

## Лекция III

11 октября 2017

# Адрес переменной

Каждая переменная любого типа в С связана с сопоставленным её блоком в оперативной памяти. Длина блока изменяется в байтах, для разных типов - различна.

**Адресом** переменной называют **номер первого байта** из блока, который отведён под неё. В языке С это целое положительное число.

У переменной можно узнать адрес с помощью оператора - **&**. Вывести значение адреса переменной можно с помощью спецификатора **%p** функции **printf**.

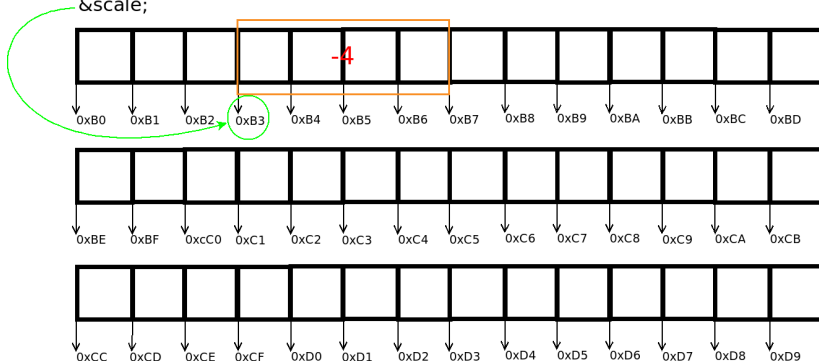
```
1 int scale = 5;
2 double rate = 3.4;
3
4 printf("Адрес rate: %p\n", &rate);
5 printf("Адрес scale: %p\n", &scale);
```

Значения адреса выводятся на экран в виде шестнадцатиричных чисел.

# Адрес переменной

Как адрес выглядит графически: переменная **scale** имеет адрес равный **0xB3** и состоит из четырёх байт.

```
int scale = -4;  
&scale;
```



**Указателем** - называют тип данных, переменные которого предназначены для хранения адресов других объектов (то есть, обыкновенных переменных простых, составных или пользовательских типов). Указатели в С являются типизированными.

Синтаксис объявления указателя

<тип\_данных> \*<имя\_переменной>;

```
1 int    *p_int;    // указатель на int
2 char   *p_char;   // указатель на char
3 double *p_double; // указатель на double
```

Для практического применения указатели должны быть типизированными, чтобы было возможно получать значения тех переменных, адрес которых хранится в них.

Операции с указателями: **присвоение значения**

Указателю может быть присвоено значение (адрес в памяти) либо с помощью операции взятия адреса у переменной, либо копированием значения из другого указателя.

```
1 int scale = 5, *p_sc;  
2  
3 p_sc = &scale;  
4 int *p2 = p_sc;
```

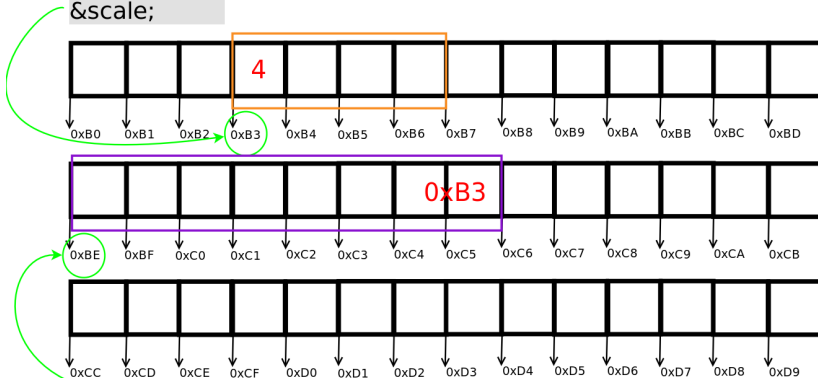
**1-я строка:** определяем переменную целого типа **scale** и указатель на целое **p\_sc**.

