0;       /* FAIL */
}

/* ПЕРЕСТАВЛЕНЫ СОСЕДНИЕ БУКВЫ */
static int swapped(uchar *word, uchar *s)
{
        register int i,j,k;
        register len = strlen(word);

        for(i=0;i < len-1;i++){
                k=0;
                for(j=0 ; j < i ; j++)
                        version[k++]=word[j];
                version[k++]=word[i+1];
                version[k++]=word[i];
                for(j=i+2 ; j < len ; j++)
                        version[k++]=word[j];
                version[k]='\0';
                if( eqi( version, s))
                        return 1;       /* OK */
        }
        return 0;  /* FAIL */
}

uchar *words[] = {
        (uchar *) "bag",
        (uchar *) "bags",
        (uchar *) "cook",
        (uchar *) "cool",
        (uchar *) "bug",
        (uchar *) "buy",
        (uchar *) "cock",
        NULL
};

#define Bcase(x, operators)     case x: { operators; } break;

char *cname[5] = {
        "переставлены буквы",
        "удвоены буквы     ",
        "потеряна буква    ",
        "ошибочная буква   ",
        "лишняя буква      "
};
```

```
static int spellmatch( uchar *word       /* IN  слово для коррекции */
                     , uchar *words[]    /* IN  таблица допустимых слов */
                     , uchar **indx      /* OUT ответ */
){
        int i, code, total = (-1);
        uchar **ptr;

        if(!*word) return -1;

        for(ptr = words; *ptr; ++ptr)
                if(eqi(word, *ptr)){
                        if(indx) *indx = *ptr;
                        return 0;
                }
        /* Нет в таблице, нужен подбор похожих */
        for(ptr = words; *ptr; ++ptr){
                uchar *s = *ptr;
                int max = 5;
                for(i=0; i < max; i++){
                        switch( i ){
                        Bcase(0,code = swapped(word, s)                   )
                        Bcase(1,code = duplicates(word, s, strlen(word)) )
                        Bcase(2,code = lost(word, s)                      )
                        Bcase(3,code = changed(word, s)                   )
                        Bcase(4,code = superfluous(word, s)               )
                        }

                        if(code){
                                total++;
                                printf("?\t%s\t%s\n", cname[i], s);
                                if(indx) *indx = s;

                                /* В случае с дубликатами не рассматривать
                                 * на наличие лишних букв
                                 */
                                if(i==1) max = 4;
                        }
                }
        }
        return total;
}

void main(){
        uchar inpbuf[BUFSIZ];
        int n;
        uchar *reply, **ptr;

        setlocale(LC_ALL, "");
        for(ptr = words; *ptr; ptr++)
                printf("#\t%s\n", *ptr);

        do{
                printf("> "); fflush(stdout);
                if(gets((char *)inpbuf) == NULL) break;

                switch(spellmatch(inpbuf, words, &reply)){
                case -1:
                        printf("Нет такого слова\n"); break;
                case 0:
                        printf("Слово '%s'\n", reply); break;
                default:
                        printf("Неоднозначно\n");
                }
        } while(1);
}
```

7.53. Пока я сам писал эту программу, я сделал две ошибки, которые должны быть весьма характерны для новичков. Про них надо бы говорить раньше, в главе про строки и в самой первой главе, но тут они пришлись как раз к месту. Вопрос: что печатает следующая программа?

```
#include <stdio.h>

char *strings[] = {
        "Первая строка"
        "Вторая строка"
        "Третяя строка",
        "Четвертая строка",
        NULL
};

void main(){
        char **p;
        for(p=strings;*p;++p)
                printf("%s\n", *p);
}
```

А печатает она вот что:

```
Первая строкаВторая строкаТретяя строка
Четвертая строка
```

Дело в том, что ANSI компилятор Си *склеивает* строки:

```
        "начало строки"     "и ее конец"
```

если они разделены пробелами в смысле **isspace**, в том числе и пустыми строками. А в нашем объявлении массива строк *strings* мы потеряли несколько разделительных запятых!

Вторая ошибка касается того, что можно забыть поставить слово **break** в операторе **switch**, и долго после этого гадать о непредсказуемом поведении любого поступающего на вход значения. Дело просто: пробегаются все случаи, управление проваливается из **case** в следующий **case**, и так много раз подряд! Это и есть причина того, что в предыдущем примере все **case** оформлены нетривиальным макросом **Bcase**.

7.54. Составьте программу кодировки и раскодировки файлов по заданному ключу (строке символов).

7.55. Составьте программу, которая запрашивает анкетные данные типа фамилии, имени, отчества, даты рождения и формирует файл. Программа должна отлавливать ошибки ввода несимвольной и нецифровой информации, выхода составляющих даты рождения за допустимые границы с выдачей сообщений об ошибках. Программа должна давать возможность корректировать вводимые данные. Все данные об одном человеке записываются в одну строку файла через пробел. Вот возможный пример части диалога (ответы пользователя выделены жирно):

```
Введите месяц рождения [1-12]: 14  <ENTER>
*** Неправильный номер месяца (14).
Введите месяц рождения [1-12]: март  <ENTER>
*** Номер месяца содержит букву 'м'.
Введите месяц рождения [1-12]: <ENTER>
Вы хотите закончить ввод ? n
Введите месяц рождения [1-12]: 11  <ENTER>
        Ноябрь
Введите дату рождения  [1-30]: _
```

В таких программах обычно ответ пользователя вводится как строка:

```
printf("Введите месяц рождения [1-12]: ");
fflush(stdout); gets(input_string);
```

затем (если надо) отбрасываются лишние пробелы в начале и в конце строки, затем введенный текст *input_string* анализируется на допустимость символов (нет ли в нем не цифр?), затем строка преобразуется к нужному типу (например, при помощи функции **atoi** переводится в целое) и проверяется допустимость полученного значения, и.т.д.

Вводимую информацию сначала заносите в структуру; затем записывайте содержимое полей структуры в файл в текстовом виде (используйте функцию **fprintf**, а не **fwrite**).

7.56. Составьте программу, осуществляющую выборку информации из файла, сформированного в предыдущей задаче, и ее распечатку в табличном виде. Выборка должна осуществляться по значению любого заданного поля (т.е. вы выбираете поле, задаете его значение и получаете те строки, в которых значение

указанного поля совпадает с заказанным вами значением). Усложнение: используйте функцию сравнения строки с регулярным выражением для выборки по *шаблону* поля (т.е. отбираются только те строки, в которых значение заданного поля удовлетворяет шаблону). Для чтения файла используйте **fscanf**, либо **fgets** и затем **sscanf**. Второй способ лучше тем, что позволяет проверить по шаблону значение *любого* поля - не только текстового, но и числового: так *1234* (строка - изображение числа) удовлетворяет шаблону "*12**".

7.57. Составьте вариант программы подсчета служебных слов языка Си, не учитывающий появление этих слов, заключенных в кавычки.

7.58. Составьте программу удаления из программы на языке Си всех комментариев. Обратите внимание на особые случаи со строками в кавычках и символьными константами; так строка

```
char s[] = "/*";
```

не является началом комментария! Комментарии записывайте в отдельный файл.

7.59. Составьте программу выдачи перекрестных ссылок, т.е. программу, которая выводит список всех идентификаторов переменных, используемых в программе, и для каждого из идентификаторов выводит список номеров строк, в которые он входит.

7.60. Разработайте простую версию препроцессора для обработки операторов **#include**. В качестве прототипа такой программы можно рассматривать такую (она понимает директивы вида **#include** *имяфайла* - без <> или ""):

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

char KEYWORD[] = "#include ";   /* with a trailing space char */

void process(char *name, char *from){
        FILE *fp;
        char buf[4096];

        if((fp = fopen(name, "r")) == NULL){
                fprintf(stderr, "%s: cannot read \"%s\", %s\n",
                                from, name, strerror(errno));
                return;
        }
        while(fgets(buf, sizeof buf, fp) != NULL){
                if(!strncmp(buf, KEYWORD, sizeof KEYWORD - 1)){
                        char *s;

                        if((s = strchr(buf, '\n')) != NULL) *s = '\0';
                        fprintf(stderr, "%s: including %s\n",
                                        name, s = buf + sizeof KEYWORD - 1);
                        process(s, name);
                } else  fputs(buf, stdout);
        }
        fclose(fp);
}
int main(int ac, char *av[]){
        int i;

        for(i=1; i < ac; i++)
                process(av[i], "MAIN");
        return 0;
}
```

7.61. Разработайте простую версию препроцессора для обработки операторов **#define**. Сначала реализуйте макросы без аргументов. Напишите обработчик макросов вида

```
#macro имя(аргу,менты)
  тело макроса - можно несколько строк
#endm
```

7.62. Напишите программу, обрабатывающую определения **#ifdef**, **#else**, **#endif**. Учтите, что эти директивы могут быть вложенными:

```
#ifdef    A
# ifdef   B
   ...              /*  defined(A) && defined(B) */
# endif /*B*/
   ...              /*  defined(A) */
#else   /*not A*/
   ...              /* !defined(A) */
# ifdef   C
   ...              /* !defined(A) && defined(C) */
# endif /*C*/
#endif  /*A*/
```

7.63. Составьте программу моделирования простейшего калькулятора, который считывает в каждой строчке по одному числу (возможно со знаком) или по одной операции сложения или умножения, осуществляет операцию и выдает результат.

7.64. Составьте программу-калькулятор, которая производит операции сложения, вычитания, умножения, деления; операнды и знак арифметической операции являются строковыми аргументами функции **main**.

7.65. Составьте программу, вычисляющую значение командной строки, представляющей собой обратную польскую запись арифметического выражения. Например, 20 10 5 + * вычисляется как 20 * (10 + 5) .

7.66. Составьте функции работы со стеком:

- добавление в стек
- удаление вершины стека (с возвратом удаленного значения)

Используйте два варианта: стек-массив и стек-список.

7.67. Составьте программу, которая использует функции работы со стеком для перевода арифметических выражений языка Си в обратную польскую запись.

```
/*#!/bin/cc $* -lm
 * Калькулятор. Иллюстрация алгоритма превращения выражений
 * в польскую запись по методу приоритетов.
 */

#include <stdio.h>
#include <stdlib.h> /* extern double atof();            */
#include <math.h>   /* extern double sin(),  ...       */
#include <ctype.h>  /* isdigit(), isalpha(), ...       */
#include <setjmp.h> /* jmp_buf                          */

jmp_buf AGAIN;              /* контрольная точка */
err(n){ longjmp(AGAIN,n);} /* прыгнуть в контрольную точку */
```

```
/* ВЫЧИСЛИТЕЛЬ ---------------------------------------- */
/* Если вместо помещения операндов в стек stk[] просто
 * печатать операнды, а вместо выполнения операций над
 * стеком просто печатать операции, мы получим "польскую"
 * запись выражения:
 *      a+b       ->      a b +
 *      (a+b)*c   ->      a b + c *
 *      a + b*c   ->      a b c * +
 */
/* стек вычислений */
#define MAXDEPTH 20 /* глубина стеков */
int sp;             /* указатель стека (stack pointer) */
double stk[MAXDEPTH];

double dpush(d) double d; /* занести число в стек */
{
   if( sp == MAXDEPTH ){ printf("Стек операндов полон\n");err(1);}
   else return( stk[sp++] = d );
}

double dpop(){            /* взять вершину стека */
   if( !sp ){ printf("Стек операндов пуст\n"); err(2); }
   else return stk[--sp];
}

static double r,p; /* вспомогательные регистры */
void add()    { dpush( dpop() + dpop()); }
void mult()   { dpush( dpop() * dpop()); }
void sub()    { r = dpop(); dpush( dpop() - r); }
void divide() { r = dpop();
  if(r == 0.0){ printf("Деление на 0\n"); err(3); }
  dpush( dpop() / r );
}
void pwr() { r = dpop(); dpush( pow( dpop(), r )); }
void dup() { dpush( dpush( dpop())); }
void xchg(){ r = dpop(); p = dpop(); dpush(r); dpush(p); }
void neg() { dpush( - dpop()); }
void dsin(){ dpush( sin( dpop())); }
void dcos(){ dpush( cos( dpop())); }
void dexp(){ dpush( exp( dpop())); }
void dlog(){ dpush( log( dpop())); }
void dsqrt(){ dpush( sqrt( dpop())); }
void dsqr(){ dup(); mult(); }
/* M_PI и M_E определены в <math.h> */
void pi()  { dpush( M_PI /* число пи */ ); }
void e()   { dpush( M_E  /* число e  */ ); }
void prn() { printf("%g\n", dpush( dpop())); }
void printstk(){
  if( !sp ){ printf("Стек операндов пуст\n"); err(4);}
  while(sp) printf("%g ", dpop());
  putchar('\n');
}
```

```
/* КОМПИЛЯТОР ---------------------------------------- */
/* номера лексем */
#define END        (-3)         /* = */
#define NUMBER     (-2)         /* число */
#define BOTTOM      0           /* псевдолексема "дно стека" */

#define OPENBRACKET   1         /* (  */
#define FUNC          2         /* f( */
#define CLOSEBRACKET  3         /* )  */
#define COMMA         4         /* ,  */

#define PLUS          5         /* +  */
#define MINUS         6         /* -  */
#define MULT          7         /* *  */
#define DIV           8         /* /  */
#define POWER         9         /* ** */

/* Приоритеты */
#define NOTDEF    333   /* не определен */
#define INFINITY 3000   /* бесконечность */

/* Стек транслятора */
typedef struct _opstack {
        int cop;        /* код операции */
        void (*f)();    /* "отложенное" действие */
} opstack;
int osp;        /* operations stack pointer */
opstack ost[MAXDEPTH];

void push(n, func) void (*func)();
{
   if(osp == MAXDEPTH){ printf("Стек операций полон\n");err(5);}
   ost[osp].cop = n;  ost[osp++].f = func;
}
int pop(){
   if( !osp ){ printf("Стек операций пуст\n"); err(6); }
   return ost[--osp].cop;
}
int top(){
   if( !osp ){ printf("Стек операций пуст\n"); err(7); }
   return ost[osp-1].cop;
}
void (*topf())(){
   return ost[osp-1].f;
}
#define drop()          (void)pop()

void nop(){ printf( "???\n" ); } /* no operation */
void obr_err(){ printf( "Не хватает )\n" ); err(8); }
```

```
/* Таблица приоритетов */
struct synt{
        int inp_prt;    /* входной приоритет     */
        int stk_prt;    /* стековый приоритет    */
        void (*op)();   /* действие над стеком вычислений */
} ops[] = {
  /* BOTTOM       */  {NOTDEF,   -1,     nop    },
  /* OPENBRACKET  */  {INFINITY,  0,     obr_err},
  /* FUNC         */  {INFINITY,  0,     obr_err},
  /* CLOSEBRACKET */  {1,        NOTDEF, nop    },  /* NOPUSH */
  /* COMMA        */  {1,        NOTDEF, nop    },  /* NOPUSH */
  /* PLUS         */  {1,         1,     add    },
  /* MINUS        */  {1,         1,     sub    },
  /* MULT         */  {2,         2,     mult   },
  /* DIV          */  {2,         2,     divide },
  /* POWER        */  {3,         3,     pwr    }
};

#define stkprt(i)    ops[i].stk_prt
#define inpprt(i)    ops[i].inp_prt
#define perform(i) (*ops[i].op)()

/* значения, заполняемые лексическим анализатором */
double value; void (*fvalue)();
int tprev;  /* предыдущая лексема */
```

```
/* Транслятор в польскую запись + интерпретатор */
void reset(){ sp = osp = 0; push(BOTTOM, NULL); tprev = END;}
void calc(){
  int t;
  do{
        if( setjmp(AGAIN))
                printf( "Стеки после ошибки сброшены\n" );
        reset();
        while((t = token()) != EOF && t != END){
                if(t == NUMBER){
                        if(tprev == NUMBER){
                             printf("%g:Два числа подряд\n",value);
                             err(9);
                        }
                        /* любое число просто заносится в стек */
                        tprev = t; dpush(value); continue;
                }
                /* иначе - оператор */
                tprev = t;
        /* Выталкивание и выполнение операций со стека */
                while(inpprt(t) <= stkprt( top()) )
                        perform( pop());
        /* Сокращение или подмена скобок */
                if(t == CLOSEBRACKET){
                        if( top() == OPENBRACKET || top() == FUNC ){
                                void (*ff)() = topf();
                                drop(); /* схлопнуть скобки */
                                /* обработка функции */
                                if(ff)        (*ff)();
                        }else{ printf( "Не хватает (\n"); err(10); }
                }
        /* Занесение операций в стек (кроме NOPUSH-операций) */
                if(t != CLOSEBRACKET && t != COMMA)
                        push(t,   t == FUNC ? fvalue : NULL );
        }
        if( t != EOF ){
          /* Довыполнить оставшиеся операции */
            while( top() != BOTTOM )
                perform( pop());
            printstk();     /* печать стека вычислений (ответ) */
        }
  } while (t != EOF);
}

/* Лексический анализатор ---------------------------- */
extern void getn(), getid(), getbrack();
int token(){    /* прочесть лексему */
        int c;
        while((c = getchar())!= EOF && (isspace(c) || c == '\n'));
        if(c == EOF) return EOF;
        ungetc(c, stdin);
        if(isdigit(c)){  getn(); return NUMBER; }
        if(isalpha(c)){ getid(); getbrack(); return FUNC; }
        return getop();
}
```

```
/* Прочесть число (с точкой) */
void getn(){
        int c, i;  char s[80];
        s[0] = getchar();
        for(i=1; isdigit(c = getchar()); i++ )  s[i] = c;
        if(c == '.'){   /* дробная часть */
           s[i] = c;
           for(i++; isdigit(c = getchar()); i++)  s[i] = c;
        }
        s[i] = '\0'; ungetc(c, stdin); value = atof(s);
}

/* Прочесть операцию */
int getop(){
        int c;
        switch( c = getchar()){
        case EOF:       return EOF;
        case '=':       return END;
        case '+':       return PLUS;
        case '-':       return MINUS;
        case '/':       return DIV;
        case '*':       c = getchar();
                        if(c == '*') return POWER;
                        else{ ungetc(c, stdin); return MULT; }
        case '(':       return OPENBRACKET;
        case ')':       return CLOSEBRACKET;
        case ',':       return COMMA;
        default:        printf( "Ошибочная операция %c\n", c);
                        return token();
        }
}

struct funcs{   /* Таблица имен функций */
        char *fname; void (*fcall)();
} tbl[] = {
        { "sin", dsin }, { "cos",   dcos  },
        { "exp", dexp }, { "sqrt",  dsqrt },
        { "sqr", dsqr }, { "pi",    pi    },
        { "sum", add  }, { "ln",    dlog  },
        { "e",   e    }, { NULL,    NULL  }
};

char *lastf;    /* имя найденной функции */
/* Прочесть имя функции */
void getid(){
        struct funcs *ptr = tbl;
        char name[80]; int c, i;
        *name = getchar();
        for(i=1; isalpha(c = getchar()); i++) name[i] = c;
        name[i] = '\0'; ungetc(c, stdin);
        /* поиск в таблице */
        for( ; ptr->fname; ptr++ )
                if( !strcmp(ptr->fname, name)){
                        fvalue = ptr->fcall;
                        lastf =  ptr->fname; return;
                }
        printf( "Функция \"%s\" неизвестна\n", name ); err(11);
}
```

```
/* прочесть открывающую скобку после имени функции */
void getbrack(){
  int c;
  while((c = getchar()) != EOF && c != '(' )
  if( !isspace(c) && c != '\n' ){
      printf("Между именем функции %s и ( символ %c\n", lastf, c);
      ungetc(c, stdin); err(12);
  }
}

void main(){ calc();}

/* Примеры:
        ( sin( pi() / 4 + 0.1 ) + sum(2, 4 + 1)) * (5 - 4/2) =
                ответ: 23.3225
        (14 + 2 ** 3 * 7 + 2 * cos(0)) / ( 7 - 4 ) =
                ответ: 24
*/
```

