1.9 — Introduction to literals and operators

learncpp.com/cpp-tutorial/introduction-to-literals-and-operators/

Literals

Consider the following two statements:

```
std::cout << "Hello world!";
int x { 5 };
```

What are '"Hello world!"' and '5'? They are literals. A **literal** (also known as a **literal constant**) is a fixed value that has been inserted directly into the source code.

Literals and variables both have a value (and a type). Unlike a variable (whose value can be set and changed through initialization and assignment respectively), the value of a literal is fixed (`5` is always `5`). This is why literals are called constants.

To further highlight the difference between literals and variables, let's examine this short program:

```
#include <iostream>

int main()
{
    std::cout << 5 << '\n'; // print the value of a literal

    int x { 5 };
    std::cout << x << '\n'; // print the value of a variable
    return 0;
}
```

On line 5, we're printing the value `5` to the console. When the compiler compiles this, it will generate code that causes `std::cout` to print the value `5`. This value `5` is compiled into the executable and can be used directly.

On line 7, we're creating a variable named `x`, and initializing it with value `5`. The compiler will generate code that copies the literal value `5` into whatever memory location is given to `x`. On line 8, when we print `x`, the compiler will generate code that causes `std::cout` to print the value at the memory location of `x` (which has value `5`).

Thus, both output statements do the same thing (print the value 5). But in the case of the literal, the value `5` can be printed directly. In the case of the variable, the value `5` must be fetched from the memory the variable represents.

This also explains why a literal is constant while a variable can be changed. A literal's value is placed directly in the executable, and the executable itself can't be changed after it is created. A variable's value is placed in memory, and the value of memory can be changed while the executable is running.

Key insight

Literals are values that are inserted directly into the source code. These values usually appear directly in the executable code (unless they are optimized out).

Objects and variables represent memory locations that hold values. These values can be fetched on demand.

Related content

We talk more about literals in lesson 5.2 -- Literals.

Operators

In mathematics, an **operation** is a process involving zero or more input values (called **operands**) that produces a new value (called an *output value*). The specific operation to be performed is denoted by a symbol called an **operator**.

For example, as children we all learn that *2* + *3* equals *5*. In this case, the literals *2* and *3* are the operands, and the symbol + is the operator that tells us to apply mathematical addition on the operands to produce the new value *5*.

In C++, operations work as you'd expect. For example:

```
#include <iostream>

int main()
{
    std::cout << 1 + 2 << '\n';

    return 0;
}
```

In this program, the literals `1` and `2` are operands to the plus (`+`) operator, which produces the output value `3`. This output value is then printed to the console. In C++, the output value of an operation is often called a **return value**.

You are likely already quite familiar with standard arithmetic operators from common usage in mathematics, including addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). In C++, assignment (`=`) is an operator as well, as are insertion (`<<`), extraction (`>>`), and equality (`==`). While most operators have symbols for names (e.g. `+`, or `==`), there are also a number of operators that are keywords (e.g. `new`, `delete`, and `throw`).

Author’s note

For reasons that will become clear when we discuss operators in more detail, for operators that are symbols, it is common to append the operator’s symbol to the word *operator*.

For example, the plus operator would be written `operator+`, and the extraction operator would be written `operator>>`.

The number of operands that an operator takes as input is called the operator’s **arity**. Few people know what this word means, so don’t drop it in a conversation and expect anybody to have any idea what you’re talking about. Operators in C++ come in four different arities:

Unary operators act on one operand. An example of a unary operator is the `-` operator. For example, given `-5`, `operator-` takes literal operand `5` and flips its sign to produce new output value `-5`.

Binary operators act on two operands (often called *left* and *right*, as the left operand appears on the left side of the operator, and the right operand appears on the right side of the operator). An example of a binary operator is the `+` operator. For example, given `3 + 4`, `operator+` takes the left operand `3` and the right operand `4` and applies mathematical addition to produce new output value `7`. The insertion (`<<`) and extraction (`>>`) operators are binary operators, taking `std::cout` or `std::cin` on the left side, and the value to output or variable to input to on the right side.

Ternary operators act on three operands. There is only one of these in C++ (the conditional operator), which we’ll cover later.

