СТРОКИ В ПРОГРАММАХ НА СИ

В языке СИ, в отличие от многих языков программирования, отсутствует строковый тип, хотя есть понятие строки (строковый литерал). Используемые в программе строковые литералы размещаются компилятором СИ в статической памяти. Строка в СИ представляется в памяти ЭВМ как массив элементов типа char, последним символом в котором является **нуль-символ**('\0'). Все функции для работы со строками в Си ориентированы именно на такое представление строк (0 в конце).

1. Определение (и инициализация)строки

Существуют четыре **способа** определения (с инициализацией в месте описания) строк в СИ:

- как символьный массив с явным указанием размера памяти (const-указатель),

  char m1[10] = "Строка"; // это не присваивание, а инициализация
  // (компилятор выполняет до запуска программы)

- как символьный массив без явного указания размера памяти (const-указатель),

  char m2[] = "Строка";

NB: Этот вариант отличен от других тем, что во всех местах его использования кроме заголовка функции при отсутствии явной инициализации компилятор выдаёт ошибку (а в заголовке функции не выдает).

- как указатель на char (переменная-указатель),

  char *m3 = "Строка".

- четвертый способ является развернутой формой второго способа:

  char m4[ ] = {'С', 'т', 'р', 'о', 'к', 'а', '\0'};

В **первом случае** (когда размер требуемой памяти указан явно) компилятором выделяется память (статически) под строковый литерал и память под массив m1 размером 10 байт (в которой мы должны предусмотреть и место для нуль-символа – если строковый литерал справа от знака присваивания будет из 10 символов, то нуль-символ в массив не попадет).

1

Во втором (когда размер требуемой памяти явно не указан) выделяется память (статически) под строковый литерал и память под массив m2, но размер выделяемой под m2 памяти равен числу всех значащих символов строкового литерала плюс еще один байт для размещения завершающего нуль-символа (в данном случае выделяется 7 байт). При этом m1 и m2 рассматриваются компилятором как синонимы адресов первых элементов соответствующих массивов, т.е. m1 и m2 интерпретируются в программе на СИ как константы-указатели, т.е. в программе нельзя изменить m1 и m2, т.к. это будет воспринято как попытка изменить положение первого символа массива в памяти.

В **третьем случае** компилятором будет выделена область статической памяти (7 байт) для размещения строкового литерала и память под указатель m3, т.е. имеется отличие от первых двух случаев: m3 в отличие от m1 и m2 является переменной указателем.

2. Массивы символьных строк

Массив символьных строк может описываться (и инициализируется в месте описания) на СИ как **массив указателей** вида

char *str1[3] = {«Это строка», «Еще одна строка»},

т.е. с явным указанием числа строк в массиве (= числу указателей в массиве укакзателей)

или

char *str2[]= {«Новая строка», «Совсем новая строка»},

т.е. без явного указания числа строк.

В обоих случаях компилятором выделяется память (статически) для двух строковых литералов. Кроме того, в первом случае в памяти выделяется область для размещения массива из 3 указателей, первые два из которых получит начальное значение, равное адресу начала размещения в памяти соответствующего строкового литерала. Во втором случае компилятором будет выделена область для размещения массива из двух указателей, которые получат начальные значения, равные адресам начала размещения в памяти соответствующих строковых литералов.

В этих примерах справедливы следующие равенства:

*(str1[0]) == ‘Э’ - первый символ первой строки **!= str1[0][0]** и **!= вся первая строка**

*(str1[1]) == ‘Е’ - первый символ второй строки (первого массива).

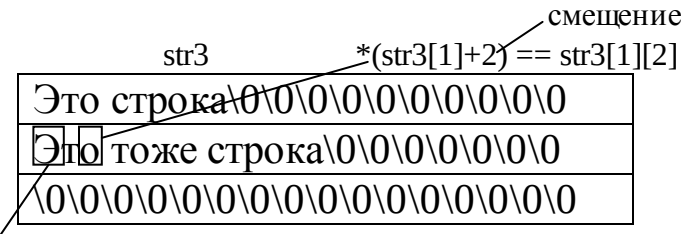
Описанные выше символьные массивы часто называются «**свободными** » или «**рваными**», так как число символов в их строках не задано явно и потому длины хранимых строк различны. Хранятся такие массивы в памяти следующим компактным образом:

число строк число символов в строке

Однако размер строки в символьном массиве можно указывать явно, например

char str3[3][20] = {"Это строка", "Это тоже строка"},

и размещаться в памяти этот символьный массив будет уже как **прямоугольный** следующего вида:



*(str3[1]) == str3[1][0]

3. Присваивание значения строке

Ранее мы рассмотрели случаи, когда **определение строки совмещено с ее инициализацией**. Если же это не так, т.е. если вначале только определяется строка (чтобы компилятор выделил необходимую область памяти), а затем в момент выполнения программы заполняется область памяти для строки нужными значениями, то можно попасть **в ловушку.**

Как мы уже говорили, строки можно объявить как указатели на тип данных char или как массивы данных типа char. Между этими двумя случаями существует одно **важное различие**:

- если используется переменная-указатель вида char *a, то память для содержимого строки (адрес которой передаётся указателю) **не резервируется** (только под указатель);

- если используется массив данных, то память **резервируется автоматически**, а имя массива является синонимом адреса начала зарезервированной области памяти.