7.68. Приведем еще один арифметический вычислитель, использующий классический рекурсивный подход:

```
/* Калькулятор на основе рекурсивного грамматического разбора.
 * По мотивам арифметической части программы csh (СиШелл).
 * csh написан Биллом Джоем (Bill Joy).
    : var1 = (x = 1+3) * (y=x + x++)            36
    : s = s + 1                                 ошибка
    : y                                         9
    : s = (1 + 1 << 2) == 1 + (1<<2)            0
    : var1 + 3 + -77                            -38
    : a1 = 3; a2 = (a4=a3 = 2; a1++)+a4+2       8
    : sum(a=2;b=3, a++, a*3-b)                 12
 */

#include <stdio.h>
#include <ctype.h>
#include <setjmp.h>

typedef enum { NUM, ID, OP, OPEN, CLOSE, UNKNOWN, COMMA, SMC } TokenType;

char *toknames[] = { "number", "identifier", "operation",
  "open_paren", "close_paren", "unknown", "comma", "semicolon" };

typedef struct _Token {
        char *token;            /* лексема (слово)     */
        struct _Token *next;    /* ссылка на следующую */
        TokenType type;         /* тип лексемы         */
} Token;

extern void *malloc(unsigned); extern char *strchr(char *, char);

char *strdup(const char *s){
      char *p = (char *)malloc(strlen(s)+1);
      if(p) strcpy(p,s); return p;
}

/* Лексический разбор ----------------------------------------*/
/* Очистить цепочку токенов */
void freelex(Token **p){
     Token *thisTok = *p;
     while( thisTok ){ Token *nextTok = thisTok->next;
        free((char *) thisTok->token); free((char *) thisTok);
        thisTok = nextTok;
     }
     *p = NULL;
}
```

```
/* Добавить токен в хвост списка */
void addtoken(Token **hd, Token **tl, char s[], TokenType t){
     Token *newTok = (Token *) malloc(sizeof(Token));
     newTok->next  = (Token *) NULL;
     newTok->token = strdup(s); newTok->type = t;
     if(*hd == NULL) *hd = *tl = newTok;
     else{  (*tl)->next = newTok; *tl = newTok; }
}

/* Разобрать строку в список лексем (токенов) */
#define opsym(c) ((c) && strchr("+-=!~^|&*/%<>", (c)))
#define is_alpha(c) (isalpha(c) || (c) == '_')
#define is_alnum(c) (isalnum(c) || (c) == '_')

void lex(Token **hd, Token **tl, register char *s){
      char *p, csave; TokenType type;

      while(*s){
          while( isspace(*s)) ++s; p = s;
          if( !*s ) break;
               if(isdigit (*s)){ type = NUM; while(isdigit (*s))s++; }
          else if(is_alpha(*s)){ type = ID;  while(is_alnum(*s))s++; }
          else if(*s == '('){    type = OPEN;  s++; }
          else if(*s == ')'){    type = CLOSE; s++; }
          else if(*s == ','){    type = COMMA; s++; }
          else if(*s == ';'){    type = SMC;   s++; }
          else if(opsym(*s)){    type = OP;  while(opsym(*s))  s++; }
          else {                 type = UNKNOWN;               s++; }
          csave = *s; *s = '\0'; addtoken(hd, tl, p, type); *s = csave;
      }
}

/* Распечатка списка лексем */
void printlex(char *msg, Token *t){
     if(msg && *msg) printf("%s: ", msg);
     for(; t != NULL; t = t->next)
        printf("%s`%s' ", toknames[(int)t->type], t->token);
     putchar('\n');
}

/* Система переменных ---------------------------------------- */
#define NEXT(v)          *v = (*v)->next
#define TOKEN(v)         (*v)->token
#define TYPE(v)          (*v)->type
#define eq(str1, str2)  (!strcmp(str1, str2))
jmp_buf breakpoint;
#define ERR(msg,val) { printf("%s\n", msg);longjmp(breakpoint, val+1);}

typedef struct {
     char *name;         /* Имя переменной      */
     int value;          /* Значение переменной */
     int isset;          /* Получила ли значение ? */
} Var;
#define MAXV 40
Var vars[MAXV];

/* Получить значение переменной */
int getVar(char *name){ Var *ptr;
   for(ptr=vars; ptr->name; ptr++)
       if(eq(name, ptr->name)){
          if(ptr->isset) return ptr->value;
          printf("%s: ", name); ERR("variable is unbound yet", 0);
       }
   printf("%s: ", name); ERR("undefined variable", 0);
}
```

```
/* Создать новую переменную       */
Var *internVar(char *name){ Var *ptr;
   for(ptr=vars; ptr->name; ptr++)
       if(eq(name, ptr->name)) return ptr;
   ptr->name = strdup(name);
   ptr->isset = 0; ptr->value = 0; return ptr;
}

/* Установить значение переменной */
void setVar(Var *ptr, int val){ ptr->isset = 1; ptr->value = val; }

/* Распечатать значения переменных */
void printVars(){ Var *ptr;
     for(ptr=vars; ptr->name; ++ptr)
         printf("\t%s %s %d\n", ptr->isset ? "BOUND  ":"UNBOUND",
                     ptr->name, ptr->value);
}

/* Синтаксический разбор и одновременное вычисление ----------- */
/* Вычисление встроенных функций */
int apply(char *name, int args[], int nargs){
    if(eq(name, "power2")){
        if(nargs != 1) ERR("power2: wrong argument count", 0);
        return (1 << args[0]);
    } else if(eq(name, "min")){
        if(nargs != 2) ERR("min: wrong argument count", 0);
        return (args[0] < args[1] ? args[0] : args[1]);
    } else if(eq(name, "max")){
        if(nargs != 2) ERR("max: wrong argument count", 0);
        return (args[0] < args[1] ? args[1] : args[0]);
    } else if(eq(name, "sum")){ register i, sum;
        for(i=0, sum=0; i < nargs; sum += args[i++]);
        return sum;
    } else if(eq(name, "rand")){
        switch(nargs){
        case 0:  return rand();
        case 1:  return rand() % args[0];
        case 2:  return args[0] + rand() % (args[1] - args[0] + 1);
        default: ERR("rand: wrong argument count", 0);
        }
    }
    ERR("Unknown function", args[0]);
}

/* Вычислить выражение из списка лексем.        */
/* Синтаксис задан праворекурсивной грамматикой */
int expr(Token *t){ int val = 0;
    if(val = setjmp(breakpoint)) return val - 1;
    val = expression(&t);
    if(t){ printlex(NULL, t); ERR("Extra tokens", val); }
    return val;
}

/* <EXPRESSION> =   <EXPASS>  |
                    <EXPASS>  ";" <EXPRESSION>          */
int expression(Token **v){ int arg = expass(v);
    if(*v && TYPE(v) == SMC ){
        NEXT(v); return expression(v);
    } else return arg;
}
```

```
/* <EXPASS> =       <ПЕРЕМЕННАЯ> "=" <EXPASS> |
                    <EXP0>                                  */
int expass(Token **v){ int arg;
    if(*v && (*v)->next && (*v)->next->type == OP &&
       eq((*v)->next->token, "=")){ Var *ptr;
           /* присваивание (assignment) */
           if( TYPE(v) != ID ) /* слева нужна переменная */
                ERR("Lvalue needed", 0);
           ptr = internVar(TOKEN(v));
           NEXT(v); NEXT(v); setVar(ptr, arg = expass(v)); return arg;
    }
    return exp0(v);
}

/* <EXP0>  =  <EXP1>  |   <EXP1> "||" <EXP0>  */
int exp0(Token **v){ int arg = exp1(v);
    if(*v && TYPE(v) == OP && eq(TOKEN(v), "||")){
          NEXT(v); return(exp0(v) || arg );
          /* помещаем arg ВТОРЫМ, чтобы второй операнд вычислялся
           * ВСЕГДА (иначе не будет исчерпан список токенов и
           * возникнет ошибка в expr(); Это не совсем по правилам Си.
           */
    } else return arg;
}

/* <EXP1>  =  <EXP2>  |   <EXP2> "&&" <EXP1>     */
int exp1(Token **v){ int arg = exp2(v);
    if(*v && TYPE(v) == OP && eq(TOKEN(v), "&&")){
          NEXT(v); return(exp1(v) && arg);
    } else return arg;
}

/* <EXP2>  =  <EXP2A>  |   <EXP2A> "|" <EXP2>    */
int exp2(Token **v){ int arg = exp2a(v);
    if(*v && TYPE(v) == OP && eq(TOKEN(v), "|")){
          NEXT(v); return( arg | exp2(v));
    } else return arg;
}

/* <EXP2A>  =  <EXP2B>  |   <EXP2B> "^" <EXP2A>  */
int exp2a(Token **v){ int arg = exp2b(v);
    if(*v && TYPE(v) == OP && eq(TOKEN(v), "^")){
          NEXT(v); return( arg ^ exp2a(v));
    } else return arg;
}

/* <EXP2B>  =  <EXP2C>  |   <EXP2C> "&" <EXP2B>  */
int exp2b(Token **v){ int arg = exp2c(v);
    if(*v && TYPE(v) == OP && eq(TOKEN(v), "&")){
          NEXT(v); return( arg & exp2b(v));
    } else return arg;
}

/* <EXP2C>  =  <EXP3>  |   <EXP3> "==" <EXP3>
                       |   <EXP3> "!=" <EXP3>    */
int exp2c(Token **v){ int arg = exp3(v);
           if(*v && TYPE(v) == OP && eq(TOKEN(v), "==")){
           NEXT(v); return( arg == exp3(v));
    } else if(*v && TYPE(v) == OP && eq(TOKEN(v), "!=")){
           NEXT(v); return( arg != exp3(v));
    } else return arg;
}
```

```
/* <EXP3>  =  <EXP3A>  |   <EXP3A> ">"  <EXP3>
                       |   <EXP3A> "<"  <EXP3>
                       |   <EXP3A> ">=" <EXP3>
                       |   <EXP3A> "<=" <EXP3>    */
int exp3(Token **v){ int arg = exp3a(v);
          if(*v && TYPE(v) == OP && eq(TOKEN(v), ">")){
          NEXT(v); return( arg && exp3(v));
    }else if(*v && TYPE(v) == OP && eq(TOKEN(v), "<")){
          NEXT(v); return( arg && exp3(v));
    }else if(*v && TYPE(v) == OP && eq(TOKEN(v), ">=")){
          NEXT(v); return( arg && exp3(v));
    }else if(*v && TYPE(v) == OP && eq(TOKEN(v), "<=")){
          NEXT(v); return( arg && exp3(v));
    } else return arg;
}

/* <EXP3A>  =  <EXP4>  |   <EXP4> "<<" <EXP3A>
                       |   <EXP4> ">>" <EXP3A>    */
int exp3a(Token **v){ int arg = exp4(v);
          if(*v && TYPE(v) == OP && eq(TOKEN(v), "<<")){
          NEXT(v); return( arg << exp3a(v));
    }else if(*v && TYPE(v) == OP && eq(TOKEN(v), ">>")){
          NEXT(v); return( arg && exp3a(v));
    } else return arg;
}

/* <EXP4>  =  <EXP5>  |   <EXP5> "+" <EXP4>
                      |   <EXP5> "-" <EXP4>       */
int exp4(Token **v){ int arg = exp5(v);
          if(*v && TYPE(v) == OP && eq(TOKEN(v), "+")){
          NEXT(v); return( arg + exp4(v));
    }else if(*v && TYPE(v) == OP && eq(TOKEN(v), "-")){
          NEXT(v); return( arg - exp4(v));
    } else return arg;
}

/* <EXP5>  =  <EXP6>  |   <EXP6> "*" <EXP5>
                      |   <EXP6> "/" <EXP5>
                      |   <EXP6> "%" <EXP5>       */
int exp5(Token **v){ int arg = exp6(v), arg1;
          if(*v && TYPE(v) == OP && eq(TOKEN(v), "*")){
          NEXT(v); return( arg * exp5(v));
    }else if(*v && TYPE(v) == OP && eq(TOKEN(v), "/")){
          NEXT(v); if((arg1 = exp5(v)) == 0) ERR("Zero divide", arg);
          return( arg / arg1);
    }else if(*v && TYPE(v) == OP && eq(TOKEN(v), "%")){
          NEXT(v); if((arg1 = exp5(v)) == 0) ERR("Zero module", arg);
          return( arg % arg1);
    } else return arg;
}
```

```
/* <EXP6>  = "!"<EXP6> | "~"<EXP6> | "-"<EXP6>
     | "(" <EXPRESSION> ")"
     |  <ИМЯФУНКЦИИ> "(" [ <EXPRESSION> [ "," <EXPRESSION> ]... ] ")"
     |  <ЧИСЛО>
     |  <СН_ПЕРЕМЕННАЯ>                              */
int exp6(Token **v){ int arg;
    if( !*v) ERR("Lost token", 0);
    if(TYPE(v) == OP && eq(TOKEN(v), "!")){
        NEXT(v); return !exp6(v);
    }
    if(TYPE(v) == OP && eq(TOKEN(v), "~")){
        NEXT(v); return ~exp6(v);
    }
    if(TYPE(v) == OP && eq(TOKEN(v), "-")){
        NEXT(v); return -exp6(v);    /* унарный минус */
    }
    if(TYPE(v) == OPEN){
        NEXT(v); arg = expression(v);
        if( !*v || TYPE(v) != CLOSE) ERR("Lost ')'", arg);
        NEXT(v); return arg;
    }
    if(TYPE(v) == NUM){  /* изображение числа */
        arg = atoi(TOKEN(v)); NEXT(v); return arg;
    }
    if(TYPE(v) == ID){
        char *name = (*v)->token; int args[20], nargs = 0;
        NEXT(v);
        if(! (*v && TYPE(v) == OPEN)){  /* Переменная */
           return expvar(v, name);
        }
        /* Функция */
        args[0] = 0;
        do{ NEXT(v);
            if( *v && TYPE(v) == CLOSE ) break; /* f() */
            args[nargs++] = expression(v);
        }   while( *v && TYPE(v) == COMMA);

        if(! (*v && TYPE(v) == CLOSE)) ERR("Error in '()'", args[0]);
        NEXT(v);
        return apply(name, args, nargs);
    }
    printlex(TOKEN(v), *v); ERR("Unknown token type", 0);
}

/* <СН_ПЕРЕМЕННАЯ>  =   <ПЕРЕМЕННАЯ>      |
                        <ПЕРЕМЕННАЯ> "++" |
                        <ПЕРЕМЕННАЯ> "--"
 Наши операции ++ и -- соответствуют ++x и --x из Си         */
int expvar(Token **v, char *name){
    int arg = getVar(name); Var *ptr = internVar(name);
    if(*v && TYPE(v) == OP){
      if(eq(TOKEN(v), "++")){ NEXT(v); setVar(ptr, ++arg); return arg; }
      if(eq(TOKEN(v), "--")){ NEXT(v); setVar(ptr, --arg); return arg; }
    }
    return arg;
}
```

```
/* Головная функция ------------------------------------------ */
char input[256];
Token *head, *tail;

void main(){
    do{ printf(": "); fflush(stdout);
      if( !gets(input)) break;
      if(!*input){ printVars(); continue; }
      if(eq(input, "!!")) ; /* ничего не делать, т.е. повторить */
      else{ if(head) freelex(&head); lex(&head, &tail, input); }
      printf("Result: %d\n", expr(head));
    } while(1); putchar('\n');
}
```

7.69. Напишите программу, выделяющую *n*-ое поле из каждой строки файла. Поля разделяются двоеточиями. Предусмотрите задание символа-разделителя из аргументов программы. Используйте эту программу для выделения поля "домашний каталог" из файла */etc/passwd*. Для выделения очередного поля можно использовать следующую процедуру:

```
main(){
   char c, *next, *strchr(); int nfield;
   char *s = "11111:222222222:333333:444444";

   for(nfield=0;;nfield++){
     if(next = strchr(s, ':')){
        c= *next; *next= '\0';
     }
     printf( "Поле #%d: '%s'\n", nfield, s);
        /* можно сделать с полем s что-то еще */
     if(next){ *next= c; s= next+1; continue; }
     else    { break; /* последнее поле */    }
   }
}
```

7.70. Разработайте архитектуру и систему команд учебной машины и напишите интерпретатор учебного ассемблера, отрабатывающего по крайней мере такие команды:

```
mov пересылка (:=)     add сложение
sub вычитание          cmp сравнение и выработка признака
jmp переход            jeq переход, если ==
jlt переход, если <    jle переход, если <=
neg изменение знака    not инвертирование признака
```

7.71. Напишите программу, преобразующую определения функций Си в "старом" стиле в "новый" стиль стандарта **ANSI** ("прототипы" функций).

```
f(x, y, s, v)
      int x;
      char *s;
      struct elem *v;
{ ... }
```

преобразуется в

```
int f(int x, int y, char *s, struct elem *v)
{ ... }
```

(обратите внимание, что переменная *y* и сама функция **f** описаны по умолчанию как **int**). Еще пример:

```
char *ff()    { ... }
     заменяется на
char *ff(void){ ... }
```

В данной задаче вам возможно придется использовать программу **lex**.