**3-я строка:** получаем адрес **scale** и записываем его значение в переменную-указатель **p\_sc**.

**4-я строка:** копируем значение (целое число) из **p\_sc** в переменную-указатель **p2**.

В памяти картина следующая. Обратите внимание, сама по себе **p\_sc** - просто переменная, со своим адресом.

```
int scale = 4;  
&scale;
```



```
int *p_sc = &scale;  
printf("Адрес scale: %p\n", p_sc);  
printf("Адрес p_sc: %p\n", &p_sc);
```

Операции с указателями: **присвоение значения**.

Существует специальное значение для указателей, которое означает, что сама переменная-указатель не содержит реальный адрес какого-либо байта. Для обозначения такого спец. значения в С применяется специальная константа **NULL**. Также эта константа известна как **нулевой адрес**. Сам указатель с таким значением иногда называется как **нулевой указатель**.

```
1 double *p1 = NULL;
2 ...
3
4 if (p1 != NULL) {
5     // работаем с указателем p1
6 }
```

**Правило для работы с указателями:** переменная-указатель **всегда** должна быть определена, а не объявлена. То есть, ей надо или присвоить реальный адрес, или значение **NULL**.

Операции с указателями: **разыменование** - получение значения переменной, адрес которой сохранён в указателе.

Синтаксис

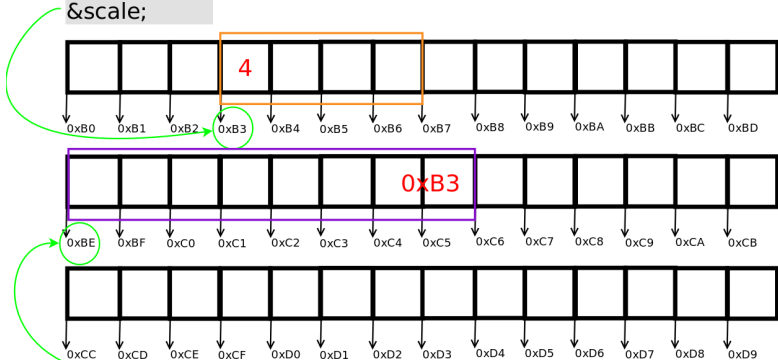
`*<имя_переменной>;`

```
1 int scale = 5;
2 int *p_sc = &scale;
3
4 // Вывод 5 на экран
5 printf("p_sc ссылается на значение: %d\n", *p_sc;
6
7 // При разыменовании указатель может
8 // участвовать в арифметических выражениях
9 int rate = (*p_sc) + 15;
10
11 // изменяем значение scale через указатель p_sc
12 *p_sc = 15;
```



Схематично разыменование можно показать так:

```
int scale = 4;  
&scale;
```



```
int *p_sc = &scale;  
printf("Значение scale через указатель: %d\n", *p_sc);  
*p_sc = 15;  
printf("А теперь: %d", *p_sc);
```

Операции с указателями: **разыменование**

**Предупреждение:** разыменование объявленной, но не определённой, переменной-указателя **чрезвычайно опасно** в коде на C. Тоже самое верно и для указателей, которым присвоен **NULL**.

**Так никогда не делать!**

```
1 char *p_ch;  
2 printf("Что же за символ у нас: %c", *p_ch);  
3  
4 int *p_int = NULL;  
5 printf("Может с int получится: %d", *p_int);
```

