

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegEx](#)
- [Ассемблер](#)
- [Купить .PDF](#)


## Урок №84. Символьные константы строк C-style

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 16 Авг 2020 |

 21355

[↑](#)  7

Из материалов [урока №79](#) мы уже знаем, как создать и инициализировать строку C-style:

```
1 #include <iostream>
2
3 int main()
4 {
5     char myName[] = "John";
6     std::cout << myName;
7
8     return 0;
9 }
```

Язык C++ поддерживает еще один способ создания символьных констант строк C-style — через [указатели](#):

```
1 #include <iostream>
2
3 int main()
4 {
5     const char *myName = "John";
6     std::cout << myName;
7
8     return 0;
9 }
```

Хотя обе эти программы работают и выдают одинаковые результаты, выделение памяти в них выполняется по-разному.

В первом случае в программе выделяется память для **фиксированного массива** длиной 5 и инициализируется эта память строкой `John\0`. Поскольку память была специально выделена для массива, то мы можем изменять её содержимое. Сам массив рассматривается как обычная локальная переменная, поэтому, когда он выходит из **области видимости**, память, используемая им, освобождается для других объектов.

Что происходит в случае с **символьной константой**? Компилятор помещает строку `John\0` в память типа `read-only` (только чтение), а затем создает указатель, который указывает на эту строку. Несколько строковых литералов с одним и тем же содержимым могут указывать на один и тот же адрес. Поскольку эта память доступна только для чтения, а также потому, что внесение изменений в строковый литерал может повлиять на дальнейшее его использование, лучше всего перестраховаться, объявив строку константой (типа `const`). Также, поскольку строки, объявленные таким образом, существуют на протяжении всей жизни программы (они имеют статическую продолжительность, а не автоматическую, как большинство других локально определенных литералов), нам не нужно беспокоиться о проблемах, связанных с областью видимости. Поэтому следующее в порядке вещей:

```
1 const char* getName()  
2 {  
3     return "John";  
4 }
```

В фрагменте, приведенном выше, функция `getName()` возвращает указатель на строку C-style `John`. Всё хорошо, так как `John` не выходит из области видимости, когда `getName()` завершает свое выполнение, поэтому вызывающий объект всё равно имеет доступ к строке.

## **`std::cout` и указатели типа `char`**

На этом этапе вы, возможно, уже успели заметить то, как `std::cout` обрабатывает указатели разных типов. Рассмотрим следующий пример:

```
1 #include <iostream>  
2  
3 int main()  
4 {  
5     int nArray[5] = { 9, 7, 5, 3, 1 };  
6     char cArray[] = "Hello!";  
7     const char *name = "John";  
8  
9     std::cout << nArray << '\n'; // nArray распадается в указатель типа int  
10    std::cout << cArray << '\n'; // cArray распадается в указатель типа char  
11    std::cout << name << '\n'; // name уже и так является указателем типа char  
12  
13    return 0;  
14 }
```

Результат выполнения программы на моем компьютере:

0046FAE8

Hello!

John

Почему в массиве типа `int` выводится адрес, а в массивах [типа `char`](#) — строки?

Дело в том, что при передаче указателя не типа `char`, в результате выводится просто содержимое этого указателя (адрес памяти). Однако, если вы передадите объект типа `char*` или `const char*`, то `std::cout` предположит, что вы намереваетесь вывести строку. Следовательно, вместо вывода значения указателя — выведется строка, на которую тот указывает!

Хотя это всё замечательно в 99% случаев, но это может привести и к неожиданным результатам, например:

```
1 #include <iostream>
2
3 int main()
4 {
5     char a = 'R';
6     std::cout << &a;
7
8     return 0;
9 }
```

Здесь мы намереваемся вывести адрес переменной `a`. Тем не менее, `&a` имеет тип `char*`, поэтому `std::cout` выведет это как строку!

Результат выполнения программы на моем компьютере:

R|||4; ;■A

Почему так? `std::cout` предположил, что `&a` (типа `char*`) является строкой. Поэтому сначала вывелось `R`, а затем вывод продолжился. Следующим в памяти был мусор. В конце концов, `std::cout` столкнулся с ячейкой памяти, имеющей значение `0`, которое он интерпретировал как нуль-терминатор, и, соответственно, прекратил вывод. То, что вы видите в результате, может отличаться, в зависимости от того, что находится в памяти после переменной `a`.