В списке аргументов прототипа должны быть явно указаны типы *всех* аргументов - описатель **int** нельзя опускать. Так

```
q(x, s) char *s; { ... }  // не прототип, допустимо.
                           // x - int по умолчанию.
q(x,     char *s);        // недопустимо.
q(int x, char *s);        // верно.
```

Собственно под "прототипом" понимают предварительное описание функции в новом стиле - где вместо тела {...} сразу после заголовка стоит точка с запятой.

```
long f(long x, long y);                 /* прототип */
        ...
long f(long x, long y){ return x+y; }  /* реализация */
```

В прототипе имена аргументов можно опускать:

```
long f(long, long);                /* прототип */
char *strchr(char *, char);
```

Это предварительное описание помещают где-нибудь в начале программы, до первого вызова функции. В современном Си прототипы заменяют описания вида

```
extern long f();
```

о которых мы говорили раньше. Прототипы предоставляют программисту механизм для автоматического контроля формата вызова функции. Так, если функция имеет прототип

```
double f( double );
```

и вызывается как

```
double x = f( 12 );
```

то компилятор автоматически превратит это в

```
double x = f( (double) 12 );
```

(поскольку существует приведение типа от **int** к **double**); если же написано

```
f( "привет" );
```

то компилятор сообщит об ошибке (так как нет преобразования типа (**char ***) в **double**).

Прототип принуждает компилятор проверять:

a) соответствие ТИПОВ фактических параметров (при вызове) типам формальных параметров (в прототипе);

b) соответствие КОЛИЧЕСТВА фактических и формальных параметров;

c) тип возвращаемого функцией значения.

Прототипы обычно помещают в include-файлы. Так в **ANSI** стандарте Си предусмотрен файл, подключаемый

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

в котором определены прототипы функций из стандартной библиотеки языка Си. Черезвычайно полезно писать эту директиву include, чтобы компилятор проверял, верно ли вы вызываете стандартные функции.

Заметим, что если вы определили прототипы каких-то функций, но в своей программе используете *не все* из этих функций, то функции, соответствующие "лишним" прототипам, НЕ будут добавляться к вашей программе из библиотеки. Т.е. прототипы - это *указание* компилятору; ни в какие машинные команды они не транслируются. То же самое касается описаний внешних переменных и функций в виде

```
extern int x;
extern char *func();
```

Если вы не используете переменную или функцию с таким именем, то эти строки не имеют никакого эффекта (как бы вообще отсутствуют).

7.72. Обратная задача: напишите преобразователь из нового стиля в старый.

```
int f( int x, char *y ){ ... }
```

переводить в

```
int f( x, y ) int x; char *y; { ... }
```

7.73. Довольно легко использовать прототипы таким образом, что они потеряют всякий смысл. Для этого надо написать программу, состоящую из нескольких файлов, и в каждом файле использовать свои прототипы для одной и той же функции. Так бывает, когда вы поменяли функцию и прототип в одном файле, быть может во втором, но забыли сделать это в остальных.

```
--------
файл a.c
--------
void g(void);
void h(void);

int x = 0, y = 13;

void f(int arg){
        printf("f(%d)\n", arg);
        x = arg;
        x++;
}

int main(int ac, char *av[]){
        h();
        f(1);
        g();
        printf("x=%d y=%d\n", x, y);
        return 0;
}

--------
файл b.c
--------
extern int x, y;

int f(int);

void g(){
        y = f(5);
}

--------
файл c.c
--------
void f();

void h(){
        f();
}
```

Выдача программы:

```
abs@wizard$ cc a.c b.c c.c -o aaa
a.c:
b.c:
c.c:
abs@wizard$ aaa
f(-277792360)
f(1)
f(5)
x=6 y=5
abs@wizard$
```

Обратите внимание, что во всех трех файлах f() имеет разные прототипы! Поэтому программа печатает нечто, что довольно-таки бессмысленно!

Решение таково: стараться вынести прототипы в include-файл, чтобы все файлы программы включали одни и те же прототипы. Стараться, чтобы этот include-файл включался также в файл с самим определением функции. В таком случае изменение только заголовка функции или только прототипа вызовет ругань компилятора о несоответствии. Вот как должен выглядеть наш проект:

```
-------------
файл header.h
-------------
extern int x, y;
void f(int arg);
int main(int ac, char *av[]);
void g(void);
void h(void);

--------
файл a.c
--------
#include "header.h"

int x = 0, y = 13;

void f(int arg){
        printf("f(%d)\n", arg);
        x = arg;
        x++;
}

int main(int ac, char *av[]){
        h();
        f(1);
        g();
        printf("x=%d y=%d\n", x, y);
        return 0;
}

--------
файл b.c
--------
#include "header.h"

void g(){
        y = f(5);
}

--------
файл c.c
--------
#include "header.h"

void h(){
        f();
}
```

Попытка компиляции:

```
abs@wizard$ cc a.c b.c c.c -o aaa
a.c:
b.c:
"b.c", line 4: operand cannot have void type: op "="
"b.c", line 4: assignment type mismatch:
        int "=" void
cc: acomp failed for b.c
c.c:
"c.c", line 4: prototype mismatch: 0 args passed, 1 expected
cc: acomp failed for c.c
```

8. Экранные библиотеки и работа с видеопамятью.

Терминал в **UNIX** с точки зрения программ - это файл. Он представляет собой *два* устройства: при записи **write**() в этот файл осуществляется вывод на *экран*; при чтении **read**()-ом из этого файла - читается информация с *клавиатуры*.

Современные терминалы в определенном смысле являются устройствами прямого доступа:

- информация может быть выведена в любое место экрана, а не только последовательно строка за строкой.
- некоторые терминалы позволяют прочесть содержимое произвольной области экрана в вашу программу.

Традиционные терминалы являются самостоятельными устройствами, общающимися с компьютером через линию связи. Протокол† общения образует *систему команд терминала* и может быть различен для терминалов разных моделей. Поэтому библиотека работы с традиционным терминалом должна решать следующие проблемы:

- настройка на систему команд данного устройства, чтобы одна и та же программа работала на разных типах терминалов.
- эмуляция недостающих в системе команд; максимальное использование предоставленных терминалом возможностей.
- мимнимизация передачи данных через линию связи (для ускорения работы).
- было бы полезно, чтобы библиотека предоставляла пользователю некоторые логические абстракции, вроде ОКОН - прямоугольных областей на экране, ведущих себя подобно маленьким терминалам.

В **UNIX** эти задачи решает стандартная библиотека **curses** (а только первую задачу - более простая библиотека **termcap**). Для настройки на систему команд конкретного дисплея эти библиотеки считывают описание системы команд, хранящееся в файле */etc/***termcap**. Кроме них бывают и другие экранные библиотеки, а также существуют иные способы работы с экраном (через видеопамять, см. ниже).

В задачах данного раздела вам придется пользоваться библиотекой **curses**. При компиляции программ эта библиотека подключается при помощи указания ключа **-l***curses*, как в следующем примере:

```
cc progr.c -Ox -o progr -lcurses -lm
```

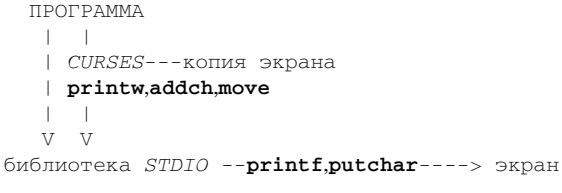
Здесь подключаются две библиотеки: */usr/lib/libcurses.a* (работа с экраном) и */usr/lib/libm.a* (математические функции, вроде **sin**, **fabs**). Ключи для подключения библиотек должны быть записаны в команде САМЫМИ ПОСЛЕДНИМИ. Заметим, что стандартная библиотека языка Си (содержащая системные вызовы, библиотеку **stdio** (функции **printf**, **scanf**, **fread**, **fseek**, ...), разные часто употребляемые функции (**strlen**, **strcat**, **sleep**, **malloc**, **rand**, ...)) */lib/libc.a* подключается автоматически и не требует указания ключа **-l***c*.

В начале своей программы вы должны написать директиву

```
#include <curses.h>
```

подключающую файл */usr/include/curses.h*, в котором описаны форматы данных, используемых библиотекой **curses**, некоторые предопределенные константы и.т.п. (это надо, чтобы ваша программа пользовалась именно этими стандартными соглашениями). Посмотрите в этот файл!

Когда вы пользуетесь **curses**-ом, вы НЕ должны пользоваться функциями стандартной библиотеки **stdio** для непосредственного вывода на экран; так вы не должны пользоваться функциями **printf**, **putchar**. Это происходит потому, что **curses** хранит в памяти процесса копию содержимого экрана, и если вы выводите что-либо на экран терминала обходя функции библиотеки **curses**, то реальное содержимое экрана и позиция курсора на нем перестают соответствовать хранимым в памяти, и библиотека **curses** начнет выводить неправильное изображение.



---

† Под *протоколом* в программировании подразумевают ряд соглашений двух сторон (сервера и клиентов; двух машин в сети (кстати, термин для обозначения машины в сети - "host" или "site")) о формате (правилах оформления) и смысле данных в передаваемых друг другу сообщениях. Аналогия из жизни - человеческие речь и язык. Речь всех людей состоит из одних и тех же звуков и может быть записана одними и теми же буквами (а данные - байтами). Но если два человека говорят на разных языках - т.е. по-разному конструируют фразы и интерпретируют звуки - они не поймут друг друга!

Таким образом, **curses** является дополнительным "слоем" между вашей программой и стандартным выводом и игнорировать этот слой не следует.

Напомним, что изображение, создаваемое при помощи библиотеки **curses**, сначала формируется в памяти программы без выполнения каких-либо операций с экраном дисплея (т.е. все функции **wmove**, **waddch**, **waddstr**, **wprintw** изменяют только ОБРАЗЫ окон в памяти, а на экране ничего не происходит!). И лишь только *ПОСЛЕ* того, как вы вызовете функцию **refresh**() ("обновить"), все изменения происшедшие в окнах будут отображены на экране дисплея (такое *одновременное* обновление всех изменившихся частей экрана позволяет провести ряд оптимизаций). Если вы забудете сделать **refresh** - экран останется неизменным. Обычно эту функцию вызывают перед тем, как запросить у пользователя какой-либо *ввод* с клавиатуры, чтобы пользователь увидел текущую "свежую" картинку. Хранение содержимого окон в памяти программы позволяет ей считывать содержимое окон, тогда как большинство *обычных* терминалов не способны выдать в компьютер содержимое какой-либо области экрана.

Общение с терминалом через линию связи (или вообще через последовательный протокол) является довольно медленным. На персональных компьютерах существует другой способ работы с экраном: через прямой доступ в так называемую "*видеопамять*" - специальную область памяти компьютера, содержимое которой аппаратно отображается на экране консоли. Работа с экраном превращается для программиста в работу с этим массивом байт (запись/чтение). Программы, пользующиеся этим способом, просты и работают очень быстро (ибо доступ к памяти черезвычайно быстр, и сделанные в ней изменения "проявляются" на экране почти мгновенно). Недостаток таких программ - привязанность к конкретному типу машины. Эти программы немобильны и не могут работать ни на обычных терминалах (подключаемых к линии связи), ни на машинах с другой структурой видеопамяти. Выбор между "традиционной" работой с экраном и прямым доступом (фактически - между мобильностью и скоростью) - вопрос принципиальный, тем не менее принятие решения зависит только от вас. Видеопамять **IBM PC** в текстовом режиме 80x25 16 цветов имеет следующую структуру:

```
   struct symbol{       /* IBM PC family           */
        char chr;       /* код символа             */
        char attr;      /* атрибуты символа (цвет) */
   } mem[ 25 ] [ 80 ];  /* 25 строк по 80 символов */

            Структура байта атрибутов:
  ----------------------------------------
  | 7   | 6 | 5 | 4 | 3         | 2 | 1 | 0 | # бита
  ------------------|-----------------------
  |blink| R | G | B | intensity | r | g | b | цвет
  ------------------|-----------------------
  background (фон)  | foreground (цвет букв)

 R - red (красный) G - green (зеленый) B - blue (синий)
 blink     - мерцание букв (не фона!)
 intensity - повышенная яркость
```

Координатная система на экране: верхний левый угол экрана имеет координаты (0,0), ось X горизонтальна, ось Y вертикальна и направлена *сверху вниз*.

Цвет символа получается смешиванием 3х цветов: красного, зеленого и синего (электронно-лучевая трубка дисплея имеет 3 электронные пушки, отвечающие этим цветам). Кроме того, допустимы более яркие цвета. 4 бита задают комбинацию 3х основных цветов и повышенной яркости. Образуется 2**4=16 цветов:

```
              I R G B   номер цвета
BLACK         0 0 0 0    0    черный
BLUE          0 0 0 1    1    синий
GREEN         0 0 1 0    2    зеленый
CYAN          0 0 1 1    3    цианoвый (серо-голубой)
RED           0 1 0 0    4    красный
MAGENTA       0 1 0 1    5    малиновый
BROWN         0 1 1 0    6    коричневый
LIGHTGRAY     0 1 1 1    7    светло-серый (темно-белый)

DARKGRAY      1 0 0 0    8    темно-серый
LIGHTBLUE     1 0 0 1    9    светло-синий
LIGHTGREEN    1 0 1 0   10    светло-зеленый
LIGHTCYAN     1 0 1 1   11    светло-цианoвый
LIGHTRED      1 1 0 0   12    ярко-красный
LIGHTMAGENTA  1 1 0 1   13    ярко-малиновый
YELLOW        1 1 1 0   14    желтый
WHITE         1 1 1 1   15    (ярко)-белый
```

Физический адрес видеопамяти **IBM PC** в цветном алфавитно-цифровом режиме (80x25, 16 цветов) равен 0xB800:0x0000. В **MS DOS** указатель на эту память можно получить при помощи макроса *make far pointer*: **MK_FP** (это должен быть **far** или **huge** указатель!). В **XENIX**† указатель получается при помощи системного вызова **ioctl**, причем система предоставит вам виртуальный адрес, ибо привелегия работы с физическими адресами в **UNIX** принадлежит только системе. Работу с экраном в **XENIX** вы можете увидеть в примере "осыпающиеся буквы".

