5.2 — Literals

learncpp.com/cpp-tutorial/literals/

Literals are values that are inserted directly into the code. For example:

```
return 5;                     // 5 is an integer literal
bool myNameIsAlex { true };   // true is a boolean literal
double d { 3.4 };             // 3.4 is a double literal
std::cout << "Hello, world!"; // "Hello, world!" is a C-style string literal
```

Literals are sometimes called **literal constants** because their meaning cannot be redefined (`5` always means the integral value 5).

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	Note
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]	see C-style string literals section below

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, LU	unsigned long
integral	ll or LL	long long
integral	ull, uLL, Ull, ULL, llu, llU, LLu, LLU	unsigned long long
integral	z or Z	The signed version of std::size_t (C++23)
integral	uz, uZ, Uz, UZ, zu, zU, Zu, ZU	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

Most of the suffixes are not case sensitive. The exceptions are:

- `s` and `sv` must be lower case.
- Two consecutive `l` or `L` characters must have the same casing.

Because lower-case `L` can look like numeric `1` in some fonts, some developers prefer to use upper-case literals. Others use lower case suffixes except for `L`.

Best practice

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

Related content

We discuss string literals and suffixes further in lesson 5.9 -- Introduction to std::string and 5.10 -- Introduction to std::string_view.

Additional (rarely used) suffixes exist for complex numbers and chrono (time) literals. These are documented here.

In most cases, suffixes aren't needed (except for `f`).

For advanced readers

Excepting the `f` suffix, suffixes are most often used in cases where type deduction is involved. See 10.8 -- Type deduction for objects using the auto keyword and 13.14 -- Class template argument deduction (CTAD) and deduction guides.

Integral literals

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

```
#include <iostream>

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

    return 0;
}
```

In most cases, it's fine to use non-suffixed `int` literals, even when initializing non-`int` types:

```
int main()
{
    int a { 5 };          // ok: types match
    unsigned int b { 6 }; // ok: compiler will convert int value 6 to unsigned int
value 6
    long c { 7 };         // ok: compiler will convert int value 7 to long value 7

    return 0;
}
```

In such cases, the compiler will convert the int literal to the appropriate type.

In the first case, `5` is already an `int` by default, so the compiler can use this value directly to initialize `int` variable `a`. In the second case, `int` value `6` doesn't match the type of `unsigned int b`. The compiler will convert int value `6` to `unsigned int` value `6`, and then use that as an initializer for `b`. In the third case, `int` value `7` doesn't match the type of `long c`. The compiler will convert int value `7` to `long` value `7`, and then use that as an initializer for `c`.

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

String literals

In programming, a **string** is a collection of sequential characters used to represent text (such as names, words, and sentences).

The very first C++ program you wrote probably looked something like this:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!";
    return 0;
}
```

`"Hello, world!"` is a string literal. String literals are placed between double quotes to identify them as strings (as opposed to char literals, which are placed between single quotes).

Because strings are commonly used in programs, most modern programming languages include a fundamental string data type. For historical reasons, strings are not a fundamental type in C++. Rather, they have a strange, complicated type that is hard to work with (we'll cover how/why in a future lesson, once we've covered more fundamentals required to explain how they work). Such strings are often called **C strings** or **C-style strings**, as they are inherited from the C-language.

There are two non-obvious things worth knowing about C-style string literals.

1. All C-style string literals have an implicit null terminator. Consider a string such as `"hello"`. While this C-style string appears to only have five characters, it actually has six: `'h'`, `'e'`, `'l`', `'l'`, `'o'`, and `'\0'` (a character with ASCII code 0). This trailing '\0' character is a special character called a **null terminator**, and it is used to indicate the end of the string. A string that ends with a null terminator is called a **null-terminated string**.

For advanced readers

This is the reason the string `"Hello, world!"` has type `const char[14]` rather than `const char[13]` -- the hidden null terminator counts as a character.

The reason for the null-terminator is also historical: it can be used to determine where the string ends.

2. Unlike most other literals (which are values, not objects), C-style string literals are const objects that are created at the start of the program and are guaranteed to exist for the entirety of the program. This fact will become important in a few lessons, when we discuss `std::string_view`.

Key insight

C-style string literals are const objects that are created at the start of the program and are guaranteed to exist for the entirety of the program.

Unlike C-style string literals, `std::string` and `std::string_view` literals create temporary objects. These temporary objects must be used immediately, as they are destroyed at the end of the full expression in which they are created.

Related content

We discuss `std::string` and `std::string_view` literals in lesson 5.9 -- Introduction to std::string and 5.10 -- Introduction to std::string_view respectively.

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:

```
const 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:

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

const 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, see lesson 5.5 -- Constexpr variables).