acm.mipt.ru

олимпиады по программированию на Физтехе

Раздел «Язык Си» . CoffeString :

Поиск

Поиск

Раздел «Язык Си»

Главная
Зачем учить С?
Определения

Инструменты:

Поиск
Изменения
Index
Статистика

Разделы

Информация
Алгоритмы
Язык Си
Язык Ruby
Язык Ассемблера
El Judge
Парадигмы
Образование
Сети
Objective C

Logon>>

В этом разделе предполагается, что память для строки уже выделена.

Строка - это набор символов, оканчивающихся символом **'\0'**

В языке С нет специального типа для работы со строками. Для этого используются массивы char и указатели char *.

Символьная (или литеральная) константа в языке С - это **один** символ, заключенный в **одинарные** кавычки. Например, 'q' или '0'. Имеет тип int и значение, равное ASCII коду этого символа. Значение константы '0' равно 48.

Строковая константа - это ноль или более символов в двойных кавычках. Например, "Hello". Строковая константа имеет тип char *.

Как хранятся строки

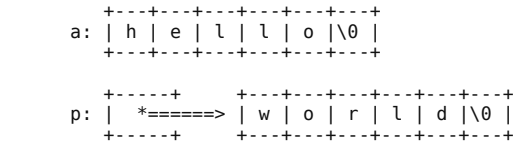
Эти три массива имеют одинаковую длину и одинаковое содержимое:

```
char b1[6] = {'w', 'o', 'r', 'l', 'd', '\0'};
char b2[] = {'w', 'o', 'r', 'l', 'd', '\0'}; // размер массива вычисляется автоматически
char b3[] = "world"; // стандарт позволяет написать при инициализации не каждый символ по отдельности, а строковую константу.
```

Хранятся массивы b1, b2, b3 одинаково - каждый массив хранится единым куском в памяти, эти массивы одинаковой длины.

Но следующие строки хранятся в памяти по-разному:

```
char a[] = "hello";    // массив из 6 символов
char * s = "world";    // 1 указатель на char, указывает на строковую константу "world"
```



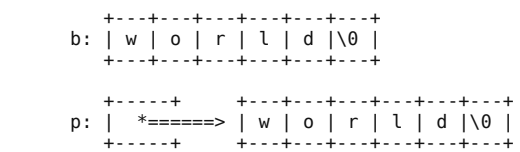
sizeof(a) равен 6, sizeof(s) равен sizeof(void*) и размеру любого другого адреса, зависит от архитектуры.

Кроме того, строковые константы могут в вашей ОС (например, Linux) хранится в read only области памяти. Т.е. их нельзя изменять, то есть код:

```
a[0] = 'H'; // заменит в массиве первый символ на H
s[0] = 'W'; // программа упадет, так как нельзя писать в read only область памяти
```

Строку НЕЛЬЗЯ сравнивать ==

Если мы напишем




```
char b[] = "world";    // массив из 6 символов
char * s = "world";    // 1 указатель на char, указывает на строковую константу "world"
if (b == s)
    printf("Равны\n");
else
    printf("НЕ равны\n");
```

Массив b содержит такое же слово, на какое указывает переменная p. Но если мы будем сравнивать их ==, то получим ложь. Будет напечатано "НЕ равны", потому что **оператор == сравнивает адреса.** Получим ложь при сравнении адреса

начала массива b и указателя p (разные адреса, p НЕ указывает в начало массива b, он указывает на строковую константу "world")

Сравнивать строки мы научимся дальше, когда будем рассказывать о стандартной функции языка C strcmp.

