

Ravesli [Ravesli](#)


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


Урок №36. Литералы и магические числа

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновлено 2 Сен 2020 |

 48641

[↑](#)  21

В языке C++ есть два вида констант: литеральные и символьные. На этом уроке мы рассмотрим литеральные константы.

Оглавление:

1. [Литеральные константы](#)
2. [Литералы в восьмеричной и шестнадцатеричной системах счисления](#)
3. [Бинарные литералы и разделитель цифр в C++14](#)
4. [Магические числа. Что с ними не так?](#)

Литеральные константы

Литеральные константы (или просто «*литералы*») — это значения, которые вставляются непосредственно в код. Поскольку они являются константами, то их значения изменить нельзя. Например:

```
1 bool myNameIsAlex = true; // true - это литеральная константа типа bool
2 int x = 5; // 5 - это литеральная константа типа int
3 int y = 2 * 3; // 2 и 3 - это литеральные константы типа int
```

С литералами типов **bool** и **int** всё понятно, а вот для литералов **типа с плавающей точкой** есть два способа объявления:

```
1 double pi = 3.14159; // 3.14159 - это литерал типа double
2 double avogadro = 6.02e23; // число avogadro - 6.02 x 10^23
```

Во втором способе объявления, число после экспонента может быть и отрицательным:

```
1 | double electron = 1.6e-19; // заряд электрона - 1.6 x 10^-19
```

Числовые литералы могут иметь суффиксы, которые определяют их типы. Эти суффиксы не являются обязательными, так как компилятор понимает из контекста, константу какого типа данных вы хотите использовать.

Тип данных	Суффикс	Значение
int	u или U	unsigned int
int	l или L	long
int	ul, uL, Ul, UL, lu, lU, Lu или LU	unsigned long
int	ll или LL	long long
int	ull, uLL, Ull, ULL, llu, llU, LLu или LLU	unsigned long long
double	f или F	float
double	l или L	long double

Суффиксы есть даже для целочисленных типов (но они почти не используются):

```
1 | unsigned int nValue = 5u; // тип int unsigned
2 | long nValue2 = 5L; // тип long
```

По умолчанию литеральные константы типа с плавающей точкой являются типа double. Для конвертации литеральных констант в тип float можно использовать суффикс f или F:

```
1 | float fValue = 5.0f; // тип float
2 | double d = 6.02e23; // тип double (по умолчанию)
```

Язык C++ также поддерживает литералы типов string и **char**:

```
1 | char c = 'A'; // 'A' - это литерал типа char
2 | std::cout << "Hello, world!"; // "Hello, world!" - это литерал строки C-style
3 | std::cout << "Hello," " world!"; // C++ связывает последовательные литералы типа str
```

Литералы хорошо использовать в коде до тех пор, пока их значения понятны и однозначны. Это выполнение операций присваивания, математических операций или вывода текста в консоль.

Литералы в восьмеричной и шестнадцатеричной системах счисления

В повседневной жизни мы используем **десятичную систему счисления**, которая состоит из десяти цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. По умолчанию язык C++ использует десятичную систему счисления для чисел в программах:

```
1 | int x = 12; // предполагается, что 12 является числом десятичной системы счисления
```

В **двоичной (бинарной) системе счисления** всего 2 цифры: 0 и 1. Значения: 0, 1, 10, 11, 100, 101, 110, 111 и т.д.

Есть еще две другие системы счисления: восьмеричная и шестнадцатеричная.

Восьмеричная система счисления состоит из 8 цифр: 0, 1, 2, 3, 4, 5, 6 и 7. Значения: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12 и т.д.

Примечание: В восьмеричной системе счисления нет цифр 8 и 9, так что сразу перескакиваем от 7 к 10.

Десятичная система счисления 0 1 2 3 4 5 6 7 8 9 10 11

Восьмеричная система счисления 0 1 2 3 4 5 6 7 10 11 12 13

Для использования литерала из восьмеричной системы счисления, используйте префикс 0 (ноль):

```
1 #include <iostream>
2
3 int main()
4 {
5     int x = 012; // 0 перед значением означает, что это восьмеричный литерал
6     std::cout << x;
7     return 0;
8 }
```

Результат выполнения программы:

10

Почему 10 вместо 12? Потому что [std::cout](#) выводит числа в десятичной системе счисления, а 12 в восьмеричной системе = 10 в десятичной.