**Правило для работы с указателями:** никогда не разыменовывать объявленный или **нулевой** указатель.

Операции с указателями: **разыменование**

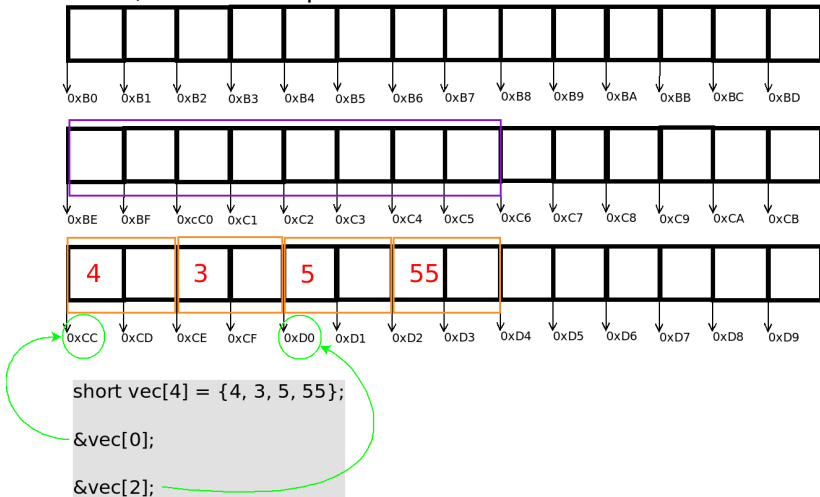
Указатели дают возможность *менять* значения переменных, которые передаются в функции. Как пример:

```
1 void my_swap(int *i1, int *i2)
2 { // Обмен значениями двух переменных
3     if (i1 != NULL && i2 != NULL) {
4         int tmp = *i1;
5         *i1 = *i2;
6         *i2 = tmp;
7     }
8 }
9
10 int n1 = 11, n2 = -15;
11 my_swap(&n1, &n2);
12 printf("n1 = %d, n2 = %d\n", n1, n2);
```

**Стоит отметить**, что сами переменные-указатели передаются в функции **по-значению**. Здесь отличий от переменных обычных типов нет.

# Указатели и массивы

Вспомним, как массив располагается в памяти

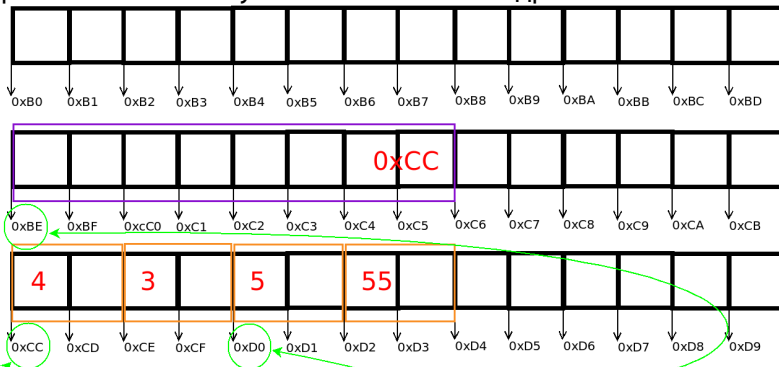


## Связь указателей и массивов

- Имя переменной-массива (выше - **vec**) является указателем на его первый элемент
- Массивы передаются в функцию как указатели. Поэтому, в любую функцию, принимающую указатель, можно передать переменную-массива. И наоборот
- Но: переменной массива **нельзя** присвоить никакой другой адрес (в отличие от переменной-указателя)

```
1 void print_array(short* arr, size_t count);  
2 ...  
3 short vec[4] = {4, 3, 5, 55};  
4  
5 print_array(vec, 4);
```

Картина в памяти: в указатель записали адрес массива



```
short vec[4] = {4, 3, 5, 55};
```

```
&vec[0];  
&vec[2];
```

```
short *p_vec = vec;
```

```
printf("Число: %d", *p_vec);
```

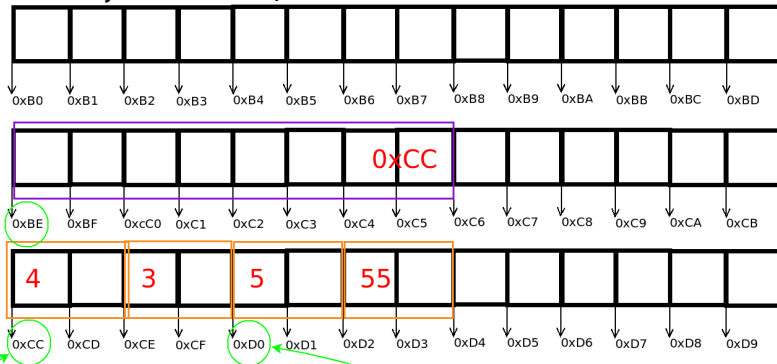
Операции с указателями: **сложение с целым числом**

Результатом операции прибавления целого числа **n** к указателю является новый указатель, значение (адрес) которого смещено на  $n * \text{sizeof}(< \text{type} >)$  байт (вправо или влево - зависит от знака **n**).

```
1 short vec[4] = {4, 3, 5, 55};  
2 short p_vec = vec;  
3  
4 // Перемещаемся на третий элемент  
5 printf("Значение третьего элемента: %d\n",  
6        *(p_vec + 2));
```

Смещение происходит блоками, размер которого определяется типом указателя (указатель на **int**, **double**, **char** и прочие).