Как напечатать строку

Печатать строку можно **printf** по формату **%s** (string). Печатаются все символы от указанного адреса до '\0'. Сам символ '\0' не печатается.

```
char a[] = "hello";    // массив из 6 символов
char * s = "world";    // 1 указатель на char, указывает на строковую константу "world"

printf("%s\n", a);     // hello
printf("%s\n", s);     // world

printf("%s\n", a+1);   // ello
printf("%s\n", s+1);   // orld
```

Как прочитать строку

Чтобы прочитать строку, память для нее должна уже быть выделена. Либо объявлен массив подходящей длины, либо выделена динамическая память. В примерах будем объявлять массивы нужной длины.

scanf по формату %s

Формат %s позволяет функции scanf читать набор символов. Выясним на практике, как он работает.

```
#include <stdio.h>
int main() {
    char a[100];
    scanf("%s", a);        // читаем в массив a (НЕ ДЕЛАЙТЕ ТАК, НУЖЕН КОНТРОЛЬ ПЕРЕПОЛНЕНИЯ МАССИВА)
    printf("<%s>\n", a);   // печатаем прочитанное, спереди и сзади печатаем символ < и > ,
                           // чтобы увидеть где началась и закончилась строка
    return 0;
}
```

Скомпилируем файл в исполняемый модуль a.out и будем его запускать (вторая строка - что вводим, последняя - что печатает программа):

```
$./a.out
qaz wsxedc
qaz
```

Оказывается, читается не вся строка, а "слово" до пробельного символа (пробела, табуляции, \n и так далее).

```
$./a.out
   qaz123. wsxedc
qaz123.
```

Пробельные символы спереди пропускаются.

Проблема: пользователь может ввести больше 100 символов и мы выйдем за границы массива char a[100]; Что делать? Использовать модификацию к форматеру %s, чтобы указать максимальное количество прочитанных символов.

```
#include <stdio.h>
int main() {
    char a[10] = "hello";
    scanf("%3s", a);       // читаем в массив a НЕ БОЛЕЕ 3 символов
    printf("<%s>\n", a);   // печатаем прочитанное, спереди и сзади печатаем символ < и > ,
                           // чтобы увидеть где началась и закончилась строка
    return 0;
}
```

Запускаем программу:

```
$./a.out
qazxcvbnm wsxedc
qaz
```

Так как у нас закончилась печать на букве z, после нее был поставлен символ '\0'.

 То есть при указанном ограничении в 3 символа записали 4 символа.

Т.е. для массива char a[10] нужно писать scanf("%9s", a);

Что делать, если нужно прочитать не слово, а строку?

fgets

Строку можно прочитать стандартной функцией gets, но так НЕ НАДО ДЕЛАТЬ!

Даже в help по этой функции пишут, что не нужно ее использовать.

При компиляции мы получаем предупреждение, что не надо использовать gets.

```
char a[10];
gets(a);
```

Почему не надо использовать gets? Потому что в функции нигде не указывается, сколько символов можно прочитать, чтобы не выйти за границы массива. Мы никак это не контролируем.

Что делать? Использовать похожую функцию **char *fgets(char *s, int size, FILE *stream)**

```
#include <stdio.h>
int main() {
    char a[10];
    fgets(a, 5, stdin);  // читаем в массив a НЕ БОЛЕЕ 5 символов с stdin (клавиатуры)
```

```
    printf("%s\n", a);   // печатаем прочитанное
    return 0;
}
```

Запускаем программу:

```
$./a.out
1234567890
1234
```

Заметим, что записалось 5 символов ВМЕСТЕ с символом '\0'. Т.е. для массива char a[10] можно писать fgets(a, 10, stdin);

Как завершить ввод?

Допустим, мы читаем по словам (или по строкам) текст. Как сказать, что текст закончился?

```
char s[1001];
while(1 == scanf("%1000s", s)) {
    printf("%s\n", s);
}
```

Чтобы закончить ввод текста с клавиатуры, введите ^D (Linux, Mac) или ^Z (Windows)

Задача (про капитана Флинта)

Капитан Флинт зарыл клад на Острове сокровищ. Он оставил описание, как найти клад. Описание состоит из строк вида: “North 5”, где первое слово – одно из “North”, “South”, “East”, “West”, а второе число – количество шагов, необходимое пройти в этом направлении.

Напишите программу, которая по описанию пути к кладу определяет точные координаты клада, считая, что начало координат находится в начале пути, ось OX направлена на восток, ось OY – на север.

