# 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

```
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);
}
```

Разберитесь с принципом формирования массива **UU**.

* 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).

** = "Куча" (**heap**, **pool**) - область статической памяти, увеличивающаяся по мере надобности, и предназначенная как раз для хранения динамически отведенных данных.

*** 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).



Назад | Содержание | Вперед

[ Главная ] [ Гостевая ]