Подобное вряд ли случится с вами на практике (так как вы вряд ли захотите выводить адреса памяти), но это хорошая демонстрация того, как всё работает «под капотом» и как программы могут *случайно* «сойти с рельсов».

Оценить статью:

★★★★★ (242 оценок, среднее: 4,91 из 5)



[← Урок №83. Адресная арифметика и индексация массивов](#)[Урок №85. Динамическое выделение памяти](#)

## Комментариев: 7



1. *HillBilly:*

[1 марта 2020 в 17:13](#)

Вопрос №1: Если символьный указатель объявлен в параметрах функции или в теле он тоже имеет статическую продолжительность и существуют на протяжении всей жизни программы или удаляется после блоков {}?

Пример:

```
1 const char* pepe(const char* name = "54321")
2 {
3     const char* temp = "12345";
4     return temp;
5 };
```

Вопрос №2: Произдет ли утечка памяти при вызове функции рере? Или все 3 переменные: name, temp и её возвращаемая копия будут всегда указывать на одни и теже адреса в памяти???

[Ответить](#)



2. *Kris:*

[9 января 2020 в 22:53](#)

Очень странно по поводу литералов. В смысле, что если мы берем литерал "John", эта штука сэйвится в область глобальных переменных типа read-only на все время работы программы. Значит ли это, что если я пишу так:

```
1 const char* p = "John";
2 p = "wtf?";
```

то "John" будет храниться в памяти до конца, т.е., фактически, произойдет утечка памяти? Это звучит нелепо, как минимум. Логичнее было бы, если бы "John" был бы временной областью памяти, которая сама автоматически очищалась бы, как это делают временные результаты выполнения программы. Т.е., лежало бы на стеке. Но эта штука, скорее всего, реально не лежит на стеке, ибо если изменить ее, то произойдет выброс исключения, поэтому она хранится где-то в специальной области памяти. Но кто же очищает за нас память, когда мы меняем указатель? Кто — то же должен очистить память, иначе было бы невозможно без утечки памяти изменить указатель.

[Ответить](#)



1. *Наталья:*

[3 октября 2020 в 21:19](#)

Строковые литералы глобальны. Но глобальная область памяти — это не динамическая память, она очищается по завершении работы программы.

[Ответить](#)



3. *alexk:*

[15 января 2019 в 19:02](#)

Особенно интересно вот так:

```

1  #include <iostream>
2  using std::cout;
3
4  int main() {
5      char a{'a'}, b{'b'}, c{'c'}, d{'d'}, e{'e'}, f{'f'};
6
7      char* p = &a;
8      cout << a << b << c << d << e << f << '\n';
9      cout << "p = " << p << '\n';
10
11     cout << '\n';
12     f = '\0';
13     cout << a << b << c << d << e << f << '\n';
14     cout << "p = " << p << '\n';
15
16     return 0;
17 }
```

и результат:

>>>>> Console - START >>>>>

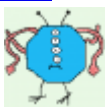
abcdef

p = abcdef: ???

abcde

p = abcde

[Ответить](#)



1. *kmish:*

[7 февраля 2019 в 17:53](#)

Мой вывод твоей программы:

```
abcdef
```

```
p = a||||||CF^6o.#
```

```
abcde
```

```
p = a||||||CF^6o.#
```

[Ответить](#)


1. *alexk:*

[9 февраля 2019 в 17:36](#)

на MSVS-2010 + Win7 такая прога и у меня кажет так же как и у ВАС ... ?! ... 😊 ...

а вот на xubuntu 16.04 при компиляции строкой:  
g++ -Wall -std=c++1y ...

вариант xubuntu, лично мне, БОЛЬШЕ нравится ! ... 😊 ...

[Ответить](#)


2. *Наталья:*

[3 октября 2020 в 21:21](#)

Вы создаёте не массив! И нет гарантий, что эти переменные расположены последовательно. Результаты могут отличаться на разных системах.

[Ответить](#)

## Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены \*

Имя \*

Email \*

Комментарий






☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию

☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.

[TELEGRAM](#)  [КАНАЛ](#)  
[ПАБЛИК](#) 



## ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020