4.15 — Literals

ALEX OCTOBER 3, 2022

Literals are unnamed 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
```

Literals are sometimes called **literal constants** because their values cannot be reassigned.

The type of a literal

Just like objects have a type, all literals have a type. The type of a literal is deduced from the literal's value. For example, a literal that is a whole number (e.g. `5`) is deduced to be of type `int`.

By default:

Literal value	Examples	Default literal type
integer value	5, 0, -3	int
boolean value	true, false	bool
floating point value	1.2, 0.0, 3.4	double (not float!)
character	'a', '\n'	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
integral	u or U	unsigned int
integral	l or L	long
integral	ul, uL, Ul, UL, lu, lU, Lu, or LU	unsigned long
integral	ll or LL	long long
integral	ull, uLL, Ull, ULL, llu, llU, LLu, or LLU	unsigned long long
integral	z or Z	The signed version of std::size_t (C++23)
integral	uz or UZ	std::size_t (C++23)
floating point	f or F	float
floating point	l or L	long double
string	s	std::string
string	sv	std::string_view

Suffixes are not case sensitive. Because lower-case `L` can look like numeric `1` in some fonts, some developers prefer to use upper-case literals.

Best practice

Prefer literal suffix L (upper case) over l (lower case).

Related content

We discuss string literals and suffixes in lesson 4.17 -- Introduction to std::string (https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/) and 4.18 -- Introduction to std::string_view (https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring_view/).

Additional (rarely used) suffixes exist for complex numbers and chrono (time) literals. These are documented here (https://en.cppreference.com/w/cpp/symbol_index/literals).

Integral literals

You generally won't need to use suffixes for integral literals, but here are examples:

```
#include <iostream>

int main()
{
    std::cout << 5; // 5 (no suffix) is type int (by
default)
    std::cout << 5L; // 5L is type long

    return 0;
}
```

One exception is the `u` (or 'U') suffix, which is used to denote an unsigned integer literal:

```
#include <iostream>

int main()
{
    unsigned int x { 5u }; // 5u is type unsigned
int
    std::cout << x;

    return 0;
}
```

Floating point literals

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

```
#include <iostream>

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

    return 0;
}
```

New programmers are often confused about why the following causes a compiler warning:

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

Because `4.1` has no suffix, the literal has type `double`, not `float`. When the compiler determines the type of a literal, it doesn't care what you're doing with the literal (e.g. in this case, using it to initialize a `float` variable). Since the type of the literal (`double`) doesn't match the type of the variable it is being used to initialize (`float`), the literal value must be converted to a `float` so it can then be used to initialize variable `f`. Converting a value from a `double` to a `float` can result in a loss of precision, so the compiler will issue a warning.

The solution here is one of the following:

```
float f { 4.1f }; // use 'f' suffix so the literal is a float and matches variable type of
float
double d { 4.1 }; // change variable to type double so it matches the literal type double
```

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 electronCharge { 1.6e-19 }; // charge on an electron is 1.6 x
10^-19
```

Magic numbers

A **magic number** is a literal (usually a number) that either has an unclear meaning or may need to be changed later.

Here are two statements showing examples of magic numbers:

```
constexpr int maxStudentsPerSchool{ numClassrooms * 30
};
setMax(30);
```

What do the literals `30` mean in these contexts? In the former, you can probably guess that it's the number of students per class, but it's not immediately obvious. In the latter, who knows. We'd have to go look at the function to know what it does.

In complex programs, it can be very difficult to infer what a literal 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.

To do so, we need to update one or more literal from `30` to `35`. But which literals? The `30` in the initializer of `maxStudentsPerSchool` seems obvious. But what about the `30` used as an argument 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` (of which there might be hundreds), and then make an individual determination as to whether it needs to change or not. That can be seriously time consuming (and error prone).

Fortunately, both the lack of context and the issues around updating can be easily addressed by using symbolic constants:

```
constexpr int maxStudentsPerClass { 30 };
constexpr int totalStudents{ numClassrooms * maxStudentsPerClass }; // now obvious what this 30
means

constexpr int maxNameLength{ 30 };
setMax(maxNameLength); // now obvious this 30 is used in a different context
```

The name of the constant provides context, and we only need to update a value in one place to make a change to the value across our entire program.

Note that magic numbers aren't always numbers -- they can also be text (e.g. names) or other types.

Literals used in obvious contexts that are unlikely to change are typically not considered magic. The values `-1`, `0`, `0.0`, and `1` are often used in such contexts:

```
int idGenerator { 0 };         // fine: we're starting our id generator with value
0
idGenerator = idGenerator + 1; // fine: we're just incrementing our generator
```

Other numbers may also be obvious in context (and thus, not considered magic):

```
int kmtoM(int km)
{
    return km * 1000; // fine: it's obvious 1000 is a conversion
factor
}
```

Best practice

Avoid magic numbers in your code (use constexpr variables instead).

Next lesson
4.16 Numeral systems (decimal, binary, hexadecimal, and octal)

Back to table of contents

Previous lesson
4.14 Compile-time constants, constant expressions, and constexpr