

Строковый тип данных

В языке C для строк не предусмотрено отдельного типа данных, строки могут определяться следующим образом:

- как строковые константы;
- как массивы символов;
- через указатель на символьный тип;
- как массивы строк.

Любая последовательность символов, заключенная в двойные кавычки " ", рассматривается как **строковая константа**. Одинарные кавычки ' ' задают отдельный символ.

Для корректного вывода любая строка должна заканчиваться нуль-символом '\0', целочисленное значение которого равно 0. При объявлении строковой константы нуль-символ добавляется к ней автоматически. Так, последовательность символов, представляющая собой строковую константу, будет размещена в оперативной памяти компьютера, включая нулевой байт.

Под хранение строки выделяются последовательно идущие ячейки памяти. Таким образом, строка представляет собой массив символов. Для хранения кода каждого символа строки отводится 1 байт (тип **char**).

Строковые константы размещаются в статической памяти. Начальный адрес последовательности символов в двойных кавычках трактуется как адрес строки. Строковые константы часто используются для осуществления диалога с пользователем в

```
char c = 'C'; //отдельный символ
```

```
char m[25]; //массив символов задаёт строку. Необходимо сообщить размер
```

```
//компилятор определяет сам размер массива символов, если инициализация задана  
//при объявлении массива
```

```
char m2[]="I learn" //две подряд идущие пары "" задают одну строку  
        " the";    //так можно переносить строковые константы в коде
```

```
char m3[]={ 'l', 'a', 'n', 'g', 'u', 'a', 'g', 'e', '!', '\0' };
```

```

int i;
for(i = 0; i<23;i++) //объединяем массивы символов
    m[i]= i<11? m2[i] : i == 11 || i == 13 ? ' ': i == 12? c : m3[i-14];

printf("%s", m); // I learn the C Language!

```

этом случае имена m2 и m3 являются указателями на первые элементы массивов:

- m2 - эквивалентно &m2[0]
- m2[0] - эквивалентно 'I'
- m2[3] - эквивалентно 'e'

Для задания строки можно использовать **указатель на символьный тип**:

```
char *m4;  
m4 = m3;  
*m4      // эквивалентно m3[0]  
*(m4+2)  // эквивалентно m3[2]
```

Здесь m3 является константой-указателем. Нельзя изменить m3, так как это означало бы изменение положения (адреса) массива в памяти, в отличие от m4.

Для указателя можно использовать операцию увеличения (перемещения на следующий символ): m4++

Иногда в программах возникает необходимость описание **массива символьных строк**. В этом случае можно использовать индекс строки для доступа к нескольким разным строкам:

```
char *text[4] = {"first line", "second line", "third line"};
```

Инициализация выполняется по правилам, определенным для массивов. Тексты в кавычках эквивалентны инициализации каждой строки в массиве. Запятая разделяет соседние последовательности.

Кроме того, можно явно задавать размер строк символов, используя описание, подобное такому: **char text[4][25]**

Разница заключается в том, что такая форма задает «прямоугольный» массив, в котором все строки имеют одинаковую длину.

Описание же **char* text[4]** определяет свободный массив, где длина каждой строки определяется тем указателем, который эту строку инициализирует. Свободный массив не тратит память напрасно.

Операции со строками

Большинство строковых операций языка **C**, работает с указателями. Для размещения в оперативной памяти строки символов необходимо:

- выделить блок оперативной памяти под массив;
- проинициализировать строку.

Для выделения памяти под хранение строки могут использоваться функции динамического выделения памяти. При этом необходимо учитывать требуемый размер строки:

```
char *name = malloc(10); //выделяем память под 10 символов
scanf("%9s", name); //%9s ограничит ввод первыми 9 символами, 10й будет '\0'
printf("%s", name);
free(name);
```

Вывод строк можно осуществлять через **printf**:

```
printf("%s", str); // str – указатель на строку  
printf(str); // или сокращённо
```

Для вывода строк также может использоваться функция **puts**:

```
int puts (char *s);
```

которая печатает строку *s* и переводит курсор на новую строку (в отличие от **printf**). Для вывода символов может использоваться функция

```
char putchar(char);
```

Для ввода строки может использоваться функция **scanf()**. Однако она предназначена скорее для получения слова, а не строки. Если применять формат "%s" для ввода, строка вводится до (но не включая) следующего пустого символа, которым может быть пробел, табуляция или перевод строки. Для ввода строки, включая пробелы, используется функция:

Количество символов, считываемых `gets()`, не ограничивается. Поэтому программист должен сам следить за тем, чтобы не выйти за границы массива, на который указывает `str`. Поэтому вместо неё можно использовать более безопасную версию функции:

```
char* fgets(char *s, size_t n, stdin);
```

Где `n` – максимальное количество символов, выделенных в буфере `s`, а `stdin` – стандартный поток ввода.