## Сложение указателя с целым числом на схеме с памятью



```
short vec[4] = {4, 3, 5, 55};
```

```
short *p_vec = vec;
```

```
p_vec;
```

```
p_vec + 2;
```



Операции с указателями: **сложение с целым числом**

Как видно из примеров, особенно полезно сложение при использовании указателя для работы с элементами массива.

```
1 short vec[4] = {4, 3, 5, 55};  
2 short *p_vec = vec;  
3  
4 // Инкремент, как сложение  
5 // с единицей, также допустим  
6 p_vec++;  
7 p_vec += 1;  
8 p_vec -= 2;
```

Операции с указателями: **индексация**

**Индексация** указателя выполняет два действия:

- 1 Сместиться на количество блоков, равных индексы, от текущего адреса
- 2 Получить значение по адресу, **после смещения**

```
1 short vec[4] = {4, 3, 5, 55};
2 short *p_vec = vec;
3
4 if (p_vec[2] == *(p_vec + 2)) {
5     puts("Значения равны");
6 }
7
8 printf("4-ый элемент: %d", vec[3]);
```

Операции с указателями: **вычитание однотипных указателей**

Результатом вычитания является целое число (как положительное, так и отрицательное), показывающее количество блоков памяти между двумя указателями. Под блоком памяти, напоминаем, понимается размер типа указателя.

```
1 short vec[4] = {4, 3, 5, 55};
2 short *p1 = vec, *p2 = &vec[3];
3
4 int diff = p2 - p1;
5 // Печатает 3
6 printf("От начала массива до последнего "
7        "элемента поместились ещё %d эл-ов\n",
8        diff);
9
10 diff = p1 - p2;
11 // Печатает -3
12 printf("А так: %d\n", diff);
```

В отличие от переменных, можно объявлять **указатель на тип void**.

- 1 Указателю на **void** может быть присвоено значение любого другого указателя
- 2 Указатель на **void** может быть присвоен любой другой указатель только с использованием **явного приведения типа** (про приведение - два слайда ниже).
- 3 Для указателей на **void** запрещены операции **разыменования, индексации и вычитания указателей**.

```
1 short vec[4] = {4, 3, 5, 55};  
2 short *p1 = vec, *p2;  
3  
4 void *pv1 = p1;  
5 p2 = (short *) pv1;
```

# Явное приведение значений разных типов в C

# Неявные приведения

В арифметических операторах целое значение приводится к действительному типу тогда и только тогда, когда вещественным является один из операндов.

```
1 int i1 = 5, i2 = 9;
2 double d1 = 2.0;
3
4 double result = (i2 / i1) * d1;
5 printf("Результат (1): %lf\n", result);
6 // Будет выведено 2.0
7
8 result = (i2 / d1) * i1;
9 printf("Результат (2): %lf\n", result);
10 // Будет выведено 22.5
11
12 int tot_part = result;
13 printf("Целая часть равна: %d", tot_part);
14 // Будет выведено 22
```

Значения могут быть приведены к другому типу явно. Общий синтаксис:

( <тип> ) <значение>;

Пример с численными типами

```
1 double rate = 34.505;  
2 int scale = 6;  
3  
4 int tot_rate = (double) rate;  
5 double real_scale = (int) scale;
```

Более полезный случай использования явного приведения - преобразование указателей

```
1 double rate = 34.505;
2 int scale = 6;
3
4 double *d_pointer = &rate;
5 int *i_pointer = &scale;
6
7 char *b_pointer;
8
9 b_pointer = (char*) d_pointer;
10 printf("Первый байт rate равен %d", *b_pointer);
11
12 b_pointer = (char*) i_pointer;
13 printf("Первый байт scale равен %d", *b_pointer);
```