Восьмеричная система счисления используется крайне редко.

Шестнадцатеричная система счисления состоит из 16 символов: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Десятичная система 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Шестнадцатеричная система 0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11

Для использования литерала из шестнадцатеричной системы счисления, используйте префикс 0x:

```
1 #include <iostream>
2
3 int main()
4 {
5     int x = 0xF; // 0x перед значением означает, что это шестнадцатеричный литерал
6     std::cout << x;
7     return 0;
8 }
```

Результат выполнения программы:

15

Поскольку в этой системе 16 символов, то одна шестнадцатеричная цифра занимает 4 бита. Следовательно, две шестнадцатеричные цифры занимают 1 байт.

Рассмотрим 32-битное целое число из двоичной системы счисления: 0011 1010 0111 1111 1001 1000 0010 0110. Из-за длины и повторения цифр его сложно прочесть. В шестнадцатеричной системе счисления это же значение будет выглядеть следующим образом: 3A7F 9826. Такой удобный/сжатый формат является преимуществом шестнадцатеричной системы счисления, поэтому шестнадцатеричные значения часто используются для представления адресов памяти или необработанных значений в памяти.

До C++14 использовать литерал из двоичной системы счисления было невозможно. Тем не менее, шестнадцатеричная система счисления может нам в этом помочь:

```
1 #include <iostream>
2
3 int main()
4 {
5     int bin(0);
6     bin = 0x01; // присваиваем переменной бинарный литерал 0000 0001
7     bin = 0x02; // присваиваем переменной бинарный литерал 0000 0010
8     bin = 0x04; // присваиваем переменной бинарный литерал 0000 0100
9     bin = 0x08; // присваиваем переменной бинарный литерал 0000 1000
10    bin = 0x10; // присваиваем переменной бинарный литерал 0001 0000
11    bin = 0x20; // присваиваем переменной бинарный литерал 0010 0000
12    bin = 0x40; // присваиваем переменной бинарный литерал 0100 0000
13    bin = 0x80; // присваиваем переменной бинарный литерал 1000 0000
14    bin = 0xFF; // присваиваем переменной бинарный литерал 1111 1111
15    bin = 0xB3; // присваиваем переменной бинарный литерал 1011 0011
16    bin = 0xF770; // присваиваем переменной бинарный литерал 1111 0111 0111 0000
17
18    return 0;
19 }
```

Бинарные литералы и разделитель цифр в C++14

В C++14 мы можем использовать бинарные (двоичные) литералы, добавляя префикс 0b:

```
1 #include <iostream>
2
3 int main()
4 {
5     int bin(0);
6     bin = 0b1; // присваиваем переменной бинарный литерал 0000 0001
7     bin = 0b11; // присваиваем переменной бинарный литерал 0000 0011
8     bin = 0b1010; // присваиваем переменной бинарный литерал 0000 1010
9     bin = 0b11110000; // присваиваем переменной бинарный литерал 1111 0000
10
11    return 0;
12 }
```

Поскольку длинные литералы читать трудно, то в C++14 добавили возможность использовать одинарную кавычку ' в качестве разделителя цифр:

```
1 #include <iostream>
```

```
2
3 int main()
4 {
5     int bin = 0b1011'0010; // присваиваем переменной бинарный литерал 1011 0010
6     long value = 2'532'673'462; // намного проще читать, нежели 2532673462
7
8     return 0;
9 }
```

Если ваш компилятор не поддерживает C++14, то использовать бинарные литералы и разделитель цифр вы не сможете — компилятор выдаст ошибку.

Магические числа. Что с ними не так?

Рассмотрим следующий фрагмент кода:

```
1 int maxStudents = numClassrooms * 30;
```

В вышеприведенном примере число 30 является магическим числом. **Магическое число** — это хорошо закодированный литерал (обычно, число) в строке кода, который не имеет никакого контекста. Что это за 30, что оно означает/обозначает? Хотя из вышеприведенного примера можно догадаться, что число 30 обозначает максимальное количество учеников, находящихся в одном кабинете — в большинстве случаев, это не будет столь очевидным и понятным. В более сложных программах контекст подобных чисел разгадать намного сложнее (если только не будет соответствующих [комментариев](#)).