Программа получает на вход последовательность строк указанного вида, завершающуюся строкой со словом “Treasure!”. Программа должна вывести два целых числа: координаты клада.

Пример ввода:

```
North 5
East 3
South 1
Treasure!
```

Пример вывода:

```
3 4
```

(Примечание: мы будем признательны, если сможем указать автора задачи. Эта задача столько раз кочевала по разным контестам для школьников, что пора писать ~~слова народные~~ "задача классическая").

Направление и шаги мы будем читать так:

```
char sdir[10];
int step;

scanf("%9s%d", sdir, &step);
```

Теперь нужно узнать – какое именно слово лежит в массиве sdir. Т.е. сравнить строку с образцом.

Стандартные функции языка C

Полное описание функций тут

Не забудьте для работы с этими функциями написать

```
#include <string.h>
```

Рассмотрим наиболее используемые функции:

strlen

size_t strlen(const char *s);

Возвращает количество символов в строке БЕЗ подсчета '\0'

```
printf("%d\n", strlen("abc"));   // 3
```

Попробуем написать такую же функцию mystrlen и проверить ее

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

size_t mystrlen(const char *s) {
    int i;
    for (i = 0; s[i] != '\0'; i++)
        ;                              // делать в цикле ничего не нужно, пустое тело цикла
    return i;
}
int main() {
    char * s = "abc";
    printf("%zd\n", strlen(s));     // эталонная функция
    printf("%zd\n", mystrlen(s));   // наша функция
    return 0;
}
```

Как понять, что возвращать из mystrlen, i, или i-1, или i+1?

Попробуем посчитать в уме длину строки "z". При i=0 учитываем z, i++ сделает i=1, потом проверка s[i]!='\0' даст ложь и мы выйдем из цикла. i=1. Вернуть нам надо тоже 1. Т.е return i.

Проверяйте свой алгоритм мысленно на коротких примерах, строках длины 1, 0, максимум 3.

Напишем эту функцию через указатели.

Пусть указатель p сначала указывает на начало строки s, потом в цикле сдвигается на 1 символ p = p+1 или p++, пока его содержимое *p не станет равно '\0' (концу строки).

Как тогда вычислить длину строки? Пусть начало строки "abc" лежит по адресу 100 (s = 100). Тогда буква a лежит по адресу 100, b по адресу 101, c по адресу 102, \0 по адресу 103.

Когда мы закончим цикл, p будет содержать адрес 103 (был бы 102, мы бы цикл продолжали, там буква c). Вернуть нужно число 3. В переменной p у нас число 103, в переменной s число 100. Значит возвращаем p - s.

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

size_t mystrlen(const char *s) {
    const char * p;
    for (p = s; *p != '\0'; p++)   // указатель двигается от начала строки до конца
        ;                          // делать в цикле ничего не нужно, пустое тело цикла
    return p - s;
}
int main() {
    char * s = "abc";
    printf("%zd\n", strlen(s));     // эталонная функция
    printf("%zd\n", mystrlen(s));   // наша функция
    return 0;
}
```

напоминаем, что тип **size_t** печатается по формату **%zd**

Дополнительно: разберите что делает код:

```
size_t mystrlen(const char *s) {
    const char *p = s;
    while (*p++)
        ;
    return p - s;
}
```

Делает ли эта функция то же самое, или есть ошибка?

strcmp, strncmp - сравнение строк

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

Функция strcmp() сравнивает две строки: s1 и s2. Она возвращает целое число, которое меньше, больше нуля или равно ему, если s1 соответственно меньше, больше или равно s2.

Функция strncmp() работает аналогичным образом, но сравнивает только первые n символов строки s1.