## Как текст представлен в С?

- **char** - ещё один фундаментальный тип данных в языке C
- Используется для хранения одиночных символов. Сами символы указываются в **одинарных** кавычках, например: 'a', 'b', 'd', '4', '%', '!'
- Символы хранятся в виде **целых чисел**. Фактически, **char** является ещё одним целочисленным типом
- Это единственный тип, размер которого ограничен стандартом языка. Размер **char** **всегда** равен 1 байту
- Но всё просто не бывает: стандарт не обговаривает, должен ли тип **char** быть знаковым или беззнаковым
- Существуют специальные символы, которые в текстовом виде представлены более чем одним знаком: '\n', '\t', '\r'
- Также стандартом определены беззнаковый **unsigned char** и знаковый **signed char** типы

**Кодировка** - специальная таблица, связывающая каждый символ с соответствующим ему целым числом

Базовая для ЭВМ - **ASCII**. Как правило, входит в любую другую расширенную кодировку. Основные характеристики

- 128 символов
- целочисленные коды: от нуля до 127
- каждый символ занимает один байт
- Включает в себя все цифры, буквы английского алфавита в нижнем/верхнем регистрах и некоторые другие символы (тильда, процент, '@', решётка и т.п.)
- Коды букв и цифр - идут последовательно

```
1 char symbol = '%';
2 putchar(symbol); // печатаем знак процента
3
4 symbol = '#'; // знак переноса строки
5 printf("\nВыводим символ %с %с и ещё раз: %с↵
    \n",
6       symbol, '\n', symbol);
7
8 // А поскольку char — целочисленный:
9 symbol = '5';
10 symbol += 2;
11
12 // Выводим на экран семёрку
13 putchar(symbol);
```

```
1 char sym = '5';
2 printf("Значение sym: %c\n", sym);
3
4 printf("Код sym: %d", sym);
5
6 int is_less = '2' < '5';
7 // Переменная is_less здесь равна 1
8
9 // Печать всех букв английского алфавита
10 sym = 'a';
11
12 while (sym <= 'z') {
13     printf("%c ", sym);
14     ++sym;
15 }
```

Ещё одна кодировка, про которую полезно знать:

## UTF8

- символ может состоять от 1 до 4 байт (следовательно, от 8 до 32 бит)
- вводится понятие "code point равное 1 байту (8 битам) символа"
- включает в себя все возможные региональные алфавиты
- Коды от 0 до 127 совпадают с кодировкой ASCII

**Строка в C** - это одномерный массив значений типа **char**, последним символом которой считается специальный нулевой символ (он же - символ окончания строки): `\0` (целочисленный код - 0). Обратите внимание, символ конца строки **не обязан совпадать** с последним элементом массива.

```
1 char phrase1[] = "Первая строка!";  
2 // puts — функция из stdio.h для  
3 // вывода строки на консоль  
4 puts(phrase1);  
5  
6 char phrase2[] = { 'A', 'B', 'C', '\0' };  
7 printf("Второй символ в phrase2: %c", ←  
    phrase2[1];  
8  
9 // Массив, который пока строкой не является  
10 char test_str[15];
```

# C-строки. Ввод с консоли

```
1 char word[21];  
2  
3 printf("Введите слово: ");  
4 scanf("%20s", word);
```

Три ключевые особенности ввода:

- 1 **"%20s"** у **scanf** означает ввод строки (до первого пробела или переноса), с ограничением максимальной длины в 20 символов (байт). Хотя размер введённой строки может быть и меньше
- 2 Символ окончания строки проставляется функцией по умолчанию
- 3 Поскольку переменная-массива **phrase** фактически хранит адрес первого элемента, знак **&** в **scanf** указывать не нужно



# С-строки. Ввод с консоли

Безопасный ввод строки

`fgets(str, max_chars, stream_pointer)`

- 1 **str** - массив типа **char**, куда будут сохранена вводимая строка
- 2 **max\_chars** - целое число, обозначающее **максимальное число символов**, которые будут сохранены в **str**
- 3 **stream\_pointer** - указатель на структуру, ассоциированную с вводом текста. Для терминала это всегда переменная **stdin**, которая определена в **<stdio.h>**

```
1 const size_t SZ = 140;
2 ...
3 char phrase[SZ];
4
5 printf("Введите фразу: ");
6 fgets(phrase, 140, stdin);
```