Использование магических чисел является плохой практикой, так как в дополнение к тому, что они не предоставляют никакого контекста (для чего и почему используются), они также могут создавать проблемы, если их значения необходимо будет изменить. Предположим, что школа закупила новые парты. Эта покупка, соответственно, увеличила максимально возможное количество учеников, находящихся в одном кабинете, с 30 до 36 — это нужно будет продумать и отобразить в нашей программе.

Рассмотрим следующий фрагмент кода:

```
1 int maxStudents = numClassrooms * 30;
2 setMax(30);
```

Чтобы обновить число учеников в кабинете, нам нужно изменить значение константы с 30 на 36. Но что делать с вызовом функции `setMax(30)`? [Аргумент](#) 30 и константа 30 в коде, приведенном выше, являются одним и тем же, верно? Если да, то нам нужно будет обновить это значение. Если нет, то нам не следует вообще трогать этот вызов функции. Если же проводить автоматический глобальный поиск и замену числа 30, то можно ненароком изменить и аргумент функции `setMax()`, в то время, когда его вообще не следовало бы трогать. Поэтому вам придется просмотреть весь код «вручную», в поисках числа 30, а затем в каждом конкретном случае определить — изменить ли 30 на 36 или нет. Это может занять очень много времени, кроме того, вероятность возникновения новых ошибок повышается в разы.

К счастью, есть лучший вариант — использовать символьные константы. О них мы поговорим на следующем уроке.

Правило: Старайтесь свести к минимуму использование магических чисел в ваших программах.

Оценить статью:

★★★★★ (306 оценок, среднее: 4,88 из 5)



← [Урок №35. Символьный тип данных char](#)



[Урок №37. const, constexpr и символьные константы](#) →

Комментариев: 21



1. *Анатолий:*
[7 мая 2020 в 04:06](#)

Ну 30 конечно никто не пишет. а пишут или именованную константу или макрос. Насчёт литералов. Да удобно. Но можно использовать и свои литералы. Последние версии C++ это позволяют. Хотя код становится малопонятным если вы туда понапишете кучу своих литералов... В C кстати бинарный литерал тоже есть. Это не прерогатива C++. Насчёт групп ну мне не нравится. Я не люблю группированные числа... Особенно в виде строк, а не констант. С ними столько возни... Но это моё личное мнение.

[Ответить](#)



2. *Анатолий:*
[7 мая 2020 в 04:00](#)

Я всегда сравниваю C/C++ с фортраном. Как там? Там вообще задаётся размер типа... А функции универсальны... Если ваш комп не поддерживает этот размер вылетит ошибка... Я не пропагандирую фортран. Но концепция типов в C/C++ мягко говоря ущербна. Ибо там изначально заложили в названия типов их размер. В фортране такое тоже было. Но сейчас нет. И правильно. Я молу легко работать с любыми типами и они все работают. И там не надо выдумывать дурацкие названия. Просто пишешь типа `real(16)` и всё у вас тип `_float128`. Или `integer(16)` и вуаля у вас тип `__int128`. Никаких проблем. Да, фортран это вам не Си... Хотите его преобразовать в строку? Да пожалуйста! Там нет многочисленных функций преобразований как в Си. Всего 2 `write` и `read`, но они делают всё и гораздо лучше где либо. И в любой системе счисления. Даже двоичной. Конечно есть библиотека `gmp`. Там можно использовать числа любого размера. главное чтобы в память поместилось... Но то библиотека. А это аппаратная поддержка! А функция скажем `sin` там всего одна, но на все случаи жизни. Да фортран не поддерживает беззнаковые числа. Ну и ладно... Но слышал, что компилятор `ifort` их поддерживает. но он платный... Поэтому если на C/C++ вам понадобятся эти типы данных, то ни одна библиотека не будет с ними работать. Но сами типы там есть... вам придётся все эти функции писать самим. Особенно если вам понадобится двоичный формат... Или подключите модуль на фортране, там это уже есть. В Qt кстати поддержка функций есть... Но нет

преобразований в строки. Эта среда более продвинута, чем C/C++. Хотя это только если вы используете это Qt.

[Ответить](#)



3. *Хей Лонг:*

[12 апреля 2020 в 17:53](#)

Немного мудрёные формулировки с магическими числами. Константы предполагаются как неизменяемые. Если есть вероятность, что константу придётся изменить, надо изначально создавать переменную, не придумывая названий для "изменяемой константы". Хотя можно написать десять программ для десяти разных чисел парт в классе, но программистами всё равно станут только адекватные люди.