Для ввода символов может использоваться функция

```
char getchar();
```

которая возвращает значение символа, введенного с клавиатуры.

Основные функции стандартной библиотеки string.h

Функция	Описание
<code>char *strcat(char *s1, const char *s2)</code>	присоединяет s2 к s1, возвращает s1
<code>char *strncat(char *s1, const char *s2, int n)</code>	Как и strcat, но не более n символов, завершает строку символом '\0'
<code>char *strcpy(char *s1, const char *s2)</code>	копирует строку s2 в строку s1, включая '\0', возвращает s1
<code>char *strncpy(char *s1, const char *s2, int n)</code>	Как и strcpy, но не более n символов
<code>int strcmp(const char *s1, const char *s2)</code>	сравнивает s1 и s2, возвращает значение 0, если строки эквивалентны
<code>int strncmp(const char *s1, const char *s2, int n)</code>	Как и strcmp, но не более n символов
<code>int strlen(const char *s)</code>	возвращает количество символов в строке s
<code>char *strset(char *s, char c)</code>	заполняет строку s символами, код которых равен значению c, возвращает указатель на строку s
<code>char *strnset(char *s, char c, int n)</code>	Как и strset, только первые n символов строки

Функция	Описание
<pre>int sprintf(char *buf, const char *format, arg-list)</pre>	идентична printf(), вывод производится в массив, указанный аргументом buf. Возвращаемая величина равна количеству символов, действительно занесенных в массив.
<pre>char *strchr(const char *str, int ch)</pre>	возвращает указатель на первое вхождение символа ch в строку, на которую указывает str. Если символ ch не найден, возвращается NULL.
<pre>int atoi(const char *str) float atof(const char *str) long long atoll(const char *str)</pre>	преобразуют строку str соответственно в int, float или long long. Число может завершаться любым символом, который не входит в состав строкового представления числа.
<pre>int toupper(int ch) int tolower(int ch)</pre>	возвращает соответствующий верхнему\нижнему регистру эквивалент символа ch, если ch — это буква. В противном случае ch возвращается неизменным. <u>Нужно подключить ctype.h</u>

Следует иметь в виду, что все эти функции не производит проверки границ, поэтому программист должен сам позаботиться о том, чтобы заполняемый символьный массив не был переполнен.

Рассмотрим пример замены в тексте строки на другую строку той же длины:

```
#define MAX_TEXT 100

int main()
{
    //Пример замены в тексте строки на другую строку той же длины
    char s1[20], s2[20], text[MAX_TEXT+1]; //максимум MAX_TEXT символов + '\0'

    system("chcp 1251"); //подключаем русскую кодировку

    puts("Введите текст:");
    fgets(text, MAX_TEXT, stdin); //получаем текст

    if(strlen(text) < 5)
        puts("Введите текст больше 5 символов"), exit(1);
```

```
puts("Введите два слова для замены \"чего\" на \"что\" через пробел:");
scanf("%19s %19s",s1,s2); //получаем строки замены
int s1Len=strlen(s1), s2Len=strlen(s2);

if(s1Len != s2Len)
    puts("работает только для одинаковых по длине строках :("), exit(1);

int textLen=strlen(text);
int i=0;
while(i<textLen-s1Len) //в цикле проходим все символы text
{
    if(strncmp(&text[i],s1,s1Len)==0) //и если нашли совпадение
    {
        memcpy(&text[i],s2,s2Len * sizeof(char)); //копируем подстроку в строку
        i+=s1Len;
        continue;
    }
    i++;
}

puts(text);

return 0;
}
```

D:_ | _ | _ | \LLP_хъЎшш_ёшьхёv\str\str.exe

Текущая кодовая страница: 1251

Введите текст:

Мама мыла раму

Введите две строки для замены "чего" на "что" через пробел:

раму стул

Мама мыла стул

Process exited after 27.63 seconds with return value 0

Для продолжения нажмите любую клавишу . . .