8.1.

```
/*#! /bin/cc  fall.c -o fall -lx
 *      "Осыпающиеся буквы".
 *      Использование видеопамяти IBM PC в ОС XENIX.
 *      Данная программа иллюстрирует доступ к экрану
 *      персонального компьютера как к массиву байт;
 *      все изменения в массиве немедленно отображаются на экране.
 *          Функция nap() находится в библиотеке -lx
 *      Показана также работа с портами IBM PC при помощи ioctl().
 */
#include <stdio.h>
#include <fcntl.h>         /* O_RDWR */
#include <signal.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/at_ansi.h>
#include <sys/kd.h>        /* for System V/4 and Interactive UNIX only */
/*#include <sys/machdep.h>    for XENIX and SCO UNIX only */
#include <sys/sysmacros.h>

#ifdef M_I386
# define far    /* на 32-битной машине far не требуется */
#endif

char far *screen;   /* видеопамять как массив байт */
                    /* far - "длинный" (32-битный) адрес(segment,offset) */
int     segm;       /* сегмент с видеопамятью */

#define COLS  80        /* число колонок на экране */
#define LINES 25        /* число строк */

#define DELAY 20        /* задержка (миллисекунд) */
int  ega;               /* дескриптор для доступа к драйверу EGA */
```

---

† **XENIX** - (произносится "зиникс") версия **UNIX** для **IBM PC**, первоначально разработанная фирмой **Microsoft** и поставляемая фирмой **Santa Cruz Operation** (**SCO**).

```
/* структура для обмена с портами */
static struct port_io_struct    PORT[ 4 /* не более 4 за раз */] = {
       /* операция       номер порта   данные */
       /*   .dir           .port        .data */

 /* Переустановить flip/flop:
  * заставить порт 0x3C0 ожидать пары адрес/значение
  * при последовательной записи байтов в этот порт.
  */
    {   IN_ON_PORT,     0x3DA,       -1               },
       /* IN-чтение */
 /* Теперь 3c0 ожидает пары адрес/значение */
    {   OUT_ON_PORT,    0x3C0,       -1 /* адрес */   },
    {   OUT_ON_PORT,    0x3C0,       -1 /* значение*/ },
       /* OUT-запись */
 /* переинициализировать дисплей, установив бит #5 порта 3c0 */
    {   OUT_ON_PORT,    0x3C0,       0x20             }
};

void closescr(nsig){             /* конец работы */
        setbgcolor(0);  /* установить черный фон экрана */
        exit(0);
}

/* получение доступа к видеопамяти адаптера VGA/EGA/CGA */
void openscr () {
    static struct videodev {
        char *dev; int mapmode;
    } vd[] = {
        { "/dev/vga", MAPVGA },
        { "/dev/ega", MAPEGA },
        { "/dev/cga", MAPCGA },
        { NULL,       -1     }
    }, *v; /* устройство для доступа к видеоадаптеру */
    for(v=vd; v->dev;v++ )
        if((ega = open (v->dev, O_RDWR)) >= 0 ) goto ok;
    fprintf( stderr, "Can't open video adapter\n" );
    exit(1);

ok:
    /* fprintf(stderr, "Adapter:%s\n", v->dev); */
    /* получить адрес видеопамяти и доступ к ней */
#ifdef M_I386
    screen = (char *) ioctl (ega, v->mapmode, 0);
#else
    segm = ioctl (ega, v->mapmode, 0);
    screen = sotofar (segm, 0); /* (segment,offset) to far pointer */
#endif
    signal( SIGINT, closescr );
}

/* макросы для доступа к байтам "символ" и "атрибуты"
 * в координатах (x,y) экрана.
 */
#define GET(x,y)         screen[ ((x) + (y) * COLS ) * 2 ]
#define PUT(x,y, c)      screen[ ((x) + (y) * COLS ) * 2 ]  = (c)

#define GETATTR(x,y)         screen[ ((x) + (y) * COLS ) * 2 + 1 ]
#define PUTATTR(x,y, a)      screen[ ((x) + (y) * COLS ) * 2 + 1 ] = (a)

/* символ изображается как черный пробел ? */
#define white(c,a) ((isspace(c) || c==0) && (attr & 0160)==0)
```

```
/* установить цвет фона экрана */
void setbgcolor( color ){
     PORT[1].data = 0;  /* регистр номер 0 палитры содержит цвет фона */
     /* всего в палитре 16 регистров (0x00...0xFF) */

     PORT[2].data = color ;
     /* новое значение цвета, составленное как битовая маска
      *  RGBrgb  (r- красный, g- зеленый, b- синий, RGB- дополнительные
      *  тусклые цвета)
      */

     /* выполнить обмены с портами */
     if( ioctl( ega, EGAIO, PORT ) < 0 ){
           fprintf( stderr, "Can't out port\n" );
           perror( "out" );
     }
}

void main(ac, av) char **av;{
        void fall();

        openscr();
        if( ac == 1 ){
            setbgcolor(020);    /* темно-зеленый фон экрана */
            fall();             /* осыпание букв */
        } else {
            if(*av[1] == 'g')
/* Установить режим адаптера graphics 640x350 16-colors */
                ioctl( ega, SW_CG640x350, NULL);
/* Если вы хотите получить адрес видеопамяти в графическом режиме,
 * вы должны СНАЧАЛА включить этот режим,
 * ЗАТЕМ сделать screen=ioctl(ega, v->mapmode, NULL);
 * и ЕЩЕ РАЗ сделать включение графического режима.
 */
/* Установить режим адаптера text 80x25 16-colors       */
            else ioctl( ega, SW_ENHC80x25, NULL);
        }
        closescr(0);
}

/* осыпать буквы вниз */
void fall(){
        register i, j;
        int rest;
        int nextcol;
        int n;
        int changed = 1;        /* не 0, если еще не все буквы опали */
        char mask [ COLS ];

        while( changed ){
           changed = 0;
           for( i = 0 ; i < COLS ; i++ )
                mask[ i ] = 0;

           for( i = 0 ; i < COLS ; i++ ){
                rest = COLS - i;   /* осталось осыпать колонок */
                nextcol = rand() % rest;
```

```
                j = 0;  /* индекс в mask */
                n = 0;  /* счетчик */

                for(;;){
                        if( mask[j] == 0 ){
                                if( n == nextcol ) break;
                                n++;
                        } j++;
                }

                changed += fallColumn( j );
                mask[j] = 1;
           }
        }
}

/* осыпать буквы в одном столбце */
int fallColumn( x ){
        register int y;
        char ch, attr;
        int firstspace = (-1);
        int firstnospace = (-1);

Again:
        /* find the falled array */
        for( y=LINES-1; y >= 0 ; y-- ){

                ch = GET( x, y );
                attr = GETATTR( x,y );

                if( white(ch, attr)){
                    firstspace = y;
                    goto FindNoSpace;
                }
        }

AllNoSpaces:
        return 0;       /* ничего не изменилось */
FindNoSpace:            /* найти не пробел */
        for( ; y >= 0 ; y-- ){

                ch = GET( x, y );
                attr = GETATTR( x, y );

                if( !white(ch, attr)){
                        firstnospace = y;
                        goto Fall;
                }
        }
```

```
AllSpaces:       /* в данном столбце все упало */
        return 0;
Fall:
        /* "уронить" букву */
        for( y = firstnospace ; y < firstspace ; y++ ){
                /* переместить символ на экране на одну позицию вниз */
                ch   = GET( x, y );
                attr = GETATTR( x, y );

                PUT( x, y, 0 );
                PUTATTR( x, y, 0 );

                PUT( x, y+1 , ch );
                PUTATTR( x, y+1, attr );

                nap( DELAY );   /* подождать DELAY миллисекунд */
        }
        return 1;               /* что-то изменилось */
}
```

8.2. Для работы может оказаться более удобным иметь указатель на видеопамять как на массив структур. Приведем пример для системы **MS DOS**:

```
#include <dos.h>  /* там определено MK_FP */
char far *screen =
         MK_FP(0xB800 /*сегмент*/, 0x0000 /*смещение*/);
struct symb{
        char chr; char attr;
} far *scr, far *ptr;
#define COLS  80        /* число колонок */
#define LINES 25        /* число строк   */
#define SCR(x,y)   scr[(x) + COLS * (y)]
/* x из 0..79, y из 0..24 */

void main(){
  int x, y;
  char c;
  scr = (struct symb far *) screen;
  /* или сразу
   * scr = (struct symb far *) MK_FP(0xB800,0x0000);
   */

  /* переписать строки экрана справа налево */
  for(x=0; x < COLS/2; x++ )
    for( y=0; y < LINES; y++ ){
         c = SCR(x,y).chr;
         SCR(x,y).chr = SCR(COLS-1-x, y).chr;
         SCR(COLS-1-x, y).chr = c;
    }

  /* сделать цвет экрана: желтым по синему */
  for(x=0; x < COLS; x++)
    for(y=0; y < LINES; y++)
         SCR(x,y).attr = (0xE | (0x1 << 4));
                    /* желтый + синий фон */

  /* прочесть любую кнопку с клавиатуры (пауза) */
  (void) getch();
}
```

И, наконец, еще удобнее работа с видеопамятью как с двумерным массивом структур:

```
#include <dos.h>  /* MS DOS */
#define COLS 80
#define LINES 25
struct symb {
       char chr; char attr;
} (far *scr)[ COLS ] = MK_FP(0xB800, 0);

void main(void){
     register x, y;
     for(y=0; y < LINES; y++)
         for(x=0; x < COLS; ++x){
             scr[y][x].chr = '?';
             scr[y][x].attr = (y << 4) | (x & 0xF);
         }
     getch();
}
```

Учтите, что при работе с экраном через видеопамять, курсор не перемещается! Если в обычной работе с экраном текст выводится в позиции курсора и курсор *автоматически* продвигается, то здесь курсор будет оставаться на своем прежнем месте. Для перемещения курсора в нужное вам место, вы должны его поставить *явным образом* по окончании записи в видеопамять (например, обращаясь к портам видеоконтроллера).

Обратите внимание, что спецификатор модели памяти **far** должен указываться перед КАЖДЫМ указателем (именно для иллюстрации этого в первом примере описан неиспользуемый указатель *ptr*).

8.3. Составьте программу сохранения содержимого экрана **IBM PC** (видеопамяти) в текстовом режиме в файл и обратно (в системе **XENIX**).

8.4. Пользуясь прямым доступом в видеопамять, напишите функции для спасения прямоугольной области экрана в массив и обратно. Вот функция для спасения в массив:

```
typedef struct {
   short xlen, ylen;
   char  *save;
} Pict;
extern void *malloc(unsigned);

Pict *gettext (int x, int y, int xlen, int ylen){
   Pict *n   = (Pict *) malloc(sizeof *n);
   register char *s; register i, j;

   n->xlen = xlen; n->ylen = ylen;
   s = n->save = (char *) malloc( 2 * xlen * ylen );
   for(i=y; i < y+ylen; i++)
        for(j=x; j < x+xlen; j++){
            *s++ = SCR(j,i).chr ;
            *s++ = SCR(j,i).attr;
        }
   return n;
}
```

Добавьте проверки на корректность *xlen*, *ylen* (в пределах экрана). Напишите функцию **puttext** для вывода спасенной области обратно; функцию **free**(*buf*) лучше в нее не вставлять.

```
void puttext (Pict *n, int x, int y){
   register char *s = n->save;
   register i, j;
   for(i=y; i < y + n->ylen; i++)
        for(j=x; j < x + n->xlen; j++){
            SCR(j,i).chr  = *s++;
            SCR(j,i).attr = *s++;
        }
}
/* очистка памяти текстового буфера */
void deltext(Pict *n){ free(n->save); free(n); }
```

Приведем еще одну полезную функцию, которая может вам пригодиться - это аналог **printf** при прямой работе с видеопамятью.

```
#include <stdarg.h>
/* текущий цвет: белый по синему */
static char currentColor = 0x1F;

int videoprintf (int x, int y, char *fmt, ...){
    char buf[512], *s;
    va_list var;

    /* clipping (отсечение по границам экрана) */
    if( y < 0 || y >= LINES ) return x;

    va_start(var, fmt);
    vsprintf(buf, fmt, var);
    va_end(var);

    for(s=buf; *s; s++, x++){
        /* отсечение */
        if(x < 0    ) continue;
        if(x >= COLS) break;
        SCR(x,y).chr = *s;
        SCR(x,y).attr = currentColor;
    }
    return x;
}
void setcolor (int col){ currentColor = col; }
```

8.5. Пользуясь написанными функциями, реализуйте функции для "выскакивающих" окон (*pop-up* window):

```
Pict *save;
  save = gettext (x,y,xlen,ylen);

  // ... рисуем цветными пробелами прямоугольник с
  // углами (x,y) вверху-слева и (x+xlen-1,y+ylen-1)
  // внизу-справа...

  // ...рисуем некие таблицы, меню, текст в этой зоне...

  // стираем нарисованное окно, восстановив то изображение,
  // поверх которого оно "всплыло".
  puttext (save,x,y);
  deltext (save);
```

Для начала напишите "выскакивающее" окно с сообщением; окно должно исчезать по нажатию любой клавиши.

```
c = message(x, y,   text);
```

Размер окна вычисляйте по длине строки *text*. Код клавиши возвращайте в качестве значения функции.

Теперь сделайте *text* массивом строк: char **text*[]; (последняя строка - NULL).

8.6. Сделайте так, чтобы "выскакивающие" окна имели тень. Для этого надо сохранить в некоторый буфер атрибуты символов (сами символы не надо!), находящихся на местах $:

```
##########
##########$
##########$
 $$$$$$$$$
```

а затем прописать этим символам на экране атрибут 0x07 (белый по черному). При стирании окна (**puttext**-ом) следует восстановить спасенные атрибуты этих символов (стереть тень). Если окно имеет размер *xlen*ylen*, то размер буфера равен *xlen+ylen-1* байт.

8.7. Напишите функцию, рисующую на экране прямоугольную рамку. Используйте ее для рисования рамки окна.

8.8. Напишите "выскакивающее" окно, которое проявляется на экране как бы расширяясь из точки:

Вам следует написать функцию **box**(*x*,*y*,*width*,*height*), рисующую цветной прямоугольник с верхним левым углом (*x*,*y*) и размером (*width*,*height*). Пусть конечное окно задается углом (*x0*,*y0*) и размером (*W*,*H*). Тогда "вырастание" окна описывается таким алгоритмом:

```
void zoom(int x0, int y0, int W, int H){
    int x, y, w, h, hprev; /* промежуточное окно */
    for(hprev=0, w=1; w < W; w++){
        h = H * w; h /= W;  /* W/H == w/h */
        if(h == hprev) continue;
        hprev = h;
        x = x0 + (W - w)/2; /* чтобы центры окон */
        y = y0 + (H - h)/2; /* совпадали         */
        box(x, y, w, h);
        delay(10);      /* задержка 10 миллисек. */
    }
    box(x0, y0, W, H);
}
```

8.9. Составьте библиотеку функций, аналогичных библиотеке **curses**, для ЭВМ **IBM PC** в ОС **XENIX**. Используйте прямой доступ в видеопамять.

8.10. Напишите рекурсивное решение задачи "ханойские башни" (перекладывание дисков: есть три стержня, на один из них надеты диски убывающего к вершине диаметра. Требуется переложить их на третий стержень, никогда не кладя диск большего диаметра поверх диска меньшего диаметра). Усложнение - используйте пакет **curses** для изображения перекладывания дисков на экране терминала. Указание: идея рекурсивного алгоритма:

```
carry(n, from, to, by) = if( n > 0 ){
                carry( n-1, from, by, to );
                перенесиОдинДиск( from, to );
                carry( n-1, by,   to, from );
            }
Вызов:     carry( n, 0, 1, 2 );
n    - сколько дисков перенести (n > 0).
from - откуда (номер стержня).
to   - куда.
by   - при помощи (промежуточный стержень).
```

n дисков потребуют (2**n*)-1 переносов.

8.11. Напишите программу, ищущую выход из лабиринта ("червяк в лабиринте"). Лабиринт загружается из файла *.maze* (не забудьте про расширение табуляций!). Алгоритм имеет рекурсивную природу и выглядит примерно так:

```
#include <setjmp.h>
jmp_buf jmp; int found = 0;

maze(){ /* Это головная функция */
    if( setjmp(jmp) == 0 ){ /* начало */
        if( неСтенка(x_входа, y_входа))
            GO( x_входа, y_входа);
    }
}
GO(x, y){       /* пойти в точку (x, y) */
    if( этоВыход(x, y)){ found = 1;  /* нашел выход */
        пометить(x, y); longjmp(jmp, 1);}
    пометить(x, y);
    if( неСтенка(x-1,y)) GO(x-1, y);  /* влево */
    if( неСтенка(x,y-1)) GO(x, y-1);  /* вверх */
    if( неСтенка(x+1,y)) GO(x+1, y);  /* вправо */
```

```
    if( неСтенка(x,y+1)) GO(x, y+1);  /* вниз   */
    снятьПометку(x, y);
}
#define пометить(x, y)      лабиринт[y][x] = '*'
#define снятьПометку(x, y) лабиринт[y][x] = ' '
#define этоВыход(x, y)    (x == x_выхода && y == y_выхода)
/* можно искать "золото":  (лабиринт[y][x] == '$') */

неСтенка(x, y){ /* стенку изображайте символом @ или # */
  if( координатыВнеПоля(x, y)) return 0; /*край лабиринта*/
  return (лабиринт[y][x] == ' ');
}
```

Отобразите массив *лабиринт* на видеопамять (или воспользуйтесь **curses**-ом). Вы увидите червяка, ползающего по лабиринту в своих исканиях.

8.12. Используя библиотеку **termcap** напишите функции для:

- очистки экрана.
- позиционирования курсора.
- включения/выключения режима выделения текста инверсией.

