2.8 — Literals

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 10TH, 2018

C++ has two kinds of constants: literal, and symbolic. In this lesson, we'll cover literals.

Literal constants

Literal constants (usually just called "literals") are values inserted directly into the code. They are constants because you can't change their values. For example:

```
bool myNameIsAlex = true; // boolean variable 'myNameIsAlex' is assigned is a boolean literal constant
 'true'
int x = 5; // integer variable 'x' is assigned integer literal constant 5
std::cout << 2 * 3; // 2 and 3 are integer literals
```

While boolean and integer literals are pretty straightforward, there are two different ways to declare floating-point literals:

```
double pi = 3.14159; // 3.14159 is a double literal
double avogadro = 6.02e23; // avogadro's number is 6.02 x 10^23
```

In the second form, the number after the exponent can be negative:

```
double electron = 1.6e-19; // charge on an electron is 1.6 x 10^-19
```

Numeric literals can have suffixes that determine their types. These suffixes are optional, as the compiler can usually tell from context what kind of constant you're intending.

Data Type	Suffix	Meaning
int	u or U	unsigned int
int	l or L	long
int	ul, uL, Ul, UL, lu, lU, Lu, or LU	unsigned long
int	ll or LL	long long
int	ull, uLL, Ull, ULL, llu, llU, LLu, or LLU	unsigned long long
double	f or F	float
double	l or L	long double

You probably won't need to use suffixes for integer types, but here are examples:

```
unsigned int nValue = 5u; // unsigned int
long nValue2 = 5L; // long
```

By default, floating point literal constants have a type of *double*. To convert them into a float value, the f or F suffix can be used:

```
float fValue = 5.0f; // float
double d = 6.02e23; // double (by default)
```

C++ also supports char and string literals:

```
char c = 'A'; // 'A' is a char literal
std::cout << "Hello, world!" // "Hello, world!" is a C-style string literal
std::cout << "Hello," " world!" // C++ will concatenate sequential string literals
```

Char literals work just like you'd expect. However, string literals are handled very strangely in C++. For now, it's fine to use string literals to print text with std::cout, but don't try and assign them to variables or pass them to functions -- it either won't work, or won't work like you'd expect. We'll talk more about C-style strings (and how to work around all of those odd issues) in future lessons.

Literals are fine to use in C++ code so long as their meanings are clear. This is most often the case when used to assign a value to a variable, do math, or print some text to the screen.

Octal and hexadecimal literals

In everyday life, we count using **decimal** numbers, where each numerical digit can be 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Decimal is also called “base 10”, because there are 10 possible digits (0 through 9). In this system, we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, … By default, numbers in C++ programs are assumed to be decimal.

```
int x = 12; // 12 is assumed to be a decimal number
```

In **binary**, there are only 2 digits: 0 and 1, so it is called “base 2”. In binary, we count like this: 0, 1, 10, 11, 100, 101, 110, 111, …

There are two other “bases” that are sometimes used in computing: octal, and hexadecimal.

Octal is base 8 -- that is, the only digits available are: 0, 1, 2, 3, 4, 5, 6, and 7. In Octal, we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, … (note: no 8 and 9, so we skip from 7 to 10).

Decimal	0	1	2	3	4	5	6	7	8	9	10	11
Octal	0	1	2	3	4	5	6	7	10	11	12	13

To use an octal literal, prefix your literal with a 0:

```
#include <iostream>

int main()
{
    int x = 012; // 0 before the number means this is octal
    std::cout << x;
    return 0;
}
```

This program prints:

```
10
```

Why 10 instead of 12? Because numbers are printed in decimal, and 12 octal = 10 decimal.

Octal is hardly ever used, and we recommend you avoid it.

Hexadecimal is base 16. In hexadecimal, we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, …

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

To use a hexadecimal literal, prefix your literal with 0x.

```
#include <iostream>

int main()
{
    int x = 0xF; // 0x before the number means this is hexadecimal
    std::cout << x;
    return 0;
}
```

This program prints:

```
15
```

Because there are 16 different values for a hexadecimal digit, we can say that a single hexadecimal digit encompasses 4 bits. Consequently, a pair of hexadecimal digits can be used to exactly represent a full byte.