# Работа с текстом в С с помощью стандартной библиотеки

# Работа с текстом. Стандартная библиотека

Библиотека языка C для работы со строками:

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

Получение длины строки **в байтах**

```
size_t strlen(const char* str)
```

```
1 char text[] = "A string";  
2 // На экране покажется число 8  
3 printf("Длина строки: %lu", strlen(text));  
4  
5 char text2[] = "Строка на русском";  
6 /*  
7 А здесь — зависит от ОС и её локальных  
8 настроек. В русскоязычных Windows 7, 8,  
9 10 — скорее всего покажет длину равную 17.  
10 В современных ОС, основанных на Linux,  
11 наиболее вероятное значение — 32  
12 */  
13 printf("Длина строки: %lu", strlen(text2));
```

Посимвольное сравнение строк

```
int strcmp(const char *first,  
           const char *second)
```

Функция возвращает:

- 1 значение  $< 0$  - первый несовпадающий символ в строке **first** меньше, чем в строке **second**;
- 2 значение  $== 0$  - все символы в обеих строках совпадают;
- 3 значение  $> 0$  - первый несовпадающий символ в строке **first** больше, чем в строке **second**;

Сравнение идёт с начала строк и до тех пор, пока в одной из них не встретится символ окончания строки. Фактически сравниваются целочисленные коды каждого символа (байта).

## Сравнение строк

```
1 char str1[] = "Единица",  
2   str2[] = "единица",  
3   str3[] = "Единица";  
4  
5 printf("Сравнение str1 и str2: ")  
6 printf("%d", strcmp(str1, str2));  
7  
8 printf("Сравнение str1 и str3: ");  
9 printf("%d", strcmp(str1, str3));
```

## Сравнение строк

```
1 char key_color[] = "оранжевый", answer[50];  
2  
3 do {  
4     printf("\nУгадайте цвет: ");  
5     scanf("%49s", answer);  
6 } while ( strcmp(key_color, answer) != 0 );  
7  
8 puts("Правильный ответ!");
```

Возможный вывод программы:

```
Угадайте цвет: красный  
Угадайте цвет: жёлтый  
Угадайте цвет: оранжевый  
Правильный ответ!
```

Посимвольное сравнение начальных фрагментов строк

```
int strncmp(const char *first,  
            const char *second,  
            size_t count)
```

Сравнение идёт с начала и до тех пор, пока в одной из строк не встретится символ окончания строки, **либо** не закончится сравнение **count** символов из каждой.

Посимвольное сравнение начальных фрагментов строк

```
1 char str1[] = "12 причин для роста",
2   str2[] = "12 отговорок от учёбы",
3
4 if ( strncmp(str1, str2, 2) == 0 ) {
5     puts("Обе строки начинаются с числа 12.");
6 }
7
8 // Как переносимым образом сравнивать
9 // фрагменты с текстом, отличным от ASCII
10 if ( strncmp(str1, str2, strlen("12 причин")) ←
11     ) != 0 ) {
12     puts ("Но не с 12 причин.")
13 }
```



## Копирование строки

`char* strcpy(char *dest, const char *source)`

- Функция копирует строку **source** в строку **dest**
- Функция возвращает указатель на строку **dest**. Как правило, возвращаемое значение игнорируется
- Копируются **все** символы из строки, включая и завершающий *символ окончания строки*
- Следить за размером строк следует самостоятельно!

```
1 char source[] = "A string", dest[40];  
2 std::strcpy(dest, source);  
3  
4 std::cout << "\n" << dest;
```

## Копирование фрагмента строки

```
char* strncpy(char      *dest,  
               const char *source,  
               size_t    count)
```

- **count** - целое беззнаковое число, количество символов которое копируется в **dest**
- Если длина копируемой строки меньше, чем указано в аргументе **count**, то оставшиеся символы (байты) заполняются нулями
- Если длина копируемой строки больше, чем указано в аргументе **count**, то в строку **dest** не добавляется символ окончания `'\0'`

Копирование фрагмента строки

```
1 char source[] = "Просто длинная строка";
2 const size_t LEN = strlen("Просто длин");
3
4 char dest[LEN];
5 strncpy(dest, source, LEN);
6
7 printf("Фрагмент строки: %s", dest);
```