8.13. Используя написанные функции, реализуйте программу выбора в меню. Выбранную строку выделяйте инверсией фона.

```
/*#!/bin/cc termio.c -O -o termio -ltermcap
 * Смотри   man termio, termcap и screen.
 *              Работа с  терминалом в стиле System-V.
 *              Работа с  системой команд терминала через /etc/termcap
 *              Работа со временем.
 *              Работа с  будильником.
 */

#include <stdio.h>              /* standard input/output */
#include <sys/types.h>          /* system typedefs */
#include <termio.h>             /* terminal input/output */
#include <signal.h>             /* signals */
#include <fcntl.h>              /* file control */
#include <time.h>               /* time structure */

void setsigs(), drawItem(), drawTitle(), prSelects(), printTime();

/* Работа с описанием терминала TERMCAP ---------------------------------*/
extern char *getenv ();         /* получить переменную окружения         */
extern char *tgetstr ();        /* получить строчный описатель /termcap/ */
extern char *tgoto ();          /* подставить %-параметры      /termcap/ */

static char Tbuf[2048],         /* буфер для описания терминала, обычно 1024 */
  /* Tbuf[] можно сделать локальной автоматической переменной
   * в функции tinit(), чтобы не занимать место */
           Strings[256],       /* буфер для расшифрованных описателей */
          *p;                   /* вспомогательная перем.              */
char   *tname;                  /* название типа терминала             */
int     COLS,                  /* число колонок экрана               */
        LINES;                  /* число строк экрана                  */
char   *CM;                     /* описатель: cursor motion            */
char   *CL;                     /* описатель: clear screen             */
char   *CE;                     /* описатель: clear end of line        */
char   *SO,
       *SE;                     /* описатели: standout Start и End     */
char   *BOLD,
       *NORM;                   /* описатели: boldface and NoStandout  */
int    BSflag;                  /* можно использовать back space '\b'  */
```

```
void tinit () {      /* Функция настройки на систему команд дисплея */
    p = Strings;
    /* Прочесть описание терминала в Tbuf */
    switch (tgetent (Tbuf, tname = getenv ("TERM"))) {
         case -1:
            printf ("Нет файла TERMCAP (/etc/termcap).\n");
            exit (1);
        case 0:
            printf ("Терминал %s не описан.\n", tname);
            exit (2);
        case 1:
            break;              /* OK */
    }
    COLS =  tgetnum ("co");     /* Прочесть числовые описатели. */
    LINES = tgetnum ("li");

    CM = tgetstr ("cm", &p);    /* Прочесть строчные описатели.      */
    CL = tgetstr ("cl", &p);    /* Описатель дешифруется и заносится */
    CE = tgetstr ("ce", &p);    /* в массив по адресу p. Затем       */
    SO = tgetstr ("so", &p);    /* указатель p продвигается на       */
    SE = tgetstr ("se", &p);    /* свободное место, а адрес расшиф-  */
    BOLD = tgetstr ("md", &p);  /* рованной строки выдается из ф-ции */
    NORM = tgetstr ("me", &p);

    BSflag = tgetflag( "bs" );  /* Узнать значение флажка:
             1 - есть, 0 - нет   */
}

/* Макрос, внесенный в функцию.
   Дело в том, что tputs в качестве третьего аргумента
   требует имя функции, которую она вызывает в цикле: (*f)(c);
   Если подать на вход макрос, вроде putchar,
   а не адрес входа в функцию, мы
   и не достигнем желанного эффекта,
   и получим ругань от компилятора.
*/
void put (c) char c;
{   putchar (c);  }

/* очистить экран */
void clearScreen () {
    if (CL == NULL)      /* Функция tputs() дорасшифровывает описатель */
        return;          /* (обрабатывая задержки) и выдает его        */
    tputs (CL, 1, put);  /* посимвольно ф-цией put(c) 1 раз            */
    /* Можно выдать команду не 1 раз, а несколько: например если это   */
    /* команда сдвига курсора на 1 позицию влево '\b'                  */
}

/* очистить конец строки, курсор остается на месте */
void clearEOL () {  /* clear to the end of line */
    if (CE == NULL)
        return;
    tputs (CE, 1, put);
}

/* позиционировать курсор */
void gotoXY (x, y) {  /* y - по вертикали СВЕРХУ-ВНИЗ. */
    if (x < 0 || y < 0 || x >= COLS || y >= LINES) {
        printf ("Точка (%d,%d) вне экрана\n", x, y);
        return;
    }
    /* CM - описатель, содержащий 2 параметра. Подстановку параметров
     * делает функция tgoto() */
    tputs (tgoto (CM, x, y), 1, put);
}
```

```
/* включить выделение */
void standout () {
    if (SO) tputs (SO, 1, put);
}

/* выключить выделение */
void standend () {
    if (SE) tputs (SE, 1, put);
    /* else normal(); */
}

/* включить жирный шрифт */
void bold () {
    if (BOLD) tputs (BOLD, 1, put);
}

/* выключить любой необычный шрифт */
void normal () {
    if (NORM) tputs (NORM, 1, put);
    else      standend();
}
```

```
/* Управление драйвером терминала ---------------------------------- */

#define ESC '\033'
#define ctrl(c)         ((c) & 037 )

int     curMode = 0;
int     inited = 0;

struct termio   old,
                new;
int     fdtty;

void ttinit () {
 /* открыть терминал в режиме "чтение без ожидания" */
    fdtty = open ("/dev/tty", O_RDWR | O_NDELAY);

 /* узнать текущие режимы драйвера */
    ioctl (fdtty, TCGETA, &old);

    new = old;

 /* input flags */
 /* отменить преобразование кода '\r' в '\n' на вводе              */
    new.c_iflag &= ~ICRNL;
    if ((old.c_cflag & CSIZE) == CS8)  /* 8-битный код             */
         new.c_iflag &= ~ISTRIP;       /* отменить & 0177 на вводе */

 /* output flags */
 /* отменить TAB3 - замену табуляций '\t' на пробелы               */
 /* отменить ONLCR - замену '\n' на пару '\r\n' на выводе          */
    new.c_oflag &= ~(TAB3 | ONLCR);

 /* local flags */
 /* выключить режим ICANON, включить CBREAK                       */
 /* выключить эхоотображение набираемых символов                   */
    new.c_lflag &= ~(ICANON | ECHO);

 /* control chars */      /* при вводе с клавиш ждать не более ... */
    new.c_cc[VMIN]  = 1;  /* 1 символа и */
    new.c_cc[VTIME] = 0;  /* 0 секунд    */
    /* Это соответствует режиму CBREAK */

  /* Символы, нажатие которых заставляет драйвер терминала послать сигнал
   * либо отредактировать набранную строку. Значение 0 означает,
   * что соответствующего символа не будет */
    new.c_cc[VINTR]  = ctrl ('C'); /* символ, генерящий SIGINT         */
    new.c_cc[VQUIT]  = '\0';       /* символ, генерящий SIGQUIT        */
    new.c_cc[VERASE] = '\0';       /* забой (отмена последнего символа)*/
    new.c_cc[VKILL]  = '\0';       /* символ отмены строки              */
  /* По умолчанию эти кнопки равны: DEL, CTRL/\, BACKSPACE, CTRL/U     */

    setsigs ();
    inited = 1;                   /* уже инициализировано */
}

void openVisual () {      /* open visual mode (включить "экранный" режим) */
    if (!inited)
        ttinit ();
    if (curMode == 1)
        return;

    /* установить моды драйвера из структуры new */
    ioctl (fdtty, TCSETAW, &new);
    curMode = 1;        /* экранный режим */
}
```

```
void closeVisual () {        /* canon mode (включить канонический режим) */
    if (!inited)
        ttinit ();
    if (curMode == 0)
        return;

    ioctl (fdtty, TCSETAW, &old);
    curMode = 0;        /* канонический режим */
}

/* завершить процесс */
void die (nsig) {
    normal();
    closeVisual (); /* При завершении программы (в том числе по
       * сигналу) мы должны восстановить прежние режимы драйвера,
       * чтобы терминал оказался в корректном состоянии. */
    gotoXY (0, LINES - 1);
    putchar ('\n');
    if (nsig)
        printf ("Пришел сигнал #%d\n", nsig);
    exit (nsig);
}

void setsigs () {
    register    ns;

    /* Перехватывать все сигналы; завершаться по ним. */
    /* UNIX имеет 15 стандартных сигналов. */
    for (ns = 1; ns <= 15; ns++)
        signal (ns, die);
}

/* Работа с меню ----------------------------------------- */

struct menu {
    char   *m_text;             /* выдаваемая строка */
    int     m_label;            /* помечена ли она ? */
}           menuText[] = {
        /* названия песен Beatles */
    {       "Across the Universe", 0            } ,
    {       "All I've got to do",  0            } ,
    {       "All my loving",       0            } ,
    {       "All together now",    0            } ,
    {       "All You need is love",0            } ,
    {       "And I love her",      0            } ,
    {       "And your bird can sing", 0         } ,
    {       "Another girl",        0            } ,
    {       "Any time at all",     0            } ,
    {       "Ask me why",          0            } ,
    {       NULL,                  0            }
};
```

```
#define Y_TOP 6
int     nitems;                 /* количество строк в меню */
int     nselected = 0;          /* количество выбранных строк */

char    title[] =
        "ПРОБЕЛ - вниз, ЗАБОЙ - вверх, ESC - выход, \
ENTER - выбрать, TAB - отменить";

# define TIMELINE 1

void main (ac, av) char **av; {
    char  **line;
    register    i;
    int     c;
    int     n;                  /* текущая строка */
    extern char readkey ();     /* forward */

    extern char *ttyname ();    /* имя терминала */
    char   *mytty;

    extern char *getlogin ();   /* имя пользователя */
    char   *userName = getlogin ();

    srand (getpid () + getuid ());      /* инициализировать
                                         * датчик случайных чисел */
    /* считаем строки меню */
    for (nitems = 0; menuText[nitems].m_text != NULL; nitems++);

    /* инициализируем терминал */
    tinit (); ttinit();
    mytty = ttyname(fdtty);
    openVisual ();

again:
    clearScreen ();
    if (mytty != NULL && userName != NULL) {
        gotoXY (0, TIMELINE);
        bold ();
        printf ("%s", userName);
        normal ();
        printf (" at %s (%s)", mytty, tname);
    }

    drawTitle ("",    Y_TOP - 4);
    drawTitle (title, Y_TOP - 3);
    drawTitle ("",    Y_TOP - 2);

 /* рисуем меню */
    for (i = 0; i < nitems; i++) {
        drawItem (i, 20, Y_TOP + i, 0);
    }

    /* цикл перемещений по меню */
    for (n=0; ; ) {
        printTime ();   /* выдаем текущее время */
        drawItem (n, 20, Y_TOP + n, 1);

        c = getcharacter ();

        drawItem (n, 20, Y_TOP + n, 0);
```

```
            switch (c) {
                case ' ':
            go_down:
                    n++;
                    if (n == nitems)
                        n = 0;
                    break;

                case '\b': case 0177:
                    n--;
                    if (n < 0)
                        n = nitems - 1;
                    break;

                case ESC:
                    goto out;

                case '\t':          /* Unselect item */
                    if (menuText[n].m_label != 0) {
                        menuText[n].m_label = 0;
                        drawItem (n, 20, Y_TOP + n, 0);
                        nselected--;
                        prSelects ();
                    }
                    goto go_down;

                case '\r':          /* Select item */
                case '\n':
                    bold ();
                    drawTitle (menuText[n].m_text, LINES - 2);
                                    /* last but two line */
                    normal ();

                    if (menuText[n].m_label == 0) {
                        menuText[n].m_label = 1;
                        drawItem (n, 20, Y_TOP + n, 0);
                        nselected++;
                        prSelects ();
                    }
                    goto go_down;

                default:
                    goto go_down;
            }
        }

    out:
        clearScreen ();

        gotoXY (COLS / 3, LINES / 2);
        bold ();
        printf ("Нажми любую кнопку.");
        normal ();

        /* замусорить экран */
        while (!(c = readkey ())) {
            /* случайные точки */
            gotoXY (rand () % (COLS - 1), rand () % LINES);
            putchar ("@.*"[rand () % 3]);   /* выдать символ */
            fflush (stdout);
        }
```

```
    standout ();
    printf ("Нажата кнопка с кодом 0%o\n", c & 0377);
    standend ();

    if (c == ESC) {
        sleep (2);              /* подождать 2 секунды */
        goto again;
    }

    die (0);                    /* успешно завершиться,
                                 * восстановив режимы драйвера */
}

/* Нарисовать строку меню номер i
 * в координатах (x,y) с или без выделения
 */
void drawItem (i, x, y, out) {
    gotoXY (x, y);
    if (out) {
        standout ();
        bold ();
    }
    printf ("%c %s ",
            menuText[i].m_label ? '-' : ' ',  /* помечено или нет */
            menuText[i].m_text          /* сама строка */
        );

    if (out) {
        standend ();
        normal ();
    }
}

/* нарисовать центрированную строку в инверсном изображении */
void drawTitle (title, y) char  *title; {
    register int    n;
    int     length = strlen (title);    /* длина строки */

    gotoXY (0, y);
 /* clearEOL(); */
    standout ();

    for (n = 0; n < (COLS - length) / 2; n++)
        putchar (' ');

    printf ("%s", title); n += length;

 /* дорисовать инверсией до конца экрана */
    for (; n < COLS - 1; n++)
        putchar (' ');
    standend ();
}

/* выдать общее число выбранных строк */
void prSelects () {
    char    buffer[30];

    if (nselected == 0) {
        gotoXY (0, LINES - 1);
        clearEOL ();
    }
    else {
        sprintf (buffer, "Выбрано: %d/%d", nselected, nitems);
        drawTitle (buffer, LINES - 1);
    }
}
```

```
/* Работа с будильником ------------------------- */

#define PAUSE 4
int     alarmed;                /* флаг будильника */

/* реакция на сигнал "будильник" */
void onalarm (nsig) {
    alarmed = 1;
}

/* Прочесть символ с клавиатуры, но не позже чем через PAUSE секунд.
 * иначе вернуть код 'пробел'.
 */
int getcharacter () {
    int     c;

    fflush(stdout);

 /* заказать реакцию на будильник */
    signal (SIGALRM, onalarm);
    alarmed = 0;                /* сбросить флаг */

 /* заказать сигнал "будильник" через PAUSE секунд */
    alarm (PAUSE);

 /* ждать нажатия кнопки.
  * Этот оператор завершится либо при нажатии кнопки,
  * либо при получении сигнала.
  */
    c = getchar ();

 /* проверяем флаг */
    if (!alarmed) {             /* был нажат символ */
        alarm (0);              /* отменить заказ будильника */
        return c;
    }

 /* был получен сигнал "будильник" */
    return ' ';                 /* продвинуть выбранную строку вниз */
}

/* ---- NDELAY read ---------------------------- */

/* Вернуть 0 если на клавиатуре ничего не нажато,
 * иначе вернуть нажатую кнопку
 */
char    readkey () {
    char    c;
    int     nread;

    nread = read (fdtty, &c, 1);
    /* обычный read() дожидался бы нажатия кнопки.
     * O_NDELAY позволяет не ждать, но вернуть "прочитано 0 символов".
     */
    return (nread == 0) ? 0 : c;
}
```

```
/* -------- Работа со временем ----------------------- */
void printTime () {
    time_t  t;                  /* текущее время */
    struct  tm  *tm;
    extern  struct tm   *localtime ();
    char    tmbuf[30];
    static  char *week[7]   = { "Вс",  "Пн",  "Вт",  "Ср",  "Чт",  "Пт", "Сб" };
    static  char *month[12] = { "Янв", "Фев", "Мар", "Апр", "Май", "Июн",
                                "Июл", "Авг", "Сен", "Окт", "Ноя", "Дек"      };

    time (&t);                  /* узнать текущее время */
    tm = localtime (&t);        /* разложить его на компоненты */

    sprintf (tmbuf, "%2s %02d:%02d:%02d %02d-%3s-%d",
            week[tm -> tm_wday],  /* день недели (0..6)   */
            tm -> tm_hour,        /* часы   (0..23)       */
            tm -> tm_min ,        /* минуты (0..59)       */
            tm -> tm_sec ,        /* секунды (0..59)      */
            tm -> tm_mday,        /* число месяца (1..31) */
            month[tm -> tm_mon],  /* месяц (0..11)        */
            tm -> tm_year + 1900  /* год                   */
        );

    gotoXY (COLS / 2, TIMELINE);
    clearEOL ();

    gotoXY (COLS - strlen (tmbuf) - 1, TIMELINE);
    bold ();
    printf ("%s", tmbuf);
    normal ();
}
```

8.14. Напишите программу, выдающую файл на экран порциями по 20 строк и ожидающую нажатия клавиши. Усложнения:

a) добавить клавишу для возврата к началу файла.

b) используя библиотеку **termcap**, очищать экран перед выдачей очередной порции текста.

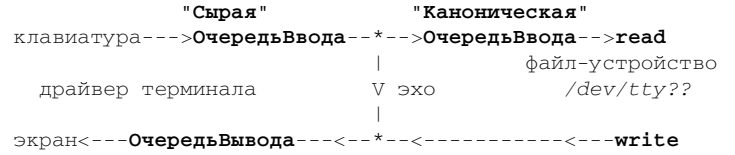
c) напишите эту программу, используя библиотеку **curses**.

d) используя **curses**, напишите программу параллельного просмотра 2-х файлов в 2-х неперекрывающихся окнах.

e) то же в перекрывающихся окнах.

8.15. Напишите функции включения и выключения режима эхо-отображения набираемых на клавиатуре символов (**ECHO**).

8.16. То же про "режим немедленного ввода" (**CBREAK**). В обычном режиме строка, набранная на клавиатуре, сначала попадает в некоторый буфер в драйвере терминала†.



Этот буфер используется для предчтения - вы можете набирать текст на клавиатуре еще *до* того, как программа запросит его **read**-ом: этот набранный текст сохранится в буфере и при поступлении запроса будет выдан из буфера. Также, в каноническом режиме **ICANON**, буфер ввода используется для редактирования

† Такие буфера носят название "character lists" - **clist**. Существуют "сырой" (raw) clist, в который попадают ВСЕ символы, вводимые с клавиатуры; и "канонический" clist, в котором хранится отредактированная строка - обработаны забой, отмена строки. Сами специальные символы (редактирования и генерации сигналов) в каноническую очередь не попадают (в режиме **ICANON**).

введенной строки: забой отменяет последний набранный символ, **CTRL/U** отменяет всю набранную строку; а также он используется для выполнения некоторых преобразований символов на вводе и выводе£.