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

int main() {
    char * s1, * s2;
    s1 = "abc";
    s2 = "aaaaaaa";
    printf("%d %s %s\n", strcmp(s1, s2), s1, s2);   // 1 abc aaaaaaa
    printf("%d %s %s\n", strcmp(s2, s1), s2, s1);   // -1 aaaaaaa abc
    printf("%d %s %s\n", strcmp(s1, s1), s1, s1);   // 0 abc abc

    s2 = "zaq";
    printf("%d %s %s\n", strcmp(s1, s2), s1, s2);   // -25 abc zaq

    return 0;
}
```

Если нужно выяснить, равна строка s образцу, например "Treasure!", то пишем

```
if (0 == strcmp(s, "Treasure!"))
```

Так как строки могут быть длинными, то для удобства чтения кода сначала пишут с чем сравниваем результат strcmp.

strcpy, strncpy - копирование строки

char *strcpy(char *dest, const char *src);

char *strncpy(char *dest, const char *src, size_t n);

Функция strcpy() копирует строку, на которую указывает src (включая завершающий символ '\0'), в массив, на который указывает dest. Строки не могут перекрываться, и в результирующей строке dest должно быть достаточно места для копии.

Функция strncpy работает аналогично, кроме того, что копируются только первые n байтов строки src. Таким образом, если в n байтах строки src нет нулевого байта, то строка результата не будет заканчиваться символом '\0'.

Если длина src меньше, чем n, то остальное место в dest будет заполнено нулями.

Функции strcpy() и strncpy() возвращают указатель на результирующую строку dest.

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

int main() {
    char a[100];          // нужно место куда копировать
    strcpy(a, "qaz");
    printf("%s\n", a);   // qaz

    strncpy(a, "wsx", 2);
    printf("%s\n", a);   // wsz (\0 не откопировалось, и печать идет от адреса a до ближайшего \0)
```

```
    return 0;
}
```

Попробуем написать mystrcpy сами. Сначала через индексы массива.

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

char * mystrcpy(char *dest, const char *src) {
    int i;
    for (i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];
    dest[i] = '\0';     // так как он в цикле не откопировался, а нужен
    return dest;
}

int main() {
    char a[100];          // нужно место куда копировать
    char b[100];          // нужно место куда копировать

    strcpy(a, "qaz");    // эталон: стандартная функция
    printf("%s\n", a);   // qaz

    mystrcpy(b, "qaz");  // тестируем нашу функцию
    printf("%s\n", b);   // qaz

    return 0;
}
```

Попробуем написать ее через указатели. Указатель s идет по строке src с начала до '\0', сдвигаясь каждый раз на 1 символ. Указатель p идет по строке dest, сдвигаясь каждый раз на 1 символ.

```
char * mystrcpy2(char *dest, const char *src) {
    char * p;
    const char * s;
    for (s = src, p = dest; *s != '\0'; s++, p++)
        *p = *s;
    *p = '\0';     // так как он в цикле не откопировался, а нужен
    return dest;
}
```

Попробуйте сами разобраться, почему эта функция работает точно так же, как и предыдущая:

```
char * mystrcpy3(char *dest, const char *src) {
    char * p = dest;
    const char * s = src;
    // ставим лишние () чтобы сказать компилятору, что мы не ошиблись,
    // потеряв знак в ==, а используем =   специально.
    while((*p++ = *s++))
        ;
    return dest;
}
```

strcat, strncat – конкатенация (склейка строк)

char *strcat(char *dest, const char *src);

char *strncat(char *dest, const char *src, size_t n);

Функция strcat() добавляет строку str к строке dest, перезаписывая символ '\0' в конце dest и добавляя к строке символ окончания '\0'. Строки не могут перекрываться, а в строке dest должно хватать свободного места для размещения объединенных строк.

Функция strncat() работает аналогичным образом, но добавляет к dest только первые n символов строки src (и дописывает в конец еще и '\0').

Функции strcat() и strncat() возвращают указатель на строку, получившуюся в результате объединения dest.

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

int main() {
    char a[100];           // нужно место куда копировать

    strcpy(a, "abc");
    printf("%s\n", a);     // abc
    strcat(a, "Hello");
    printf("%s\n", a);     // abcHello

    strncat(a, "Bye!", 2);
    printf("%s\n", a);    // abcHelloBy

    return 0;
}
```

Можно написать самим функцию mystrcat в 1 строку, используя функции strlen и strcpy.

strchr, strrchr – поиск символа в строке

char *strchr(const char *s, int c);

char *strrchr(const char *s, int c);

Функция strchr() возвращает указатель на местонахождение **первого** совпадения с символом c в строке s.