Nullary operators act on zero operands. There is also only one of these in C++ (the throw operator), which we’ll also cover later.

Note that some operators have more than one meaning depending on how they are used. For example, `operator-` has two contexts. It can be used in unary form to invert a number’s sign (e.g. to convert `5` to `-5`, or vice versa), or it can be used in binary form to do subtraction (e.g. `4 - 3`).

Chaining operators

Operators can be chained together such that the output of one operator can be used as the input for another operator. For example, given the following: `2 * 3 + 4`, the multiplication operator goes first, and converts left operand `2` and right operand `3` into return value `6` (which becomes the left operand for the plus operator). Next, the plus operator executes, and converts left operand `6` and right operand `4` into new value `10`.

We'll talk more about the order in which operators execute when we do a deep dive into the topic of operators. For now, it's enough to know that the arithmetic operators execute in the same order as they do in standard mathematics: Parenthesis first, then Exponents, then Multiplication & Division, then Addition & Subtraction. This ordering is sometimes abbreviated *PEMDAS*, or expanded to the mnemonic "Please Excuse My Dear Aunt Sally".

Author's note

In some countries, PEMDAS is taught as PEDMAS, BEDMAS, BODMAS, or BIDMAS instead.

Return values and side effects

Most operators in C++ just use their operands to calculate a return value. For example, `-5` produces return value `-5` and `2 + 3` produces return value `5`. There are a few operators that do not produce return values (such as `delete` and `throw`). We'll cover what these do later.

Some operators have additional behaviors. An operator (or function) that has some observable effect beyond producing a return value is said to have a **side effect**. For example, when `x = 5` is evaluated, the assignment operator has the side effect of assigning the value `5` to variable `x`. The changed value of `x` is observable (e.g. by printing the value of `x`) even after the operator has finished executing. `std::cout << 5` has the side effect of printing `5` to the console. We can observe the fact that `5` has been printed to the console even after `std::cout << 5` has finished executing.

Operators with side effects are usually called for the behavior of the side effect rather than for the return value (if any) those operators produce.

Nomenclature

In common language, the term "side effect" is typically used to mean a secondary (often negative or unexpected) result of some other thing happening (such as taking medicine). For example, a common side effect of taking oral antibiotics is diarrhea. As such, we often think of side effects as things we want to avoid, or things that are incidental to the primary goal.

In C++, the term "side effect" has a different meaning: it is an observable effect of an operator or function beyond producing a return value.

Since assignment has the observable effect of changing the value of an object, this is considered a side effect. We use certain operators (e.g. the assignment operator) primarily for their side effects. In such cases, the side effect is both beneficial and predictable (and it is the return value that is often incidental).

For advanced readers

For the operators that we call primarily for their return values (e.g. `operator+` or `operator*`), it's usually obvious what their return values will be (e.g. the sum or product of the operands).

For the operators we call primarily for their side effects (e.g. `operator=` or `operator<<`), it's not always obvious what return values they produce (if any). For example, what return value would you expect `x = 5` to have?

Both `operator=` and `operator<<` (when used to output values to the console) return their left operand. Thus, `x = 5` returns `x`, and `std::cout << 5` returns `std::cout`. This is done so that these operators can be chained.

For example, `x = y = 5` evaluates as `x = (y = 5)`. First `y = 5` assigns `5` to `y`. This operation then returns `y`, which can then be assigned to `x`.

`std::cout << "Hello " << "world"` evaluates as `(std::cout << "Hello ") << "world!"`. This first prints `"Hello "` to the console. This operation returns `std::cout`, which can then be used to print `"world!"` to the console as well.

We talk more about the order in which operators evaluate in lesson 6.1 -- Operator precedence and associativity.

Quiz time

Question #1

For each of the following, indicate what output they produce:
a)

```
std::cout << 3 + 4 << '\n';
```

Show Solution

b)

```
std::cout << 3 + 4 - 5 << '\n';
```

Show Solution

c)

```
std::cout << 2 + 3 * 4 << '\n';
```

Show Solution

d) Extra credit:

```
int x { 2 };
std::cout << (x = 5) << '\n';
```

Show Solution