1.10 — Introduction to expressions

ALEX AUGUST 26, 2021

Expressions

Consider the following series of statements:

```
int x{ 2 };               // initialize variable x with value 2
int y{ 2 + 3 };           // initialize variable y with value 5
int z{ (2 * 3) + 4 };    // initialize variable z with value 10
int w{ y };               // initialize variable w with value 5 (the current value of variable
y)
```

Each of these statements defines a new variable and initializes it with a value. Note that the initializers shown above make use of a variety of different constructs: literals, variables, and operators. Somehow, C++ is converting each of these literals, variables, and operators into a single value that can be used as the initialization value for the variable.

What do all of these have in common? They make use of an expression.

An expression is a combination of literals, variables, operators, and explicit function calls (not shown above) that produce a single output value. When an expression is executed, each of the terms in the expression is evaluated until a single value remains (this process is called evaluation). That single value is the result of the expression.

Here are some examples of different kinds of expressions, with comments indicating how they evaluate:

```
2               // 2 is a literal that evaluates to value 2
"Hello world!"  // "Hello world!" is a literal that evaluates to text "Hello
world!"
x               // x is a variable that evaluates to the value of x
2 + 3           // 2 + 3 uses operator + to evaluate to value 5
x = 2 + 3      // 2 + 3 evaluates to value 5, which is then assigned to variable x
std::cout << x  // x evaluates to the value of x, which is then printed to the
console
```

As you can see, literals evaluate to their own values. Variables evaluate to the value of the variable. We haven't covered function calls yet, but in the context of an expression, function calls evaluate whatever value the function returns. And operators let us combine multiple values together to produce a new value.

Note that expressions do not end in a semicolon, and cannot be compiled by themselves. For example, if you were to try compiling the expression *x = 5*, your compiler would complain (probably about a missing semicolon). Rather, expressions are always evaluated as part of statements.

For example, take this statement:

```
int x{ 2 + 3 }; // 2 + 3 is an expression that has no semicolon -- the semicolon is at the end of the statement containing the expression
```

If you were to break this statement down into its syntax, it would look like this:

```
type identifier { expression };
```

Type could be any valid type (we chose *int*). *Identifier* could be any valid name (we chose *x*). And *expression* could be any valid expression (we chose *2 + 3*, which uses 2 literals and an operator).

Key insight

Wherever you can use a single value in C++, you can use an expression instead, and the expression will be evaluated to produce a single value.

Expression statements

Certain expressions (like *x = 5*) are useful by themselves. However, we mentioned above that expressions must be part of a statement, so how can we use these expressions by themselves?

Fortunately, we can convert any expression into an equivalent statement (called an expression statement). An expression statement is a statement that consists of an expression followed by a semicolon. When the statement is executed, the expression will be evaluated (and the result of the expression will be discarded).

Thus, we can take any expression (such as*x = 5*), and turn it into an expression statement (such as*x = 5;*) that will compile.

Note that we can make expression statements that compile but are meaningless/useless (e.g.*2 * 3;*). This expression evaluates to 6, and then the value 6 is discarded.

Rule

Values calculated in an expression are discarded at the end of the expression.

Quiz time

Question #1

What is the difference between a statement and an expression?

Show Solution

Question #2

Indicate whether each of the following lines are *statements that do not contain expressions*, *statements that contain expressions*, or are *expression statements*.

a)

```
int
x;
```

Show Solution

b)

```
int x =
5;
```

Show Solution

c)

```
x =
5;
```

Show Solution

d) Extra credit:

```
std::cout << x; // Hint: operator<< is a binary
operator.
```

Show Solution

Question #3

Determine what values the following program outputs. Do not compile this program. Just work through it line by line in your head.

```cpp
#include <iostream>

int main()
{
 std::cout << 2 + 3 <<
'\n';

 int x{ 6 };
 int y{ x - 2 };
 std::cout << y << '\n';

 int z{ 0 };
 z = x;
 std::cout << z - x <<
'\n';

 return 0;
}
```

Show Solution

Next lesson
1.11 Developing your first program

Back to table of contents

Previous lesson
1.9 Introduction to literals and operators

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ co

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X