Функция strrchr() возвращает указатель на местонахождение **последнего** совпадения с символом c в строке s.

Функции strchr() и strrchr() возвращают указатель на совпадения с соответствующим символом, а если символ не найден, то возвращают **NULL**.

```
const char * s = "({[<";
char * p;
int c = getchar();
p = strchr(s, c);
if (p != NULL)
    printf("Символ %c является открывающей скобкой\n", c);
```

strstr – поиск подстроки в строке

char *strstr(const char *str, const char *substr);

Функция strstr() ищет первое вхождение подстроки substr в строке str. Завершающий символ '\0' не сравнивается.

Возвращает указатель на начало подстроки, или NULL, если подстрока не найдена.

```
char * text = "I have a dog. I have a bomb. I have a cat";
if (NULL != strstr(text, "bomb"))
    printf("WAAA! BOMB!!!!\n");
```

Прочие функции списком

strtok (TODO)

Преобразование из числа в строку – sprintf

Мы умеем печатать часы и минуты в виде 05:12 или 21:07. Как так же быстро переводить часы и минуты в строку по нужному формату?

Используйте функцию **sprintf**, которая работает почти так же и имеет почти такие же параметры, что и **printf**, но первым аргументом нужно указать строку, куда будет sprintf писать (она НИЧЕГО не печатает, печатать надо отдельно).

```
int h = 21, m = 7;
char s[100];                        // память для строки должна быть выделена
sprintf(s, "%02d:%02d", h, m);      // тут никто не печатает на экран, а заполняет массив s
printf("%s\n", s);                  // тут печатаем эту строку на экран (если нужно)
```

Преобразование из строки в число – scanf

Как вы догадались, аналогичная функция есть и для scanf. Это **sscanf**

Разберем строку "21:07" в переменные h и m (часы и минуты)

```
int h, m;
sscanf("21:07", "%d:%d", &h, &m);
```

Аналогично можно использовать эту функцию для разбора нецелых чисел. Кроме scanf есть специальные функции, преобразующие строку в число по произвольному базису с контролем ошибок.

strtol, strtod, strtou – преобразование строки в число с контролем ошибок

```
#include <stdlib.h>
```

strtol, strtoll – перевод строки в длинное целое (long int)

strtoul, strtoull – конвертирует строку в беззнаковое целое число (unsigned long integer)

strtod, strtof, strtold – конвертируют строки ASCII в число с плавающей запятой

Прототипы функций:

- **long int strtol(const char *nptr, char **endptr, int base);**
- **long long int strtoll(const char *nptr, char **endptr, int base);**
- **unsigned long int strtoul(const char *nptr, char **endptr, int base);**
- **unsigned long long int strtoull(const char *nptr, char **endptr, int base);**
- **double strtod(const char *nptr, char **endptr);**
- **float strtof(const char *nptr, char **endptr);**
- **long double strtold(const char *nptr, char **endptr);**

Функция strtol() конвертирует начальную часть строки nptr в длинное целое в соответствии с указанным base, которое должно находиться в диапазоне от 2-х до 36-х включительно или быть равным нулю.

endptr может быть равным NULL и тогда на него не обращают внимание.

Если endptr указан не NULL, то в него пишут указатель первого некорректного символа из nptr.

Если в строке вообще нет цифр, то strtol() сохраняет начальное значение nptr в *endptr (и возвращает 0). В частности, если *nptr не равно '\0', а **endptr равно '\0' по возвращении, то вся строка состоит из корректных символов.

```
long int x;
char * perr;
x = strtol("12345", NULL, 10);
printf("%ld\n", x+2);                   // 12347

x = strtol("12345abc", &perr, 10);      // в perr хотим получить указатель на первый неправильный символ в строке
printf("%ld wrong=%s\n", x+2, perr);    // 12347 wrong=abc
```

А кто конвертирует строку в число функциями atoi, atol, atod, тот не контролирует ошибки и злобный еретик.