Введенная строка попадает в программу (которая запросила данные с клавиатуры при помощи **read**, **gets**, **putchar**) только после того, как вы нажмете кнопку <**ENTER**> с кодом '**n**'. До этого вводимые символы накапливаются в буфере, но в программу не передаются - программа тем временем "спит" в вызове **read**. Как только будет нажат символ '**n**', он сам поступит в буфер, а программа будет разбужена и сможет наконец прочесть из буфера ввода набранный текст.

Для меню, редакторов и других "экранных" программ этот режим неудобен: пришлось бы слишком часто нажимать <**ENTER**>. В режиме **CBREAK** нажатая буква *немедленно* попадает в вашу программу (без ожидания нажатия '**n**'). В данном случае буфер драйвера используется только для предчтения, но не для редактирования вводимого текста. Редактирование возлагается на вас - предусмотрите его в своей программе сами!

Заметьте, что код кнопки <**ENTER**> ("конец ввода") - '**n**' - не только "проталкивает" текст в программу, но и сам попадает в буфер драйвера, а затем в вашу программу. Не забывайте его как-то обрабатывать.

В **MS DOS** функция чтения кнопки в режиме ~**ECHO**+**CBREAK** называется **getch**(). В **UNIX** аналогично ей будет работать обычный **getchar**(), если перед его использованием установить нужные режимы драйвера **tty** вызовом **ioctl**. По окончании программы режим драйвера надо восстановить (за вас это никто не сделает). Также следует восстанавливать режим драйвера при аварийном завершении программы (по любому сигналу††).

Очереди ввода и вывода используются также для синхронизации скорости работы программы (скажем, скорости наполнения буфера вывода символами, поступающими из программы через вызовы **write**) и скорости работы устройства (с которой драйвер выбирает символы с другого конца очереди и выдает их на экран); а также для преобразований символов на вводе и выводе. Пример управления всеми режимами есть в приложении.

8.17. *Функциональные* клавиши большинства дисплеев посылают в линию не один, а несколько символов. Например на терминалах, работающих в системе команд стандарта **ANSI**, кнопки со стрелками посылают такие последовательности:

стрелка вверх	"\033[A"	*кнопка Home*	"\033[H"
стрелка вниз	"\033[B"	*кнопка End*	"\033[F"
стрелка вправо	"\033[C"	*кнопка PgUp*	"\033[I"
стрелка влево	"\033[D"	*кнопка PgDn*	"\033[G"

(поскольку первым символом управляющих последовательностей обычно является символ '**033**' (*escape*), то их называют еще *escape-последовательностями*). Нам же в программе удобно воспринимать такую последовательность как *единственный* код с целым значением большим 0xFF. Склейка последовательностей символов, поступающих от функциональных клавиш, в такой внутренний код - также задача экранной библиотеки (учет системы команд дисплея на *вводе*).

Самым интересным является то, что *одиночный* символ '\033' тоже может прийти с клавиатуры - его посылает клавиша **Esc**. Поэтому если мы строим распознаватель клавиш, который при поступлении кода 033 начинает ожидать составную последовательность - мы должны выставлять таймаут, например **alarm**(1); и если по его истечении больше никаких символов не поступило - выдавать код 033 как код клавиши **Esc**.

Напишите распознаватель кодов, поступающих с клавиатуры. Коды обычных букв выдавать как есть (0..0377), коды функциональных клавиш выдавать как числа >= 0400. Учтите, что разные типы дисплеев посылают разные последовательности от одних и тех же функциональных клавиш: предусмотрите настройку на систему команд ДАННОГО дисплея при помощи библиотеки **termcap**. Распознаватель удобно строить при помощи сравнения поступающих символов с ветвями дерева (спускаясь по нужной ветви дерева при поступлении очередного символа. Как только достигли листа дерева - возвращаем код, приписанный этому листу):

£ Режимы преобразований, символы редактирования, и.т.п. управляются системным вызовом **ioctl**. Большой пример на эту тему есть в приложении.

†† Если ваша программа завершилась аварийно и моды терминала остались в "странном" состоянии, то привести терминал в чувство можно командой **stty sane**

```
---> '\033' ---> '[' ---> 'A' --> выдать 0400
            |        \--> 'B' -->        0401
            |        \--> 'C' -->        0402
            |        \--> 'D' -->        0403
            \--> 'X' ---------->        0404
                      ...
```

Нужное дерево стройте при настройке на систему команд данного дисплея.

Библиотека **curses** уже имеет такой встроенный распознаватель. Чтобы составные последовательности склеивались в специальные коды, вы должны установить режим **keypad**:

```
int c; WINDOW *window;
       ...
keypad(window, TRUE);
       ...
c = wgetch(window);
```

Без этого **wgetch**() считывает все символы поодиночке. Символические названия кодов для функциональных клавиш перечислены в <*curses.h*> и имеют вид **KEY_LEFT**, **KEY_RIGHT** и.т.п. Если вы работаете с единственным окном размером с весь экран, то в качестве параметра *window* вы должны использовать стандартное окно **stdscr** (это имя предопределено в include-файле *curses.h*).

```
# ======================================== Makefile для getch
getch: getch.o
        cc getch.o -o getch -ltermlib

getch.o: getch.c getch.h
        cc -g -DUSG -c getch.c

/* Разбор составных последовательностей клавиш с клавиатуры.  */
/* ================================================ getch.h */
#define FALSE   0
#define TRUE    1
#define BOOLEAN unsigned char
#define INPUT_CHANNEL   0
#define OUTPUT_CHANNEL  1

#define KEY_DOWN      0400
#define KEY_UP        0401
#define KEY_LEFT      0402
#define KEY_RIGHT     0403

#define KEY_PGDN      0404
#define KEY_PGUP      0405

#define KEY_HOME      0406
#define KEY_END       0407

#define KEY_BACKSPACE 0410
#define KEY_BACKTAB   0411

#define KEY_DC        0412
#define KEY_IC        0413

#define KEY_DL        0414
#define KEY_IL        0415

#define KEY_F(n)      (0416+n)

#define ESC           ' 33'
```

```
extern char *tgetstr();

void _put(char c);
void _puts(char *s);
void keyboard_access_denied(void);
char *strdup(const char *s);
void keyinit(void);
int getc_raw(void);
void keyreset(void);
int getch(void);
int lgetch(BOOLEAN);
int ggetch(BOOLEAN);
int kgetch(void);
void _sigalrm(int n);
void init_keytry(void);
void add_to_try(char *str, short code);
void keypad_on(void);
void keypad_off(void);
int dotest(void);
void tinit(void);
void main(void);
```

```
/* ================================================== getch.c
 *      The source version of getch.c file was
 *      written by Pavel Curtis.
 *
 */

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <termios.h>
#include <ctype.h>
#include <string.h>
#include <locale.h>
#include "getch.h"

#define keypad_local   S[0]
#define keypad_xmit    S[1]

#define key_backspace  S[2]
#define key_backtab    S[3]

#define key_left       S[4]
#define key_right      S[5]
#define key_up         S[6]
#define key_down       S[7]

#define key_ic         S[8]
#define key_dc         S[9]
#define key_il        S[10]
#define key_dl        S[11]

#define key_f1        S[12]
#define key_f2        S[13]
#define key_f3        S[14]
#define key_f4        S[15]
#define key_f5        S[16]
#define key_f6        S[17]
#define key_f7        S[18]
#define key_f8        S[19]
#define key_f9        S[20]
#define key_f10       S[21]     /*  f0 */
#define key_f11       S[22]     /* f11 */
#define key_f12       S[23]     /* f12 */
#define key_home      S[24]
#define key_end       S[25]
#define key_npage     S[26]
#define key_ppage     S[27]

#define TOTAL 28
```

```
/* descriptors for keys */
char *KEYS[TOTAL+1] = {
        "ke", "ks",

        "kb", "kB",
        "kl", "kr", "ku", "kd",
        "kI", "kD", "kA", "kL",

        "f1", "f2", "f3", "f4", "f5",
        "f6", "f7", "f8", "f9", "f0",
        "f.", "f-",

        "kh", "kH", "kN", "kP",

        NULL

}, *S[TOTAL];

void _put (char c)  { write( INPUT_CHANNEL, &c, 1 ); }
void _puts(char *s) { tputs ( s, 1, _put ); }

static int  _backcnt = 0;
static char _backbuf[30];

static struct try {
        struct try *child;
        struct try *sibling;
        char ch;
        short value;
}       *_keytry;

BOOLEAN keypadok = FALSE;

struct termios new_modes;

void keyboard_access_denied(){ printf( "Клавиатура недоступна.\n" ); exit(1); }
char *strdup(const char *s)  { return strcpy((char *) malloc(strlen(s)+1), s); }
```

```
/* Инициализация таблицы строк */
void keyinit(){
        char *key, nkey[80], *p;
        register i;

        keyreset();
        for( i=0; i < TOTAL; i++ ){
                p = nkey;
                printf("tgetstr(%s)...", KEYS[i]);
                key = tgetstr(KEYS[i], &p);

                if(S[i]) free(S[i]);
                if(key == NULL){
                        S[i] = NULL;   /* No such key */
                        printf("клавиша не определена.\n");
                }else{
                        /* Decrypted string */
                        S[i] = strdup(key);
                        printf("считано.\n");
                }
        }

        init_keytry();
        if( tcgetattr(INPUT_CHANNEL, &new_modes) < 0 ){
                keyboard_access_denied();
        }

        /* input flags */

        /* отменить преобразование кода '\r' в '\n' на вводе */
        new_modes.c_iflag &= ~ICRNL;
        if ((new_modes.c_cflag & CSIZE) == CS8)  /* 8-битный код */
             new_modes.c_iflag &= ~ISTRIP;       /* отменить & 0177 на вводе */

        /* output flags */

        /* отменить TAB3 - замену табуляций '\t' на пробелы */
        /* отменить ONLCR - замену '\n' на пару '\r\n' на выводе */
        new_modes.c_oflag &= ~(TAB3 | ONLCR);

        /* local flags */

        /* выключить режим ICANON, включить CBREAK */
        /* выключить эхоотображение набираемых символов */
        new_modes.c_lflag &= ~(ICANON | ECHO);

        /* control chars */      /* при вводе с клавиш ждать не более ... */
        new_modes.c_cc[VMIN]  = 1;  /* 1 символа и */
        new_modes.c_cc[VTIME] = 0;  /* 0 секунд    */
        /* Это соответствует режиму CBREAK */

        /* Символы, нажатие которых заставляет драйвер терминала послать сигнал
         * либо отредактировать набранную строку. Значение 0 означает,
         * что соответствующего символа не будет */
        new_modes.c_cc[VINTR]  = '\0'; /* символ, генерящий SIGINT  */
        new_modes.c_cc[VQUIT]  = '\0'; /* символ, генерящий SIGQUIT */
        new_modes.c_cc[VERASE] = '\0'; /* забой (отмена последнего символа)*/
        new_modes.c_cc[VKILL]  = '\0'; /* символ отмены строки      */
}
```

```
/* Чтение одного символа непосредственно с клавиатуры */
int getc_raw(){
        int n; char c;

        n = read(INPUT_CHANNEL, &c, 1);
        if (n <= 0) return EOF;
        return (c & 0xFF);
}

static BOOLEAN  _getback  = FALSE;
static char     _backchar = '\0';

/* Чтение символа - либо из буфера (если не пуст), либо с клавиатуры */
#define nextc()       (_backcnt > 0  ?  _backbuf[--_backcnt]          : \
                        _getback      ?  _getback = FALSE, _backchar  : \
                                          getc_raw())

#define putback(ch)   _backbuf[_backcnt++] = ch

void keyreset(){
        _backcnt = 0; _backchar = '\0';
        _getback = FALSE;
}

/* Функция чтения составного символа */
int getch(){
        int c = lgetch(TRUE);
        keypad_off();
        return c;
}

/*
        ВНИМАНИЕ!
                Если в процессе будет получен сигнал,
                в то время как процесс находится внутри вызова getch(),
                то системный вызов read() вернет 0 и errno == EINTR.
                В этом случае getch() вернет '\0'.
                Чтобы избежать этой ситуации используется функция lgetch()
*/
int lgetch(BOOLEAN kpad) {
        int c;

        while((c = ggetch(kpad)) <= 0);
        return c;
}

int ggetch(BOOLEAN kpad) {
        int kgetch();

        if( kpad ) keypad_on();
        else       keypad_off();

        return keypadok ? kgetch() : nextc();
}
```

```
/*
**      int kgetch()
**
**      Get an input character, but take care of keypad sequences, returning
**      an appropriate code when one matches the input.  After each character
**      is received, set a one-second alarm call.  If no more of the sequence
**      is received by the time the alarm goes off, pass through the sequence
**      gotten so far.
**
*/

#define CRNL(c)    (((c) == '\r') ? '\n' : (c))

/* борьба с русской клавиатурой */
#if !defined(XENIX) || defined(VENIX)
# define unify(c) ( (c)&(( (c)&0100 ) ? ~0240 : 0377 ))
#else
# define unify(c) (c)
#endif
```

```
/* ================================================================== */
#if !defined(XENIX) && !defined(USG) && !defined(M_UNIX) && !defined(unix)

        /* Для семейства BSD */

static BOOLEAN   alarmed;
jmp_buf          jbuf;

int kgetch()
{
        register struct try  *ptr;
        int         ch;
        char        buffer[10];     /* Assume no sequences longer than 10 */
        register char        *bufp = buffer;
        void        (*oldsig)();
        void         _sigalrm();

        ptr = _keytry;

        oldsig = signal(SIGALRM, _sigalrm);
        alarmed = FALSE;

        if( setjmp( jbuf )) /* чтоб свалиться сюда с read-а */
                ch = EOF;

        do
        {
            if( alarmed )
                break;
            ch = nextc();
            if (ch != EOF)              /* getc() returns EOF on error, too */
                *(bufp++) = ch;
            if (alarmed)
                break;

            while (ptr != (struct try *)NULL &&
                   (ch == EOF || unify(CRNL(ptr->ch)) != unify(CRNL(ch))  ))
                ptr = ptr->sibling;

            if (ptr != (struct try *)NULL)
            {
                if (ptr->value != 0)
                {
                    alarm(0);
                    signal(SIGALRM, oldsig);
                    return(ptr->value);
                }
                else
                {
                    ptr = ptr->child;
                    alarm(1);
                }
            }

        } while (ptr != (struct try *)NULL);

        alarm(0);
        signal(SIGALRM, oldsig);

        if (ch == EOF && bufp == buffer)
            return ch;
        while (--bufp > buffer)
            putback(*bufp);
        return(*bufp & 0377);
}
```

```
void _sigalrm(int n)
{
        alarmed = TRUE;
        longjmp(jbuf, 1);
}

/* ========================================================================= */
#else   /* XENIX or USG */

        /* Для семейства SYSTEM V */

static  BOOLEAN alarmed;

int kgetch()
{
        register struct try  *ptr;
        int         ch;
        char        buffer[10];     /* Assume no sequences longer than 10 */
        register char        *bufp = buffer;
        void         (*oldsig)();
        void         _sigalrm();

        ptr = _keytry;

        oldsig = signal(SIGALRM, _sigalrm);
        alarmed = FALSE;

        do
        {
            ch = nextc();
            if (ch != EOF)              /* getc() returns EOF on error, too */
                *(bufp++) = ch;
            if (alarmed)
                break;

            while (ptr != (struct try *)NULL &&
                   (ch == EOF || unify(CRNL(ptr->ch)) != unify(CRNL(ch))  ))
                ptr = ptr->sibling;

            if (ptr != (struct try *)NULL)
            {
                if (ptr->value != 0)
                {
                    alarm(0);
                    signal(SIGALRM, oldsig);
                    return(ptr->value);
                }
                else
                {
                    ptr = ptr->child;
                    alarm(1);
                }
            }

        } while (ptr != (struct try *)NULL);

        alarm(0);
        signal(SIGALRM, oldsig);

        if (ch == EOF && bufp == buffer)
            return ch;
        while (--bufp > buffer)
            putback(*bufp);
        return(*bufp & 0377);
}
```

```
void _sigalrm(int n)
{
        alarmed = TRUE;
        signal(SIGALRM, _sigalrm);
}

#endif /*XENIX*/

/* ========================================================================= */
/*
**      init_keytry()
**      Построение дерева разбора последовательностей символов.
**
*/

void init_keytry()
{
        _keytry = (struct try *) NULL;

        add_to_try(key_backspace, KEY_BACKSPACE);
        add_to_try("\b",          KEY_BACKSPACE);
        add_to_try("\177",        KEY_BACKSPACE);

        add_to_try(key_backtab,   KEY_BACKTAB);
        add_to_try(key_dc,        KEY_DC);
        add_to_try(key_dl,        KEY_DL);
        add_to_try(key_down,      KEY_DOWN);

        add_to_try(key_f1,        KEY_F(1));
        add_to_try(key_f2,        KEY_F(2));
        add_to_try(key_f3,        KEY_F(3));
        add_to_try(key_f4,        KEY_F(4));
        add_to_try(key_f5,        KEY_F(5));
        add_to_try(key_f6,        KEY_F(6));
        add_to_try(key_f7,        KEY_F(7));
        add_to_try(key_f8,        KEY_F(8));
        add_to_try(key_f9,        KEY_F(9));
        add_to_try(key_f10,       KEY_F(10));
        add_to_try(key_f11,       KEY_F(11));
        add_to_try(key_f12,       KEY_F(12));
        add_to_try(key_home,      KEY_HOME);
        add_to_try(key_ic,        KEY_IC);
        add_to_try(key_il,        KEY_IL);
        add_to_try(key_left,      KEY_LEFT);
        add_to_try(key_npage,     KEY_PGDN);
        add_to_try(key_ppage,     KEY_PGUP);
        add_to_try(key_right,     KEY_RIGHT);
        add_to_try(key_up,        KEY_UP);
        add_to_try(key_end,       KEY_END);
}
```

```
void add_to_try(char *str, short code)
{
        static BOOLEAN  out_of_memory = FALSE;
        struct try      *ptr, *savedptr;

        if (str == NULL || out_of_memory)
            return;

        if (_keytry != (struct try *) NULL)
        {
            ptr = _keytry;

            for (;;)
            {
                while (ptr->ch != *str  &&  ptr->sibling != (struct try *)NULL)
                    ptr = ptr->sibling;

                if (ptr->ch == *str)
                {
                    if (*(++str))
                    {
                        if (ptr->child != (struct try *)NULL)
                            ptr = ptr->child;
                        else
                            break;
                    }
                    else
                    {
                        ptr->value = code;
                        return;
                    }
                }
                else
                {
                    if ((ptr->sibling =
                       (struct try *) malloc(sizeof *ptr)) == (struct try *)NULL)
                    {
                        out_of_memory = TRUE;
                        return;
                    }

                    savedptr = ptr = ptr->sibling;
                    ptr->child = ptr->sibling = (struct try *)NULL;
                    ptr->ch = *str++;
                    ptr->value = 0;

                    break;
                }
            } /* end for (;;) */
        }
        else    /* _keytry == NULL :: First sequence to be added */
        {
            savedptr = ptr = _keytry = (struct try *) malloc(sizeof *ptr);

            if (ptr == (struct try *) NULL)
            {
                out_of_memory = TRUE;
                return;
            }

            ptr->child = ptr->sibling = (struct try *) NULL;
            ptr->ch = *(str++);
            ptr->value = 0;
        }

            /* at this point, we are adding to the try.  ptr->child == NULL */
```

```
            while (*str)
            {
                ptr->child = (struct try *) malloc(sizeof *ptr);

                ptr = ptr->child;

                if (ptr == (struct try *)NULL)
                {
                    out_of_memory = TRUE;

                    ptr = savedptr;
                    while (ptr != (struct try *)NULL)
                    {
                        savedptr = ptr->child;
                        free(ptr);
                        ptr = savedptr;
                    }

                    return;
                }

                ptr->child = ptr->sibling = (struct try *)NULL;
                ptr->ch = *(str++);
                ptr->value = 0;
            }

            ptr->value = code;
            return;
}

/* Включение альтернативного режима клавиатуры */
void keypad_on(){
            if( keypadok ) return;
            keypadok = TRUE;
            if( keypad_xmit ) _puts( keypad_xmit );
}

/* Включение стандартного режима клавиатуры */
void keypad_off(){
            if( !keypadok ) return;
            keypadok = FALSE;
            if( keypad_local ) _puts( keypad_local );
}
```

```
/* Тестовая функция */
int dotest()
{
        struct termios saved_modes;
        int c;
        char *s;
        char keyname[20];

        if( tcgetattr(INPUT_CHANNEL, &saved_modes) < 0 ){
err:            keyboard_access_denied();
        }
        if( tcsetattr(INPUT_CHANNEL, TCSADRAIN, &new_modes) < 0 )
                goto err;

        keyreset();

        for(;;){
                c = getch();

                switch(c){
                case KEY_DOWN:      s = "K_DOWN"  ; break;
                case KEY_UP:        s = "K_UP"    ; break;
                case KEY_LEFT:      s = "K_LEFT"  ; break;
                case KEY_RIGHT:     s = "K_RIGHT" ; break;
                case KEY_PGDN:      s = "K_PGDN"  ; break;
                case KEY_PGUP:      s = "K_PGUP"  ; break;
                case KEY_HOME:      s = "K_HOME"  ; break;
                case KEY_END:       s = "K_END"   ; break;
                case KEY_BACKSPACE: s = "K_BS"    ; break;
                case '\t':          s = "K_TAB"   ; break;
                case KEY_BACKTAB:   s = "K_BTAB"  ; break;
                case KEY_DC:        s = "K_DEL"   ; break;
                case KEY_IC:        s = "K_INS"   ; break;
                case KEY_DL:        s = "K_DL"    ; break;
                case KEY_IL:        s = "K_IL"    ; break;

                case KEY_F(1):      s = "K_F1"    ; break;
                case KEY_F(2):      s = "K_F2"    ; break;
                case KEY_F(3):      s = "K_F3"    ; break;
                case KEY_F(4):      s = "K_F4"    ; break;
                case KEY_F(5):      s = "K_F5"    ; break;
                case KEY_F(6):      s = "K_F6"    ; break;
                case KEY_F(7):      s = "K_F7"    ; break;
                case KEY_F(8):      s = "K_F8"    ; break;
                case KEY_F(9):      s = "K_F9"    ; break;
                case KEY_F(10):     s = "K_F10"   ; break;
                case KEY_F(11):     s = "K_F11"   ; break;
                case KEY_F(12):     s = "K_F12"   ; break;

                case ESC:           s = "ESC"     ; break;
                case EOF:           s = "K_EOF"   ; break;
                case '\r':          s = "K_RETURN"; break;
                case '\n':          s = "K_ENTER" ; break;
                default:
                        s = keyname;
                        if( c >= 0400 ){
                                sprintf(keyname, "K_F%d", c - KEY_F(0));
                        } else if( iscntrl(c)){
                                sprintf(keyname, "CTRL(%c)", c + 'A' - 1);
                        } else {
                                sprintf(keyname, "%c", c );
                        }
                }
                printf("Клавиша: %s\n\r", s);

                if(c == ESC)
```

```
                        break;
                }
                tcsetattr(INPUT_CHANNEL, TCSADRAIN, &saved_modes);
        }

        /* Функция настройки на систему команд дисплея */
        void tinit (void) {
            /* static */ char Tbuf[2048];
            /* Tbuf должен сохраняться все время, пока могут вызываться функции tgetstr().
             * Для этого он либо должен быть static, либо вызов функции keyinit()
             * должен находиться внутри tinit(), что и сделано.
             */
            char *tname;
            extern char *getenv();

            if((tname = getenv("TERM")) == NULL){
                printf("TERM не определено: неизвестный тип терминала.\n");
                exit(2);
            }
            printf("Терминал: %s\n", tname);

            /* Прочесть описание терминала в Tbuf */
            switch (tgetent(Tbuf, tname)) {
                 case -1:
                    printf ("Нет файла TERMCAP (/etc/termcap).\n");
                    exit (1);
                case 0:
                    printf ("Терминал '%s' не описан.\n", tname);
                    exit (2);
                case 1:
                    break;              /* OK */
            }
            if(strlen(Tbuf) >= 1024)
        printf("Описание терминала слишком длинное - возможны потери в конце описания\n");

            keyinit();  /* инициализировать строки, пока Tbuf[] доступен */
        }

        void main(void){
                setlocale(LC_ALL, "");
                tinit();
                /* keyinit(); */
                dotest();
                exit(0);
        }
```