Параметры запуска

При создании консольного приложения в языке **C**, автоматически создается функция `main` следующего вида:

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

Если программу запускать через командную строку, то есть возможность передать какую-либо информацию этой программе, указывая аргументы через пробел:

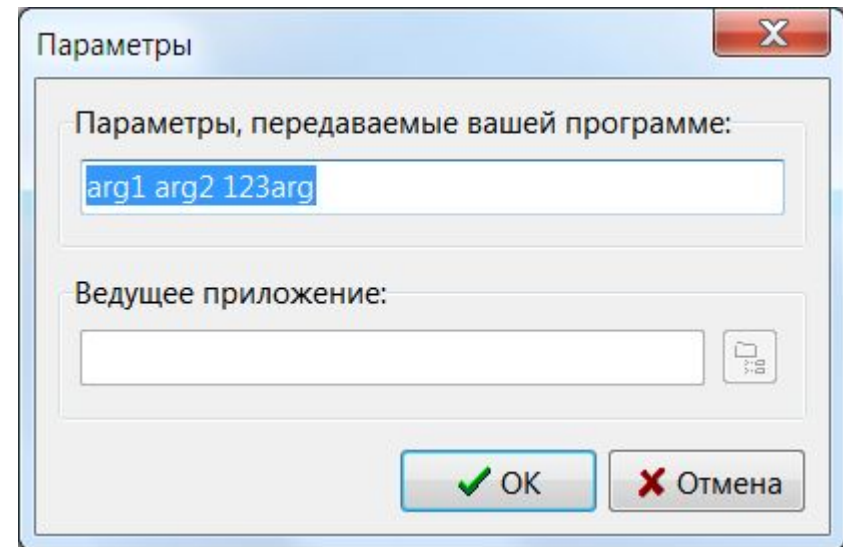
Диск:\путь\имя.exe аргумент1 аргумент2 аргумент3

Параметр `argc` указывает количество переданных аргументов, а `argv` является массивом указателей на строки – сами аргументы. Причем `argc` всегда не меньше 1, так как первым аргументом всегда передаётся полный путь к исполняемому файлу и полное имя.

```
int main(int argc, char** argv)
{
    // если передаем аргументы, то argc будет больше 1 (в 0 хранится полный путь к exe)
    if (argc > 1)
    {
        int i;
        for(i=1;i<argc;i++)
            printf("argument[%d] = %s\n",i, argv[i]);
    }
    else
        puts("Not arguments");

    return 0;
}
```

В среде Dev-C++ аргументы запуска можно задать в меню **ВЫПОЛНИТЬ -> параметры...**



D:\пфшъ\ёхяюфртрэшх\шчъюёёютэхтх_яёюуёрььшёютрэшх\тхъ...

```
argument[1] = arg1  
argument[2] = arg2  
argument[3] = 123arg
```

Process exited after 0.005054 seconds with return value 0

Для продолжения нажмите любую клавишу . . .

Компиляция программ

Исходный С файл — это всего лишь текстовый файл с С кодом, который невозможно запустить как программу или использовать как библиотеку. Поэтому каждый исходный файл требуется скомпилировать в исполняемый файл, динамическую или статическую библиотеки.

Компиляция — это процесс трансляции программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком к машинному коду, или непосредственно на машинном языке и последующую сборку исполняемой машинной программы.

Процесс компиляции программ на языке C обычно состоит из следующих этапов:

1) Препроцессинг – подготовка исходного текста программы для дальнейшей компиляции. На данном этапе происходит удаление всех комментариев и модификация программы в соответствии с заданными препроцессорными директивами (сами директивы также удаляются).

2) Компиляция – на данном этапе выполняется преобразование (трансляция) модифицированной на предыдущем шаге программы в ассемблерный код.

3) Ассемблирование – Так как процессоры исполняют команды только в бинарном виде, необходимо перевести ассемблерный код в машинный с помощью **ассемблера**, сохраняя его в объектном файле. **Объектный файл** — это созданный ассемблером промежуточный файл, хранящий кусок машинного кода программы. Этот кусок, еще не связанный вместе с другими кусками машинного кода в конечную выполняемую программу, называется объектным кодом. Объектных файлов может быть много и нужно их всех соединить в один

4) Компоновка - компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл. Процесс связывания происходит с помощью таблицы символов. При этом возможны ошибки связывания: например если функция была объявлена, но не определена, ошибка обнаружится только на этом этапе.

Таблица символов — это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, объектов и т.д., где каждому идентификатору (символу) соотносится его тип и область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах. Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

После этого этапа получается исполняемый файл, который можно уже запустить на компьютере. После запуска исполняемая программа загружается в память компьютера и начинает выполнение. На данном этапе также возможна подгрузка динамических библиотек.

