Обзор функций работы со строками

А. Г. Фенстер, fenster@fenster.name

13 марта 2010 г.

На этом занятии мы рассмотрим основные функции, работающие со строками. Как обычно, основное внимание будет уделено не нудному описанию параметров, а неочевидным моментам, возникающим при использовании этих функций.

Рекомендую прочитать статью «Назад к основам» (автор статьи — Joel Spolsky, желающие могут прочитать её же в оригинале).

Начнём разговор о строках несколько издалека — с инициализаторов массивов.

1 Инициализация массивов

Элементам массива может быть присвоено значение при определении массива. В простейшем случае, когда количество элементов при инициализации совпадает с количеством элементов массива, вопросов не возникает:

```
int A[5] = { 1, 2, 3, 4, 5 };
```

Кстати, в конце списка значений можно поставить запятую и это не будет являться синтаксической ошибкой (зато дописать несколько значений будет проще).

Если в инициализаторе меньше элементов, чем нужно, остальные элементы массива (те, которым «не хватило» инициализатора) будут занулены. Это свойство удобно использовать для зануления всего массива:

```
int A[100] = { 0 };
```

GCC позволяет даже `int A[100] = {};` но, согласно стандарту, это нарушение синтаксиса: в инициализаторе должен быть хотя бы один элемент.

Указание большего числа элементов в инициализаторе, чем нужно, является ошибкой (и вызывает ошибку или предупреждение при компиляции).

Если присутствует инициализатор массива, можно не указывать его размер: он будет вычислен автоматически.

Новый стандарт C99 позволяет инициализировать элементы массива выборочно:

```
int A[5] = { [2] = 10, [4] = 20 }; /* C99, не C++ */
```

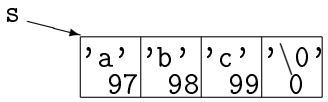
Такой код нормально компилируется свежим GCC, но далеко не все компиляторы умеют компилировать C99 (Visual Studio не умеет), а главное — это не является корректным кодом на C++, поэтому вряд ли стоит использовать такую инициализацию.

2 Кратко о строках

Как известно, специальный тип «строка» в языке C отсутствует. Под строками мы понимаем массивы элементов типа `char` (иногда `unsigned char`), ограниченные символом `'\0'`, имеющим код 0. Практически все функции, работающие со строками, останавливаются, увидев этот ноль.

Инициализировать строку **при создании** можно несколькими различными способами. Следующие строки эквивалентны и создают изменяемый массив символов длины 4:

```
char s[4] = { 'a', 'b', 'c', '\0' };
char s[4] = { 'a', 'b', 'c', 0 };
char s[4] = "abc";
char s[] =  { 'a', 'b', 'c', '\0' };
char s[] =  { 'a', 'b', 'c', 0 };
char s[] = "abc";
```



и они **не** эквивалентны инициализации

```
char *p = "abc";
```

при которой `p` указывает на константную строку и попытка изменить содержимое строки приведёт к ошибке Segmentation fault.

Использовать для присваивания значений строкам обычный оператор присваивания `=` нельзя. Если `a` — массив, то присваивание `a = ...` вызовет ошибку компиляции (не путайте присваивание и инициализацию), если же `p` — указатель, то присваивание `p = ...`, естественно, не приведёт к копированию содержимого строки, а лишь изменит «стрелочку».

Также для строк нельзя использовать операторы сравнения `>`, `<` и прочие: они будут сравнивать места расположения строк в памяти, но не их содержимое.

Для выполнения этих (а также многих других) операций со строками стандарт языка C описывает специальные функции, объявленные в `string.h`.

3 Длина строки

```
size_t strlen(const char *s);
```

Возвращает количество символов от начала `s` до завершающего символа с кодом 0. Естественно, как и любая другая функция, принимающая указатель, `strlen` не имеет возможности определить, где находится граница массива, и может выйти за неё, если программист забыл завершить строку нулём.

