# 4.13 — Literals

ALEX AUGUST 5, 2021

In programming, a constant is a fixed value that may not be changed. C++ has two kinds of constants: literal constants, and symbolic constants. We'll cover literal constants in this lesson, and symbolic constants in the next lesson.

Literal constants (usually just called literals) are values inserted directly into the code. For example:

```
return 5; // 5 is an integer literal
bool myNameIsAlex { true }; // true is a boolean literal
std::cout << 3.4; // 3.4 is a double literal
```

They are constants because their values can not be changed dynamically (you have to change them, and then recompile for the change to take effect).

Just like objects have a type, all literals have a type. The type of a literal is assumed from the value and format of the literal itself.

By default:

| Literal value | Examples | Default type |
|---|---|---|
| integral value | 5, 0, -3 | int |
| boolean value | true, false | bool |
| floating point value | 3.4, -2.2 | double (not float)! |
| char value | 'a' | char |
| C-style string | "Hello, world!" | const char[14] |

Literal suffixes

If the default type of a literal is not as desired, you can change the type of a literal by adding a suffix:

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

```
std::cout << 5; // 5 (no suffix) is type int (by default)
std::cout << 5u; // 5u is type unsigned int
std::cout << 5L; // 5L is type long
```

By default, floating point literal constants have a type of *double*. To make them float literals instead, the f (or F) suffix should be used:

```
std::cout << 5.0; // 5.0 (no suffix) is type double (by default)
std::cout << 5.0f; // 5.0f is type float
```

New programmers are often confused about why the following doesn't work as expected:

```
float f { 4.1 }; // warning: 4.1 is a double literal, not a float literal
```

Because 4.1 has no suffix, it's treated as a double literal, not a float literal. When C++ defines the type of a literal, it does not care what you're doing with the literal (e.g. in this case, using it to initialize a float variable). Therefore, the 4.1 must be converted from a double to a float before it can be assigned to variable f, and this could result in a loss of precision.

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

---

## String literals

In lesson 4.11 -- Chars, we defined a string as a collection of sequential characters. C++ supports string literals:

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

String literals are handled very strangely in C++ for historical reasons. For now, it's fine to use string literals to print text with std::cout, but don't try to 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.

---

## Scientific notation for floating point literals

There are two different ways to declare floating-point literals:

```
double pi { 3.14159 }; // 3.14159 is a double literal in standard notation
double avogadro { 6.02e23 }; // 6.02 x 10^23 is a double literal in scientific notation
```

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
```

## 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.

```cpp
#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 provide us with a useful workaround:

```cpp
#include <iostream>

int main()
{
    int bin{};    // assume 32-bit ints
    bin = 0x0001; // assign binary 0000 0000 0000 0001 to the variable
    bin = 0x0002; // assign binary 0000 0000 0000 0010 to the variable
    bin = 0x0004; // assign binary 0000 0000 0000 0100 to the variable
    bin = 0x0008; // assign binary 0000 0000 0000 1000 to the variable
    bin = 0x0010; // assign binary 0000 0000 0001 0000 to the variable
    bin = 0x0020; // assign binary 0000 0000 0010 0000 to the variable
    bin = 0x0040; // assign binary 0000 0000 0100 0000 to the variable
    bin = 0x0080; // assign binary 0000 0000 1000 0000 to the variable
    bin = 0x00FF; // assign binary 0000 0000 1111 1111 to the variable
    bin = 0x00B3; // assign binary 0000 0000 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{};        // assume 32-bit ints
    bin = 0b1;        // assign binary 0000 0000 0000 0001 to the variable
    bin = 0b11;       // assign binary 0000 0000 0000 0011 to the variable
    bin = 0b1010;     // assign binary 0000 0000 0000 1010 to the variable
    bin = 0b11110000; // assign binary 0000 0000 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.

---

## Printing decimal, octal, hexadecimal, and binary numbers

By default, C++ prints values in decimal. However, you can tell it to print in other formats. Printing in decimal, octal, or hex is easy via use of std::dec, std::oct, and std::hex:

```
#include <iostream>

int main()
{
    int x { 12 };
    std::cout << x << '\n'; // decimal (by default)
    std::cout << std::hex << x << '\n'; // hexadecimal
    std::cout << x << '\n'; // now hexadecimal
    std::cout << std::oct << x << '\n'; // octal
    std::cout << std::dec << x << '\n'; // return to decimal
    std::cout << x << '\n'; // decimal

    return 0;
}
```

This prints:

```
12
c
c
14
12
12
```

Printing in binary is a little harder, as std::cout doesn't come with this capability built-in. Fortunately, the C++ standard library includes a type called std::bitset that will do this for us (in the <bitset> header). To use std::bitset, we can define a std::bitset variable and tell std::bitset how many bits we want to store. The number of bits must be a compile time constant. std::bitset can be initialized with an unsigned integral value (in any format, including decimal, octal, hex, or binary).

```
#include <bitset> // for std::bitset
#include <iostream>

int main()
{
 // std::bitset<8> means we want to store 8 bits
 std::bitset<8> bin1{ 0b1100'0101 }; // binary literal for binary 1100 0101
 std::bitset<8> bin2{ 0xC5 }; // hexadecimal literal for binary 1100 0101

 std::cout << bin1 << ' ' << bin2 << '\n';
 std::cout << std::bitset<4>{ 0b1010 } << '\n'; // we can also print from std::bitset directly

 return 0;
}
```

This prints:

```
11000101 11000101
1010
```

We can also create a temporary (anonymous) std::bitset to print a single value. In the above code, this line:

```
std::cout << std::bitset<4>{ 0b1010 } << '\n'; // we can also print from std::bitset directly
```

creates a temporary std::bitset object with 4 bits, initializes it with *0b1010*, prints the value in binary, and then discards the temporary std::bitset.

---

## 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 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:

```
int maxStudents{ numClassrooms * 30 };
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).

Although we say magic "numbers", this affects all kinds of values. Consider this example

```
std::cout << "Enter a number less than 100: ";

int input{};
std::cin >> input;

if (input >= 100)
{
  std::cout << "Invalid input! The number has to be less than 100.";
}
```

There's only one number (100) in this example, but it's also used in the strings. If we decide to update the maximum to let's say 200, we have to update three different occasions of 100.

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

### Best practice

Don't use magic numbers in your code.

Next lesson
4.14 Const, constexpr, and symbolic constants

Back to table of contents

Previous lesson
4.12 An introduction to std::string