[Ответить](#)



4. *Анастасия:*

[31 мая 2019 в 22:41](#)

Пояснение для тех, кто, как и я, сначала не понял, как связаны следующие присваивания с комментариями:

```
1 bin = 0x10; // присваиваем переменной бинарный литерал 0001 0000
2   bin = 0x20; // присваиваем переменной бинарный литерал 0010 0000
3   bin = 0x40; // присваиваем переменной бинарный литерал 0100 0000
4   bin = 0x80; // присваиваем переменной бинарный литерал 1000 0000
5   bin = 0xFF; // присваиваем переменной бинарный литерал 1111 1111
6   bin = 0xB3; // присваиваем переменной бинарный литерал 1011 0011
7   bin = 0xF770; // присваиваем переменной бинарный литерал 1111 0111 0111 0000
```

Например, мы хотим понять, почему 0x80 — это 1000 0000.

80 в шестнадцатеричной системе переведём в привычную десятичную, для этого каждая цифра справа налево умножается на степени 16, начиная от 0: $0x80 = 0 \cdot 1 + 8 \cdot 16 = 0 + 128 = 128$. 128 переводим в двоичную систему, 128 — это 2 в 7 степени, значит двоичная запись будет 1000 0000 (тут тоже степени двойки справа налево, начиная с 0). Подробные пояснения по переводу из одной системы в другую есть в 44 уроке.

Скорей всего, всё можно понять и проще, осознав чудо превращения сразу из 16-ричной системы в двоичную. Кто осознал, делитесь.

[Ответить](#)



1. *Анастасия:*

[1 июня 2019 в 18:28](#)

Хорошая демонстрация и объяснение, как быстро переводить числа из двоичной системы в шестнадцатеричную, есть [здесь](#).

[Ответить](#)



2. Анастасия:

[11 июля 2019 в 11:47](#)

Есть гораздо более удобный способ конвертации чисел в двоичную систему и из неё. Достаточно выучить таблицу тетрад (а в ней же и триад(а в ней же диад)) чтобы на ходу переводить числа из четверичной/восьмеричной/шестнадцатеричной системы счисления. Жаль, что пока не знаю как всё быстро и на ходу в десятичную систему переводить, но пока поделюсь тем, что есть.

(exe.exe.exe, попыталась это всё расписать, но здесь нужно наглядное представление)

Пожалуй, лучшее объяснение есть у информатика бу. А вот и ссылка:

<https://www.youtube.com/watch?v=npB8IF-V4mc&list=PLgvtHXe0kJXaNH57H5yolkewq-p5hfpDR>

[Ответить](#)



3. anton:

[6 октября 2019 в 00:42](#)

Не надо переводить в десятичную и обратно. Всё просто, никакой магии:

Для кодирования 0,1 нужен 1 бит.

1	// Для кодирования 0..3 (4 состояния(4 чисел(2^2))) нужно 2 бита
2	// Для кодирования 0..8 (8 состояний(8 чисел(2^3))) нужно 3 бита
3	// Для кодирования 0..16 (16 состояний(16 чисел(2^4))) нужно 4 бита.

То есть 4 бита можно выразить либо 16-ю двоичными комбинациями:

0000

0001

0010

...

1111

Либо 16-ю символами: 0123456789ABCDEF.

Соответственно:

0000 - 0

0001 - 1

0010 - 2

...

1110 - E

1111 - F

Вот и всё.

А чтоб не запоминать таблицу, запоминаешь, что А — это десять., дальше по алфавиту загибаешь пальцы на руке.

В — первый палец (то есть результат 11)

С — второй

D — третий (то есть D — это тринадцать, а тринадцать это $8+4+1$, то есть 0b1101)

Вот так я делаю)))

[Ответить](#)



4. ХейЛонг:

[12 апреля 2020 в 17:59](#)

Мой препод по программированию приводил хороший пример. Программирование как занятия музыкой. Чтобы развиваться, нужно заниматься постоянно, часами и сутками. А вот мой пример. Чтобы начать играть на инструменте, надо выучить ноты. Базовый курс информатики помогает с пониманием систем счисления и прочих важных мелочей.

[Ответить](#)



5. *Артур:*

[10 мая 2019 в 13:58](#)

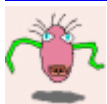
Спасибо, отличный перевод, все очень понятно.