Функции чтения, использующие динамическую память

Если вы еще не знаете, что такое функции malloc, realloc, free, то пропустите этот раздел и вернитесь к нему после изучения работы с динамической памятью.

getline – читаем строку, выделяя память динамически.

Часто бывает, что мы заранее не знаем максимальный размер строки и не можем задать размер массива, чтобы хватило "с запасом". Читать в несколько подходов тоже неудобно.

Если у вас задача не специфическая (например, расчет обтекания вращающего твердого тела в жидкой среде), то ее скорее всего уже решили и внесли в стандартные функции.

```
#include <stdio.h>

ssize_t getline(char **lineptr, size_t *n, FILE *stream);
ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);
```

getline() считывает целую строку, сохраняя адрес буфера, содержащего текст, в *lineptr. Буфер завершается null и содержит символ новой строки, если был найден разделитель для новой строки. Если *lineptr равно NULL, то процедура getline() будет создавать буфер для содержимого строки, который затем должен быть высвобожден программой пользователя. Как альтернатива, перед вызовом getline(), *lineptr может содержать указатель на буфер, размещенный через malloc() с размером *n байтов. Если буфер недостаточно велик для размещения всей считанной строки, то getline() изменяет размер буфера с помощью realloc(), обновляя *lineptr и *n при необходимости. В любом случае при успехном вызове *lineptr и *n будут обновлены для отражения адреса буфера и его размера соответственно.

getdelim() работает аналогично getline(), за исключением того, что разделитель строки, отличающийся от символа новой строки будет определен, как аргумент delimiter. Как и с getline(), символ-разделитель не добавляется, если на вводе не появилось знака разделения и уже достигнут конец файла.

При нормальном завершении работы getline() и getdelim() возвращают номер считанных символов, включая символ разделителя, но не включая завершающий символ null. Это значение может использоваться для обработки встроенных символов null при чтении строки. Обе функции возвращают -1 при ошибках чтения строки (включая условие достижения конца файла).

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

int main(void)
{
        FILE * fp;
        char * line = NULL;
        size_t len = 0;
        ssize_t read;
        fp = fopen("/etc/motd", "r");
        if (fp == NULL)
                exit(EXIT_FAILURE);
        while ((read = getline(&line, &len, fp)) != -1) {
                printf("Retrieved line of length %zu :\n", read);
                printf("%s", line);
        }
        if (line)
                free(line);
        return EXIT_SUCCESS;
}
```

Функция изначально была расширением GNU и была внесена в стандарт POSIX.1-2008.

scanf("%ms", str)

Аналогично память выделяется динамически при задании форматера чтения **%ms**. Указанная строка выделяется динамически и ее нужно потом освободить.

В стандарте начиная с C99.

```
char * name;
scanf("%ms", &name); // compiled with -std=c99 this will allocate the correct amount
                     // of memory for you. You can use "%as" if you're using -std=c89
```

Не забудьте потом написать free(name).

Вопросы для самопроверки

Вопрос 1.

```
char a[] = "hello";    // массив из 6 символов
char * s = "world";    // 1 указатель на char, указывает на строковую константу "world"
```

Можно ли написать (независимые вопросы):

- a = s;
- s = a;
- printf("%c", *a);
- printf("%c", *s);
- printf("%c", *(a+2));
- printf("%c", *(s+2));
- printf("%c", a[2]);
- printf("%c", s[2]);
- a[0] = 'H';
- s[0] = 'W';
- printf("%s", a);
- printf("%s", s);
- printf("%s", a+2);
- printf("%s", s+2);

Вопрос 2

Как хранится в памяти:

- char a1[10] = "abc";
- char a2[] = "abc"
- char a3[] = ""
- char * s1 = "abc";

Вопрос 3

Чему равен sizeof(""), почему?

-- TatyanaDerbysheva - 16 Nov 2017

(c) Материалы раздела "Язык Си" публикуются под лицензией GNU Free Documentation License.