Непонимание этой разницы может привести к **двум типам ошибок**. Рассмотрим следующую программу:

```
#include <stdio.h>
#include <string.h>
void  main(void)
{
        char *name;
        char msg[10];
        printf("Назовите имя --> ");
        scanf("%s", name);
        msg = "Здравствуйте, ";
        printf("%s %s \n", msg, name);
}
```

На первый взгляд все корректно, однако здесь допущены **две ошибки.**

Первая ошибка содержится в выражении:

scanf("%s", name).

Выражение само по себе законно и корректно. Поскольку *name* является указателем на char, то не нужно ставить перед ним адресный оператор (&). Однако память для вводимой по адресу *name* строки не зарезервирована и строка, которую мы введем, будет записана по какому-то случайному адресу, который окажется в *name*. Компилятор обнаружит это, но поскольку эта ситуация не приведет к сбою выполнения программы (т.к. строка все же будет сохранена), то компилятор выдаст лишь предупреждающее сообщение (warning), но не ошибку.

"Possible use of 'name' before definition" ("Возможно использование 'name' до ее определения")

Вторая ошибка содержится в операторе

msg = "Здравствуйте,"

Компилятор считает, что мы пытаемся заменить значение (константу) *msg* на адрес строковой константы "Здравствуйте,". Это сделать невозможно, поскольку имена массивов (*msg*) являются константами, и не могут быть модифицированы. Компилятор выдаст сообщение об ошибке:

"Lvalue required." ("Использование константы недопустимо [в левой части оператора присваивания]")

Каково решение этой проблемы? Простейший выход - изменить способ описания переменных name и msg:

```
#include <stdio.h>
#include <string.h>
void  main(void)
{
        char name[10];
        char *msg;
        printf("Назовите имя --> ");
        scanf("%s", name);
        msg = "Здравствуйте,  ";
        printf("%s %s \n", msg, name);
}
```

Здесь символьному массиву *name* выделяется память, а переменной-указателю *msg* присваивается адрес строковой константы "Здравствуйте, ". Отметим, что в операторе msg = "Здравствуйте, "; сам строковый литерал не копируется, в *msg* передается лишь указатель на его первый символ.

Если, тем не менее, мы хотим оставить старое описание *name* и *msg*, то нужно изменить программу следующим образом:

```
#include <stdio.h>
#include <string.h>  //для функции strcpy( )
#include <malloc.h>  //для функции malloc( )
void  main(void)
{
        char *name;
        char msg[10];
        name = (char *) malloc(10);
        printf("Назовите имя --> ");
        scanf("%s", name);
        strcpy(msg, "Здраствуйте,  ");
        printf("%s %s \n", msg, name);
}
```

Замечания:

1) Вызов функции *malloc()* выделяет отдельно 10 байтов памяти и присваивает адрес этого участка памяти *name*, решив нашу первую проблему.
2) Функция *strcpy()* производит посимвольное копирование из строковой константы *string* "Здравствуйте," в массив *msg*.

4. Передача строк как параметров функций

При передаче в функцию строки (вспомним, что строка в СИ хранится как массив символов) надо соблюдать следующие правила:

1) соответствующим **формальным параметром** функции объявляется или указатель на тип char, т.е.

```
char *str;
```

или т.н. «открытый» массив (массив с незафиксированными границами) символов, т.е.

```
char str[];
```

2) соответствующим **фактическим параметром** при вызове функции должно быть указано или имя указателя на тип char (он должен быть связан с ранее уже выделенной областью памяти), или имя массива символов;

3) внутри функции **доступ к элементам строки** может быть осуществлен либо с помощью операций с указателями, либо с помощью индексов массива символов.