Добавление одной строки в другую (склеивание строк)

```
char* strcat(char *dest, const char *source)
```

- Функция добавляет строку **source** в конец строки **dest**
- Функция возвращает указатель на строку **dest**. Как правило, возвращаемое значение игнорируется
- Завершающий *символ окончания строки* переносится в конец объединённой строки
- Следить за размером строк - самостоятельно

Добавление одной строки в другую

```
1 char message[140];  
2  
3 strcat(message, "Эти строки ");  
4 strcat(message, "были объединены ");  
5 strcat(message, "с помощью четырёх ");  
6 strcat(message, "вызовов функции strcat!");  
7  
8 puts(message);
```

Добавление начального фрагмента одной строки в другую

```
char* strncat(char      *dest,  
               const char *source,  
               size_t    count)
```

- Функция добавляет **count** символов из строки **source** в конец строки **dest**
- Функция возвращает указатель на строку **dest**. Как правило, возвращаемое значение игнорируется
- Завершающий *символ окончания строки* переносится в конец объединённой строки
- Если длина строки **source** меньше, чем аргумент **count**, то добавляются только символы до `'\0'`

Добавление начального фрагмента одной строки в другую

```
1 char part1[] = "Здесь должно быть ",  
2   part2[] = "сообщение было не доставлено←  
   ";  
3  
4 char str[50];  
5  
6 strcat(str, part1);  
7 strncat(str, part2, strlen("сообщение"));  
8  
9  
10 printf("Итоговая строка: \"%s\"", str);
```

Поиск первого вхождения символа в строке

```
char* strchr(const char *str, int character);
```

- Функция ищет символ **character** в строке **str**
- Функция возвращает указатель на **первый** найденный символ
- Если совпадения не найдено, возвращается **NULL**
- Несмотря на то, что второй аргумент передаётся как значение типа **int**, фактически внутри функции оно приводится к типу **char**
- Из предыдущего пункта следует *правило хорошего тона*: **strchr** должна использоваться **только** для символов из кодировки ASCII



# Работа с текстом. Стандартная библиотека

Поиск первого вхождения символа в строке

```
1 char text[] = "Скажи-ка, дядя, ведь не даром\n"
2               "Москва, спалённая пожаром,\n"
3               "Французу отдана?\n"
4               "Ведь были ж схватки боевые,\n"
5               "Да, говорят, ещё какие!\n"
6               "Недаром помнит вся Россия\n"
7               "Про день Бородина!\n";
8 printf("Ищем все запятые в тексте\n%s\n", text);
9 puts("-----");
10
11 char *p_space = strchr(text, ',');
12 while ( p_space != NULL) {
13     int place = p_space - text + 1;
14     printf("'", ' найдена на %d месте\n", place);
15     // Поиск продолжается с первого символа,
16     // идущего после ', '
17     p_space = strchr(p_space + 1, ',');
18 }
```

Поиск подстроки (фрагмента)

```
char* strstr(const char *where,  
             const char *what);
```

- **where** - строка, внутри которой ищется фрагмент **what**
- Если совпадение найдено, то есть фрагмент **what** присутствует в **where**, возвращается указатель на его первый символ. Указатель содержит адрес элемента из **исходной строки**
- Если совпадения не найдено (фрагмент **what** не присутствует в строке **where**), возвращается **NULL**

Поиск подстроки: замена слова "было" на некоторое количество символов решётки (#)

```
1 char text[] = "Что делать? Что знать? Что ←  
    получится?";  
2 const size_t SZ = strlen("Что");  
3 char sharps[SZ + 1];  
4 for (size_t i = 0; i < SZ; ++i) {  
5     sharps[i] = '#';  
6 }  
7 sharps[SZ] = '\\0';  
8  
9 printf("Исходный текст: \\n%s\\n\\n", text);  
10 char *p_str = strstr(text, "Что");  
11 while (p_str != NULL) {  
12     strncpy(p_str, sharps, SZ);  
13     p_str = strstr(p_str + SZ + 1, "Что");  
14 }  
15 printf("Итоговый текст: \\n%s\\n\\n", text);
```