4 Копирование строк

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

Понятно, что функция `strcpy`, бездумно копирующая содержимое `src` в `dest`, может выйти за пределы памяти, выделенной для `dest` (поэтому использовать её нужно с большой осторожностью). К сожалению, функция `strncpy`, не копирующая более `n` символов, решает проблему лишь частично: если длина строки `src` больше либо равна `n`, строка-результат `dest` **не будет завершена нулём** (если забыть об этом, Segmentation fault в будущем гарантирован).

Кроме того, нельзя использовать `strcpy` и `strncpy`, если результат и исходная строка могут пересечься в памяти. Например, вызов `strcpy(s + 1, s)` вызовет выход за пределы массива (завершающий 0 будет уничтожен при копировании) и, вероятнее всего, в результате произойдёт любимая многими ошибка Segmentation fault.

Обе функции в качестве результата возвращают `dest` (по аналогии с оператором присваивания, результатом которого является присвоенное значение).

Близкими по смыслу функциями являются функции, копирующие произвольные данные, для которых символ с кодом 0 не является ограничителем:

```
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```

Функция `memcpy` требует, чтобы массивы не пересекались, функция `memmove` выполняет копирование через временный буфер и может работать даже с пересекающимися массивами.

5 Конкатенация строк

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

Дописывают строку `src` в конец строки `dest`: первый символ строки `src` встанет на место, где раньше стоял завершающий 0 строки `dest`.

В отличие от функций копирования, эти две функции всегда завершают строку `dest` нулём. Кроме того, отсчёт `n` ведётся по строке `src`, а не `dest`. Таким образом, для корректного выполнения функции `strncat` необходимо, чтобы под `dest` было выделено как минимум `strlen(dest) + n + 1` байт.

Строки `src` и `dest` не должны пересекаться в памяти.

6 Сравнение строк

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

Строка `s1` считается меньшей, чем строка `s2`, если первые 0 или более символов этих строк совпадают, а затем в `s1` идёт символ с меньшим кодом, чем соответствующий символ в `s2`. Заметьте, что благодаря завершающему символу с кодом 0 в это правило замечательно вписывается случай, когда одна строка является началом другой строки: в таком случае меньшая по длине строка окажется меньше.

Первые две функции сравнивают символы согласно их коду. В частности, это говорит о том, что если используется кодировка KOI8-R, то русские буквы не будут упорядочены по алфавиту. Функция `strcoll` при сравнении использует правила, определённые в текущей локали. Не углубляясь в эту сложную тему, нужно сказать, что локаль может быть выбрана при помощи `setlocale` (`locale.h`). Следующий вызов позволит программе в Linux использовать выставленную у пользователя локаль вместо локали по умолчанию:

```
setlocale(LC_ALL, "");
```

7 Поиск в строке

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

Возвращают адрес первого или последнего найденного в строке символа. Если символ не найден, вернут `NULL` — обязательно проверяйте возвращаемое значение!

```
char *strstr(const char *haystack, const char *needle);
```

Для уменьшения путаницы параметры этой функции, производящей поиск подстроки в строке, часто называют именно так: стог (`haystack`) и иголка (`needle`). В случае, если подстрока не была найдена, возвращает `NULL`.

8 Прочие функции

```
void *memset(void *s, int c, size_t n);
```

Полезная функция, заполняющая массив `s` размера `n` байт значениями младшего байта числа `c`. В большинстве случаев, конечно, функции передаётся `c == 0` для зануления всех элементов массива.

```
char *strdup(char *s);
```

Удобная функция, описанная в стандарте POSIX, но (к сожалению) не входящая в стандарт C; тем не менее, она доступна и в GCC, и в Visual Studio. Выделяет при помощи `malloc` необходимое количество памяти под копию строки `s`, копирует туда строку и возвращает адрес новой строки (её необходимо будет освободить при помощи `free`, когда она станет ненужной). Возвращает `NULL`, если память выделить не удалось.