Понятно почему вы не перевели слово statement (заявление? утверждение? высказывание? изложение?), но в этой статье (простите за придирку)

> число avogadro

число Авогадро оставили непереведенным, несмотря на то, что оно вполне себе имеет общепризнанный перевод.

[Ответить](#)



6. *alexk:*

[24 декабря 2018 в 13:54](#)

Чтобы можно было применять цифровые разделители, т.е. записывать литералы в виде:

```
1 | long value = 2'532'673'462
```

необходимо установить ключ компиляции:

```
g++ -Wall -std=c++1y ...
```

Это справедливо для:

```
g++ (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609
```

[Ответить](#)



7. *Константин:*

[28 сентября 2018 в 00:57](#)

И ещё не могу эту последовательность разгадать после 0x: 01; 02; 04; 08; 10; 20; 40; 80; FF; особенно здесь B3; F770;

[Ответить](#)



1. *Константин:*

[18 марта 2019 в 05:14](#)

Уррр-ааа!!! Разгадал! А это значение в этот тип не помещается:

```
1 | long value = 2'532'673'462; // намного проще читать, чем 2532673462
```

Нужен long long

[Ответить](#)



8. Константин:

[28 сентября 2018 в 00:21](#)

Юра, пардон, но я не понял для чего была создана `int bin`, и, тем более, для чего мы в ней столько раз подряд значения меняли?

[Ответить](#)



9. Иван:

[21 марта 2018 в 14:35](#)

До меня дошло. Невнимательно прочитал определение литерла. Спасибо!

[Ответить](#)



1. Юрий:

[21 марта 2018 в 17:08](#)

Пожалуйста 😊

[Ответить](#)



10. Иван:

[21 марта 2018 в 11:24](#)

"Литеральные константы (обычно просто литералы) – это значения, которые вставляются непосредственно в код. Они являются константами, так как изменить их значения нельзя. "

А почему мы не можем изменить их значения? Можно же просто присвоить другое значение этому литералу, судя по примерам в начале урока

[Ответить](#)



1. Юрий:

[21 марта 2018 в 13:33](#)

Судя по какому примеру? Вы можете присвоить числу 4 (четыре) значение 5 (пять), чтобы 4 был равен 5? Или можно сделать, чтобы `true` был `false`, а `false` = `true`? Есть литералы, а есть переменные. Значение переменных можно изменять, значения литералов — нет.

[Ответить](#)



1. Дмитрий:

[4 июня 2019 в 22:55](#)

Вот не пойму я зачем давать второй раз определения литералу, при том таким образом что становится не понятным — говорится о том же литерале что определялся в 17 уроке (так просто и не двусмысленно), или о каком то новом термине. Ничего противоречивого нет конечно, вот только пан автор (я имею ввиду автора оригинала), не явно, бывает путает читателя, куда то пропал его талант объяснять просто и в достаточном объеме, без мишуры как говорится. Жаль, ведь наблюдаю регрессию по качеству учебного ресурса. Надеюсь, что только эта глава столь плохо подана.

[Ответить](#)

1. *Юрий:*

[6 июня 2019 в 14:52](#)

Ну смотрите, если регрессия увеличивается с каждым уроком — то, возможно, стоит посмотреть другие учебные курсы на просторах Интернета.



2. *Дмитрий:*

[7 июня 2019 в 19:21](#)

Я бы так и сделал, если бы первые страницы учебного материала, на вашем сайте, я бы не счел одним из лучших учебных материалов которые я встречал. В других источниках/книгах авторы то графоманией страдают, то недоговаривают там, где хотелось бы. Вот и огорчает сей момент, ведь привыкаешь что инфа пережевана хорошо, а тут неоднозначность. Юрий, автор оригинала расстроил не на долго, но думаю, было бы не лишним поблагодарить вас за ваш труд и результат, думаю ваш сайт многим помог и помогает. Конкретно эта глава, просто, не лучше описана чем в других источниках. В общем — спасибо!

[Ответить](#)

1. *Юрий:*

[8 июня 2019 в 14:43](#)

Ваш месседж понятен, некоторые уроки/темы действительно могут быть объяснены лучше, чем другие: вопрос уже в уровне этого самого "лучше" и в том, с чем сравнивать.

Пожалуйста 😊

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.

Отправить комментарий






TELEGRAM  КАНАЛ

Электронная почта



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 «Same Game»](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

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