По поводу этого алгоритма надо сказать еще пару слов. Его модификация может с успехом применяться для поиска слов в таблице (команд, ключей в базе данных, итп.): список слов превращается в дерево. В таком поисковом алгоритме не требуются таймауты, необходимые при вводе с клавиатуры, поскольку есть явные терминаторы строк - символы '\0', которых нет при вводе с клавиатуры. В чем эффективность такого алгоритма? Сравним последовательный перебор при помощи **strcmp** и поиск в дереве букв:

```
"zzzzzzzzza"
"zzzzzzzzzb"
"zzzzzzzzzbx"
"zzzzzzzzzc"
"zzzzzzzzzcx"
```

Для линейного перебора (даже в отсортированном массиве) поиск строки zzzzzzzzzcx потребует

```
zzzzzzzzza     |       11 сравнений, отказ
zzzzzzzzzb     |       11 сравнений, отказ
zzzzzzzzzbx    |       12 сравнений, отказ
zzzzzzzzzc     |       11 сравнений, отказ
zzzzzzzzzcx    V       12 сравнений, успех
```

Всего: 57 шагов. Для поиска в дереве:

```
__z__z__z__z__z__z__z__z__z__a__\0
                                |_b__\0
                                |  |_x__\0
                                |
                                |_c__\0
                                   |_x__\0
```

потребуется проход вправо (вниз) на 10 шагов, потом выбор среди 'a','b','c', потом - выбор среди '\0' и 'x'. Всего: 15 шагов. За счет того, что общий "корень" проходится ровно один раз, а не каждый раз заново. Но это и требует предварительной подготовки данных: превращения строк в дерево!

8.18. Напишите функцию для "экранного" редактирования вводимой строки в режиме **CBREAK**. Напишите аналогичную функцию на **curses**-е. В **curses**-ной версии надо уметь отрабатывать: забой (удаление символа перед курсором), отмену всей строки, смещение влево/вправо по строке, удаление символа над курсором, вставку пробела над курсором, замену символа, вставку символа, перерисовку экрана. Учтите, что параллельно с изменением картинки в окне, вы должны вносить изменения в некоторый массив (строку), которая и будет содержать результат. Эта строка должна быть аргументом функции редактирования.

Забой можно упрощенно эмулировать как

```
addstr( "\b \b" );
        или
addch( '\b' ); delch();
```

Недостатком этих способов является некорректное поведение в начале строки (при *x*==0). Исправьте это!

8.19. На **curses**-е напишите функцию редактирования текста в окне. Функция должна возвращать массив строк с обрезанными концевыми пробелами. Вариант: возвращать одну строку, в которой строки окна разделяются символами '\n'.

8.20. Напишите функцию, рисующую прямую линию из точки (*x1*,*y1*) в (*x2*,*y2*). Указание: используйте алгоритм Брезенхема (минимального отклонения). Ответ: пусть функция **putpixel**(*x*,*y*,*color*) рисует точку в координатах (*x*,*y*) цветом *color*.

```
void line(int x1, int y1,   int x2, int y2,
          int color){
    int dx, dy, i1, i2, i, kx, ky;
    register int d; /* "отклонение" */
    register int x, y;
    short /* boolean */ l;

    dy = y2 - y1; dx = x2 - x1;
    if( !dx && !dy ){
            putpixel(x1,y1, color); return;
    }
    kx = 1; /* шаг по x */
    ky = 1; /* шаг по y */
    /* Выбор тактовой оси */
    if( dx < 0 ){ dx = -dx; kx = -1; } /* Y */
    else if( dx == 0 )       kx = 0;    /* X */
    if( dy < 0 ){ dy = -dy; ky = -1; }
    if( dx < dy ){ l = 0; d = dx; dx = dy; dy = d; }
    else           l = 1;

    i1 = dy + dy; d = i1 - dx; i2 = d - dx;
    x = x1; y = y1;

    for( i=0; i < dx; i++ ){
         putpixel( x, y, color );

         if( l ) x += kx; /* шаг по такт. оси   */
         else    y += ky;
         if( d < 0 ) /* горизонтальный шаг      */
             d += i1;
         else{       /* диагональный шаг        */
             d += i2;
             if( l ) y += ky; /* прирост высоты */
             else    x += kx;
```

```
            }
        }
        putpixel(x, y, color);  /* последняя точка  */
    }
```

8.21. Составьте программу, которая строит график функции **sin**(*x*) на отрезке от 0 до 2**пи*. Учтите такие вещи: соседние точки графика следует соединять отрезком прямой, чтобы график выглядел непрерывным; не забывайте приводить **double** к **int**, т.к. координаты пикселов† - целые числа.

8.22. Напишите функцию, которая заполняет в массиве байт *count* бит подряд, начиная с *x*-ого бита от левого края массива:

```
    байт 0      |          байт 1
7 6 5 4 3 2 1 0 | 7 6  5  4  3  2  1  0 : биты в байте
0 1 2 3 4 5 6 7 | 8 9 10 11 12 13 14 15 : x
    ==========================
        x=2, count=11
```

Такой алгоритм используется в растровой машинной графике для рисования горизонтальных прямых линий (тогда массив - это видеопамять компьютера, каждый бит соответствует пикселу на экране).

Ответ (причем мы заполняем биты не просто единицами, а "узором" *pattern*):

```
void horizLine(char *addr,int x,int count,char pattern){
  static char masks[8] = {
       0xFF, 0x7F, 0x3F, 0x1F, 0x0F, 0x07, 0x03, 0x01 };
  /* индекс в этом массиве равен числу 0-битов слева */
  register i;
  char mask;
  short lbits, rbits; /* число битов слева и справа */
  short onebyte;      /* единственный байт ? */

  addr += x/8;        /* в байте 8 бит */
  mask = masks[ lbits = x & 7 ];   /* x % 8 */
  if( count >= (rbits = 8 - lbits)){
      count -= rbits; onebyte = 0;
  }else{
      mask &= ~masks[ lbits = (x+count) & 7 ];
      onebyte = 1;
  }
/* Первый байт */
  *addr = (*addr & ~mask) | (pattern & mask);
  addr++;
  /* Для pattern==0xFF можно просто
   *     *addr++ |= mask;
   * поскольку (a &~m)|(0xFF & m) = (a &~m) | m =
   *   (a|m) & (~m|m) = (a|m) & 0xFF = a | m
   * Почему здесь нельзя написать *addr++ = (*addr...) ?
   * Потому, что ++ может быть сделан ДО вычисления
   * правой части присваивания!
   */
  if(onebyte) return;
/* Средние байты */
  for(i = count/8; i > 0; --i)
      *addr++ = pattern;  /* mask==0xFF */
/* Последний байт */
  if((lbits = count & 7) == 0) return;
  /* последний байт был полным */
  mask = ~masks[lbits];
  *addr = (*addr & ~mask) | (pattern & mask);
}
```

Заметим, что для быстродействия подобные алгоритмы обычно пишутся на ассемблере.

---

† **Пиксел** (**pixel**, **pel**) - *picture element*, в машинной графике - точка растра на экране.

8.23. Напишите при помощи **curses**-а "электронные часы", отображающие текущее время большими цифрами (например, размером 8x8 обычных символов) каждые 5 секунд. Используйте **alarm**(), **pause**().

8.24. Составьте программу, реализующую простой диалоговый интерфейс, основанный на меню. Меню хранятся в текстовых файлах вида:

```
        файл menu2_12
-----------------------------------------------
ЗАГОЛОВОК_МЕНЮ
+команда_выполняемая_при_входе_в_меню
-команда_выполняемая_при_выходе_из_меню
альтернатива_1
        команда1_1
        команда1_2
альтернатива_2
        команда2_1
        команда2_2  #комментарий
        команда2_3
альтернатива_3
       >menu2_2     #это переход в другое меню
альтернатива_4
       >>menu3_7    #хранимое в файле menu3_7
...
        ...
-----------------------------------------------
```

Программа должна обеспечивать: возврат к предыдущему меню по клавише **Esc** (для этого следует хранить "историю" вызовов меню друг из друга, например в виде "полного имени меню":

```
.rootmenu.menu1_2.menu2_4.menu3_1
```

где *menuI_J* - имена файлов с меню), обеспечить выход из программы по клавишам '**q**' и **ESC**, выдачу подсказки по **F1**, выдачу полного имени меню по **F2**. Вызов меню при помощи > означает *замещение* текущего меню новым, что соответствует замене последней компоненты в полном имени меню. Вызов >> означает вызов меню как *функции*, т.е. после выбора в новом меню и выполнения нужных действий автоматически должно быть выдано то меню, из которого произошел вызов (такой вызов соответствует удлинению полного имени, а возврат из вызова - отсечению последней компоненты). Этот вызов может быть показан на экране как появление нового "выскакивающего" окна поверх окна с предыдущим меню (окно возникает чуть сдвинутым - скажем, на y=1 и x=-2), а возврат - как исчезновение этого окна. Заголовок меню должен высвечиваться в верхней строке меню:

```
   |-------------------
|--ЗАГОЛОВОК_МЕНЮ---- |
|  альтернатива_1   | |
|  альтернатива_2   | |
| *альтернатива_3   | |
|  альтернатива_4   |--
---------------------
```

Сначала реализуйте версию, в которой каждой "*альтернативе*" соответствует единственная строка "*команда*". Команды следует запускать при помощи стандартной функции **system**(*команда*).

Усложните функцию выбора в меню так, чтобы *альтернативы* можно было выбирать по первой букве при помощи нажатия кнопки с этой буквой (в любом регистре):

```
Compile
Edit
Run program
```

8.25. Напишите на **curses**-е функцию, реализующую выбор в меню - прямоугольной таблице:

```
слово1   слово4   слово7
слово2  *слово5   слово8
слово3   слово6
```

Строки - элементы меню - передаются в функцию выбора в виде массива строк. Число элементов меню заранее неизвестно и должно подсчитываться внутри функции. Учтите, что все строки могут не поместиться в таблице, поэтому надо предусмотреть "прокручивание" строк через таблицу при достижении края меню (т.е. таблица служит как бы "окошком" через которое мы обозреваем таблицу большего размера, возможно перемещая окно над ней). Предусмотрите также случай, когда таблица оказывается заполненной не полностью (как на рисунке).

8.26. Используя библиотеку **curses**, напишите программу, реализующую клеточный автомат Конвея "Жизнь". Правила: есть прямоугольное поле (вообще говоря бесконечное, но принято в конечной модели замыкать края в кольцо), в котором живут "клетки" некоторого организма. Каждая имеет 8 соседних полей. Следующее поколение "клеток" образуется по таким правилам:

- если "клетка" имеет 2 или 3 соседей - она выживает.
- если "клетка" имеет меньше 2 или больше 3 соседей - она погибает.
- в пустом поле, имеющем ровно 3х живых соседей, рождается новая "клетка".