Consider a 32-bit integer with value 0011 1010 0111 1111 1001 1000 0010 0110. Because of the length and repetition of digits, that’s not easy to read. In hexadecimal, this same value would be: 3A7F 9826. This makes hexadecimal values useful as a concise way to

represent a value in memory. For this reason, hexadecimal values are often used to represent memory addresses or raw values in memory.

Prior to C++14, there is no way to assign a binary literal. However, hexadecimal pairs provides us with an useful workaround:

```
#include <iostream>

int main()
{
    int bin(0);
    bin = 0x01; // assign binary 0000 0001 to the variable
    bin = 0x02; // assign binary 0000 0010 to the variable
    bin = 0x04; // assign binary 0000 0100 to the variable
    bin = 0x08; // assign binary 0000 1000 to the variable
    bin = 0x10; // assign binary 0001 0000 to the variable
    bin = 0x20; // assign binary 0010 0000 to the variable
    bin = 0x40; // assign binary 0100 0000 to the variable
    bin = 0x80; // assign binary 1000 0000 to the variable
    bin = 0xFF; // assign binary 1111 1111 to the variable
    bin = 0xB3; // assign binary 1011 0011 to the variable
    bin = 0xF770; // assign binary 1111 0111 0111 0000 to the variable

    return 0;
}
```

C++14 binary literals and digit separators

In C++14, we can assign binary literals by using the 0b prefix:

```
#include <iostream>

int main()
{
    int bin(0);
    bin = 0b1;  // assign binary 0000 0001 to the variable
    bin = 0b11; // assign binary 0000 0011 to the variable
    bin = 0b1010; // assign binary 0000 1010 to the variable
    bin = 0b11110000; // assign binary 1111 0000 to the variable

    return 0;
}
```

Because long literals can be hard to read, C++14 also adds the ability to use a quotation mark (‘) as a digit separator.

```
#include <iostream>

int main()
{
    int bin = 0b1011'0010;  // assign binary 1011 0010 to the variable
    long value = 2'132'673'462; // much easier to read than 2132673462

    return 0;
}
```

If your compiler isn’t C++14 compatible, your compiler will complain if you try to use either of these.

Magic numbers, and why they are bad

Consider the following snippet:

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

A number such as the 30 in the snippet above is called a magic number. A **magic number** is a hard-coded literal (usually a number) in the middle of the code that does not have any context. What does 30 mean? Although you can probably guess that in this case it’s the maximum number of students per class, it’s not absolutely clear. In more complex programs, it can be very difficult to infer what a hard-coded number represents, unless there’s a comment to explain it.

Using magic numbers is generally considered bad practice because, in addition to not providing context as to what they are being used for, they pose problems if the value needs to change. Let's assume that the school buys new desks that allow them to raise the class size from 30 to 35, and our program needs to reflect that. Consider the following program:

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

To update our program to use the new classroom size, we'd have to update the constant 30 to 35. But what about the call to setMax()? Does that 30 have the same meaning as the other 30? If so, it should be updated. If not, it should be left alone, or we might break our program somewhere else. If you do a global search-and-replace, you might inadvertently update the argument of setMax() when it wasn't supposed to change. So you have to look through all the code for every instance of the literal 30, and then determine whether it needs to change or not. That can be seriously time consuming (and error prone).

Fortunately, better options (symbolic constants) exist. We'll talk about those in the next lesson.

Rule: Don't use magic numbers in your code.

 **2.9 -- Const, constexpr, and symbolic constants**

 **Index**

 **2.7 -- Chars**

87 comments to 2.8 — Literals

« Older Comments 1 2

Aaron
July 6, 2018 at 2:40 pm · Reply

I really don't understand why suffixes exist if you're declaring the variable to be a certain type. What's the difference? And why are both necessary?

nascardriver
July 8, 2018 at 4:58 am · Reply

Hi Aaron!

```
1 double db{ 10 / 3 };
```

@db is 3, because 10 and 3 are integers. You need a suffix to tell the compiler which kind of number to use.

Johnny
May 9, 2018 at 10:35 pm · Reply