Примеры:

```
void strcpy1(char *s1, char *s2)
{
  while (( *s1 = *s2 ) != '\0' )
    {          присваивание
        s1++;
        s2++;
    }
}
```

```
void strcpy12(char *s1, char *s2)
{// это разновидность strcpy1
  while (( *s1++ = *s2++ ) != '\0' )
}
```

здесь вначале - разыменование, потом – увеличение,
а если записать *++s1, то вначале - увеличение (++), а затем разыменование (*)

```
void strcpy2(char s1[], char s2[])
{
  int i;
  i = 0;
  while (( s1[i] = s2[i] ) != '\0' )
     i++;
}
```

Вызов функций имеет вид:

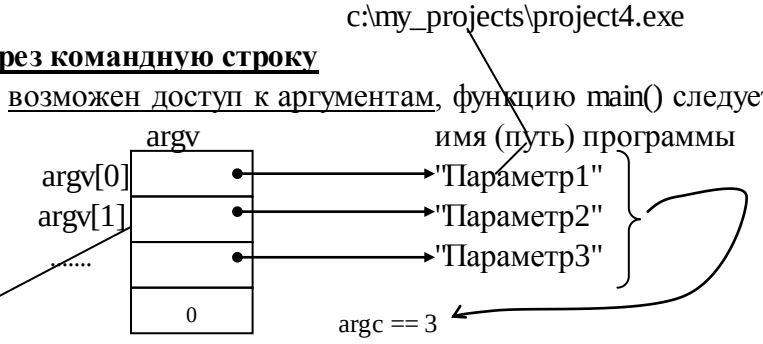
```
char str[80];
strcpy1(str, "Строка");    или    strcpy2(str, "Тоже строка");
```

5. Передача параметров функции main() через командную строку

Для того, чтобы в теле программы был возможен доступ к аргументам, функцию main() следует описать в одной из следующих форм:

```
<тип> main( int argc, char *argv[] )
{
 тело функции
}
или
<тип> main( int argc, char **argv) .
{
 тело функции
}
```

это аналогично



В этом случае функция main() будет иметь доступ ко всем словам командной строки.

Слова командной строки - преобразованные к формату ASCIIZ-строк последовательности символов, разделенные пробелами. Эти слова размещаются в командной строке, заданной при запуске программы. Число слов хранится в параметре *argc*.

Смысл параметров, передаваемых в точку входа main():

- int argc - число слов в командной строке в момент запуска программы; значение этого параметра всегда больше или равно единице (единице равно, если указано лишь имя программы);
- char **argv - свободный массив из *argc+1* элементов(строк) или, что то же самое, массив указателей (на строки) из переменного числа элементов. Каждый элемент этого массива является указателем на слово (строку) в командной строке в формате ASCIIZ-строки. argv[0] - указатель на начало имени запущенного на выполнение файла, argv[1] - указатель на начало первого параметра, argv[2] - указатель на начало второго параметра и т.д. Признаком завершения массива является нулевой указатель, т.е. argv[последний] == '\0', т.е. argv[argc] == '\0';

=argc, если считать с нуля

Пример программы, распечатывающей параметры командной строки:

```
#include <stdio.h>
void main( int argc, char **argv)
{
  int i;
  printf("В командной строке задано %d слов \n", argc);
  puts("Параметры командной строки:");
  for ( i = 0; i < argc ; i++ )
     printf("Параметр %d = %s \n", i, argv[i]);
}
```

≡ for (i = 0; argv[i] ; i++)

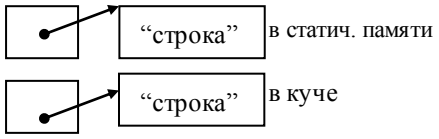
6. Ошибки при работе со строками

Первая: при выделении памяти для копирования строки забывают прибавить 1 к ее длине, то есть

```
char *s = "abcdef";
char *p = malloc (strlen(s)); // здесь забывают прибавить 1  // должно быть p = malloc (strlen(str)+1);
strcpy (p, s);
```

Вторая: используют уже освобожденную по free строку.

```
char *s = "строка";
char *p = strdup (s);
free (p); // дальше строку p использовать нельзя (нельзя, т.е. бессмысленно вызывать puts(p)).
```



Третья: копируют строки не по strcpy, а просто присваиванием:

```
char *a = "строка";
char b[10];
char *c = “еще строка”
b = a; // нельзя
c = a; // можно
```

Четвертая: удаляют (освобождают) строку "за брюхо", а не "за голову".

```
char *str = strdup ("строка");
str++;   // ушли от начала строки
free (str);
```

Пятая: надо различать следующие случаи при выделении памяти под строку в виде массива:

а) char a[] = "string 1"; // массив *a* создается 1 раз, в статической памяти, и живет до конца программы, видимость глобальная

б)

```
  void main(void)
  {
     вызов функции f()
  }
  void f ()
    {
      char b[] = "string 2"; // массив b  создается (в стеке) каждый раз при входе в функцию, видимость локальная.
                             // При выходе - исчезает (вместе с фреймом стека).
      .....................
    }
```

в)

```
  void f(void)
    {
      static char c[] = "string 3"; // массив c создается 1 раз в статической памяти  при первом входе
                                    // в функцию f()  и живет до конца программы (но видимость локальная)
      .............
      }
```

NB: во всех трех случаях (а, б, в) строковые литералы создаются в статической памяти и живут всю программу.