Предусмотрите: редактирование поля, случайное заполнение поля, останов при смерти всех "клеток", останов при стабилизации колонии.

8.27. При помощи **curses**-а напишите экранный редактор кодов доступа к файлу (в форме *rwxrwxrwx*). Расширьте программу, позволяя редактировать коды доступа у группы файлов, изображая имена файлов и коды доступа в виде таблицы:

```
НАЗВАНИЕ    КОДЫ ДОСТУПА
  файл1       rwxrw-r--
  файл2       rw-r-xr-x
  файл3       rwxrwxr--
```

Имена файлов задавайте как аргументы для **main**(). Указание: используйте для получения текущих кодов доступа системный вызов **stat**(), а для их изменения - системный вызов **chmod**().

9. Приложения.

9.1. Таблица приоритетов операций языка C++

Операции, расположенные выше, имеют больший приоритет.

```
Операторы                                  Ассоциативность
-------------------------------------------------------
1.  () []   ->   ::   .                    Left to right
2.  !  ~  +  -  ++  --  &  *
    (typecast)  sizeof  new  delete        Right to left
3.  .*   ->*                               Left to right
4.  *   /    %                             Left to right
5.  +   -                                  Left to right
6.  <<  >>                                 Left to right
7.  <   <=   >   >=                        Left to right
8.  ==  !=                                 Left to right
9.  &                                      Left to right
10. ^                                      Left to right
11. |                                      Left to right
12. &&                                     Left to right
13. ||                                     Left to right
14. ?: (условное выражение)                Right to left
15. =  *=  /=  %=  +=  -=  &=
    ^=  |=  <<=  >>=                       Right to left
16. ,                                     Left to right
```

Здесь "*****" и "**&**" в строке 2 - это адресные операции; в строке 2 "**+**" и "**-**" - унарные; "**&**" в строке 9 - это побитное "и"; "**(typecast)**" - приведение типа; "**new**" и "**delete**" - операторы управления памятью в **C++**.

Ассоциативность **Left to right** (слева направо) означает группировку операторов таким образом:

```
  A1 @ A2  @ A3    это
((A1 @ A2) @ A3)
```

Ассоциативность **Rigth to left** (справа налево) это

```
  A1 @  A2 @ A3    это
(A1 @ (A2 @ A3))
```

9.2. Правила преобразований типов.

9.2.1. В выражениях.

1. Если операнд имеет тип не int и не double, то сначала приводится:

```
signed   char  --> int  расширением знакового бита (7)
unsigned char  --> int  дополнением нулями слева
         short --> int  расширением знакового бита (15)
unsigned short --> unsigned int   дополнением нулями слева
         enum  --> int  порядковый номер в перечислимом типе
         float --> double   дробная часть дополняется нулями
```

2. Если любой операнд имеет тип double, то и другой операнд приводится к типу double. Результат: типа double. Запишем все дальнейшие преобразования в виде схемы:

```
    если есть          то другой          результат
   операнд типа        приводится к типу  имеет тип

     if(double)        -->double          double
else if(unsigned long) -->unsigned long   unsigned long
else if(long)          -->long            long
else if(unsigned int)  -->unsigned int    unsigned int
else оба операнда имеют тип int          int
```

При вызове функций их аргументы - тоже выражения, поэтому в них приводятся char,short к int и float к double. Это говорит о том, что аргументы (формальные параметры) функций можно всегда объявлять как int и double вместо char,short и float соответственно. Зато спецификатор unsigned является существенным.

9.2.2. В присваиваниях.

```
op = expr;
```

Тип выражения *expr* приводится к типу левой части - *op*. При этом возможны приведения более "длинного" типа к более "короткому" при помощи усечения, вроде:

```
int     -->  char   обрубается старший байт.
long    -->  int    обрубается старшее слово.
float   -->  int    отброс дробной части
double  -->  int      и обрубание мантиссы, если не лезет.
double  -->  float  округление дробной части.
```

Вот еще некоторые приведения типов:

```
signed    -->  unsigned   виртуально (просто знаковый бит
unsigned  -->  signed     считается значащим или наоборот).

unsigned int --> long     добавление нулей слева.
int          --> long     расширение знакового бита.

float        --> int      преобразование внутреннего
int          --> float      представления: машинно зависимо.
```

Некоторые преобразования могут идти в несколько стадий, например:

```
char --> long           это
char --> int --> long

char --> unsigned long  это
char --> int --> unsigned long
```

9.3. Таблица шестнадцатеричных чисел (HEX).

```
%d      %o      %X      побитно
--------------------------------
0        0      0x0       0000
1        1      0x1       0001
2        2      0x2       0010
3        3      0x3       0011
4        4      0x4       0100
5        5      0x5       0101
6        6      0x6       0110
7        7      0x7       0111

--------------------------------
8      010      0x8       1000
9      011      0x9       1001
10     012      0xA       1010
11     013      0xB       1011
12     014      0xC       1100
13     015      0xD       1101
14     016      0xE       1110
15     017      0xF       1111
16     020     0x10      10000
```

9.4. Таблица степеней двойки.

```
 n     2**n   |    n     2**n
-------------|--------------
 0      1     |    8      256
 1      2     |    9      512
 2      4     |   10     1024
 3      8     |   11     2048
 4     16     |   12     4096
 5     32     |   13     8192
 6     64     |   14    16384
 7    128     |   15    32768
              |   16    65536
```

9.5. Двоичный код: внутреннее представление целых чисел.

Целые числа в большинстве современных компьютеров представлены в виде *двоичного кода*. Пусть машинное слово состоит из 16 бит. Биты нумеруются справа налево начиная с 0. Обозначим условно бит номер *i* через *b*[*i*]. Значением его может быть либо 0, либо 1.

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 1| 0| 1| 1| 0| 1| 1| 0| 1| 1| 0| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Тогда **unsigned** число, записанное в слове, равно

```
d = 2**15 * b[15] +
    2**14 * b[14] +
          ...
    2**1  * b[1]  +
            b[0];
```

(2**n* - это 2 в степени *n*). Такое разложение числа *d* **единственно**. При сложении двух чисел биты складываются по правилам:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 и перенос 1 в разряд слева
```

Числа со знаком интерпретируются чуть иначе. Бит *b*[15] считается *знаковым*: 0 - число положительно или равно нулю, 1 - отрицательно. Отрицательные числа хранятся в виде *дополнительного* кода:

```
-a = ~a + 1
```

Например:

```
  2   = 0000000000000010
 ~2   = 1111111111111101
 ~2+1 = 1111111111111110 = -2

 -1   = 1111111111111111
 -2   = 1111111111111110
 -3   = 1111111111111101
 -4   = 1111111111111100
 -5   = 1111111111111011
```

Такое представление выбрано исходя из правила

```
a + (-a) = 0
```

```
      знак|
 2 =    0|000000000000010      сложим их
-2 =    1|111111111111110
---------|---------------
сумма: 10|000000000000000
```

Как видим, произошел перенос 1 в бит номер 16. Но слово содержит лишь биты 0..15 и бит *b*[16] просто игнорируется. Получается, что сумма равна

```
0000000000000000 = 0
```

что и требовалось. В двоичном коде вычитание реализуется по схеме

```
a - b = a + (-b) = a + (~b + 1)
```

Восьмеричные числа соответствуют разбиению двоичного числа на группы по 3 бита и записи каждой группы в виде соответствующей восьмеричной цифры (смотри таблицу выше). Шестнадцатеричные числа соответствуют разбиению на группы по 4 бита (*nibble*):

```
                x = 0010011111011001
        число: 0010 0111 1101 1001
16-ричное:  0x   2    7    D    9    = 0x27D9

        число: 0 010 011 111 011 001
8-ричное:   0  0  2   3   7   3   1  = 023731
```

10. Примеры.

В данном приложении приводится несколько содержательных и достаточно больших примеров, которые иллюстрируют как сам язык Си, так и некоторые возможности системы **UNIX**, а также некоторые программистские приемы решения задач при помощи Си. Многие из этих примеров содержат в качестве своих частей ответы на некоторые из задач. Некоторые примеры позаимствованы из других книг, но дополнены и исправлены. Все примеры проверены в действии. Смысл некоторых функций в примерах может оказаться вам неизвестен; однако в силу того, что данная книга не является учебником, мы отсылаем вас за подробностями к "Оперативному руководству" (man) по операционной системе **UNIX** и к документации по системе.

И в заключение - несколько слов о путях развития языка "C". Чистый язык "C" уже отстал от современных технологий программирования. Такие методы как модули (языки "Modula-2", "Ada", "CLU"), родовые пакеты ("Ada", "CLU"), объектно-ориентированное программирование ("Smalltalk", "CLU") требуют новых средств. Поэтому в настоящее время "C" постепенно вытесняется более мощным и интеллектуальным языком "C++" †, обладающим средствами для объектно-ориентированного программирования и родовых классов. Существуют также расширения стандартного "C" объектно-ориентированными возможностями ("Objective-C"). Большой простор предоставляет также сборка программ из частей, написанных на разных языках программирования (например, "C", "Pascal", "Prolog").

---

† C++ как и C разработан в AT&T; произносится "Си плас-плас"

£££ Автор благодарит авторов программ и книг по Си и **UNIX**, по которым некогда учился я сам; коллег из ИПК Минавтопрома/Демоса; программистов из сетей Usenet и Relcom, давших материалы для задач и рассуждений; слушателей курсов по Си за многочисленный материал для книги.

А.Богатырев.

```
Ученью не один мы посвятили год,
Потом других учить пришел и нам черед.
Какие ж выводы из этой всей науки?
Из праха мы пришли, нас ветер унесет.
                        Омар Хайям
```

Оглавление.

0. Напутствие в качестве вступления. 2
1. Простые программы и алгоритмы. Сюрпризы, советы. 4
2. Массивы, строки, указатели. 70
3. Мобильность и машинная зависимость программ. Проблемы с русскими буквами. 106
4. Работа с файлами. 118
5. Структуры данных. 148
6. Системные вызовы и взаимодействие с **UNIX**. 161
6.1. Файлы и каталоги. 163
6.2. Время в UNIX. 171
6.3. Свободное место на диске. 179
6.4. Сигналы. 182
6.5. Жизнь процессов. 189
6.6. Трубы и FIFO-файлы. 198
6.7. Нелокальный переход. 200
6.8. Хозяин файла, процесса, и проверка привелегий. 202
6.9. Блокировка доступа к файлам. 206
6.10. Файлы устройств. 210
6.11. Мультиплексирование ввода-вывода 222
6.12. Простой интерпретатор команд. 233
7. Текстовая обработка. 243
8. Экранные библиотеки и работа с видеопамятью. 299
9. Приложения. 338
9.1. Таблица приоритетов операций языка C++ 338
9.2. Правила преобразований типов. 338
9.3. Таблица шестнадцатеричных чисел (HEX). 339
9.4. Таблица степеней двойки. 339
9.5. Двоичный код: внутреннее представление целых чисел. 340
10. Примеры. 341
Пример 1. Размен монет. .....
Пример 2. Подсчет букв в файле. .....
Пример 3. Центрирование строк. .....
Пример 4. Разметка текста для **nroff**. .....
Пример 5. Инвертирование порядка слов в строках. .....
Пример 6. Пузырьковая сортировка. .....
Пример 7. Хэш-таблица. .....
Пример 8. Простая база данных. .....
Пример 9. Вставка/удаление строк в файл. .....
Пример 10. Безопасный **free**, позволяющий обращения к автоматическим переменным. .....
Пример 11. Поимка ошибок при работе с динамической памятью. .....
Пример 12. Копирование/перемещение файла. .....
Пример 13. Обход поддерева каталогов в **MS DOS** при помощи **chdir**. .....
Пример 14. Работа с сигналами. .....
Пример 15. Управление скоростью обмена через линию. .....
Пример 16. Просмотр файлов в окнах. .....
Пример 17. Работа с иерархией окон в **curses**. Часть проекта **uxcom**. .....
Пример 18. Поддержка содержимого каталога. Часть проекта **uxcom**. .....

Пример 19. Роллируемое меню. Часть проекта **uxcom**. .....
Пример 20. Выбор в строке-меню. Часть проекта **uxcom**. .....
Пример 21. Редактор строки. Часть проекта **uxcom**. .....
Пример 22. Выбор в прямоугольной таблице. Часть проекта **uxcom**. .....
Пример 23. **UNIX commander** - простой визуальный Шелл. Головной модуль проекта **uxcom**. .....
Пример 24. Общение двух процессов через "трубу". .....
Пример 25. Общение процессов через FIFO-файл. .....
Пример 26. Общение процессов через общую память и семафоры. .....
Пример 27. Протоколирование работы программы при помощи псевдотерминала и процессов. ....
Пример 28. Оценка фрагментированности файловой системы. .....
Пример 29. Восстановление удаленного файла в **BSD-2.9**. .....
Пример 30. Копирование файлов из **MS DOS** в **UNIX**. .....
Пример 31. Программа, печатающая свой собственный текст. .....
Пример 32. Форматирование текста Си-программы. .....
1.11. Треугольник из звездочек. 6
1.34. Простые числа. 10
1.36. Целочисленный квадратный корень. 12
1.39. Вычисление интеграла по Симпсону. 13
1.49. Сортировка Шелла. 19
1.50. Быстрая сортировка. 20
1.67. Функция чтения строки. 26
1.88. Перестановки элементов. 35
1.117. Схема Горнера. 50
1.137. Системная функция **qsort** - формат вызова. 58
1.146. Процесс компиляции программ. 66
2.58. Функция **bcopy**. 92
2.59. Функция **strdup**. 95
2.61. Упрощенный аналог функции **printf**. 96
3.9. **_ctype**[] 109
3.12. Программа транслитерации: **tr**. 112
3.16. Функция записи трассировки (отладочных выдач) в файл. 114
3.18. Условная компиляция: #ifdef 114
4.39. Быстрый доступ к строкам файла. 139
4.45. Эмуляция основ библиотеки STDIO, по мотивам 4.2 BSD. 142
5.12. Отсортированный список слов. 155
5.16. Структуры с полями переменного размера. 157
5.17. Список со "старением". 159
6.1.1. Определение типа файла. 163
6.1.3. Выдача неотсортированного содержимого каталога (**ls**). 165
6.1.5. Рекурсивный обход каталогов и подкаталогов. 166
6.2.9. Функция задержки в микросекундах. 173
6.4.3. Функция **sleep**. 187
6.10.1. Определение текущего каталога: функция **getwd**. 217
6.10.2. Канонизация полного имени файла. 221
6.11.1. Мультиплексирование ввода из нескольких файлов. 223
6.11.2. Программа **script**. 225
7.12. Программа **uniq**. 244
7.14. Расширение табуляций в пробелы, функция **untab**. 245
7.15. Функция **tabify**. 245
7.25. Поиск методом половинного деления. 247
7.31. Программа печати в две полосы. 251
7.33. Инвертирование порядка строк в файле. 255
7.34. Перенос неразбиваемых блоков текста. 257

7.36. Двоичная сортировка строк при помощи дерева. 259
7.41. Функция **match**. 266
7.43. Функция контекстной замены по регулярному выражению. 270
7.44. Алгоритм быстрого поиска подстроки в строке. 272
7.52. Коррекция правописания. 276
7.67. Калькулятор-1. 283
7.68. Калькулятор-2. 289
8.1. Осыпающиеся буквы. 301
8.13. Использование библиотеки **termcap**. 309
8.17. Разбор ESC-последовательностей с клавиатуры. 320

11. Список литературы.

(1) Б.Керниган, Д.Ритчи, А.Фьюер. *Язык программирования Си. Задачи по языку Си.* - М.: Финансы и статистика, 1985.

(2) М.Уэйт, С.Прата, Д.Мартин. *Язык Си. Руководство для начинающих.* - М.: Мир, 1988.

(3) М.Болски. *Язык программирования Си. Справочник.* - М.: Радио и связь, 1988.

(4) Л.Хэнкок, М.Кригер. *Введение в программирование на языке Си.* - М.: Радио и связь, 1986.

(5) М.Дансмур, Г.Дейвис. *ОС UNIX и программирование на языке Си.* - М.: Радио и связь, 1989.

(6) Р.Берри, Б.Микинз. *Язык Си. Введение для программистов.* - М.: Финансы и статистика, 1988.

(7) М.Беляков, А.Ливеровский, В.Семик, В.Шяудкулис. *Инструментальная мобильная операционная система ИНМОС.* - М.: Финансы и статистика, 1985.

(8) К.Кристиан. *Введение в операционную систему UNIX.* - М.: Финансы и статистика, 1985.

(9) Р.Готье. *Руководство по операционной системе UNIX.* - М.: Финансы и статистика, 1986.

(10) М.Банахан, Э.Раттер. *Введение в операционную систему UNIX.* - М.: Радио и связь, 1986.

(11) С.Баурн. *Операционная система UNIX.* - М.: Мир, 1986.

(12) П.Браун. *Введение в операционную систему UNIX.* - М.: Мир, 1987.

(13) M.Bach. *The design of the UNIX operating system.* - Prentice Hall, Englewood Cliffs, N.J., 1986.

(14) S.Dewhurst, K.Stark. *Programming in C++.* - Prentice Hall, 1989.

(15) M.Ellis, B.Stroustrup. *The annotated C++ Reference Manual.* - Addison-Wesley, 1990.