1.10 — Introduction to expressions

Expressions

Consider the following series of statements:

```
// five() is a function that returns the value 5
int five()
{
    return 5;
}

int main()
{
    int a{ 2 };             // initialize variable a with literal value 2
    int b{ 2 + 3 };         // initialize variable b with computed value 5
    int c{ (2 * 3) + 4 };   // initialize variable c with computed value 10
    int d{ b };             // initialize variable d with variable value 5
    int e{ five() };        // initialize variable e with function return value 5

    return 0;
}
```

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, operators, and function calls. Somehow, C++ is converting all of these different things into a single value that can then be used as the initialization value for the variable.

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

An **expression** is a sequence of literals, variables, operators, and function calls that calculates a single value. The process of executing an expression is called **evaluation**, and the single value produced is called the **result** of the expression.

Related content

While most expressions are used to calculate a value, expressions can also identify an object (which can be evaluated to get the value held by the object) or a function (which can be called to get the value returned by the function). We talk more about this in lesson 12.2 -- Value categories (lvalues and rvalues).

For now, we'll assume all expressions calculate values.

When an expression is evaluated, each of the terms inside the expression are evaluated, until a single value remains. 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           // operator+ uses operands 2 and 3 to evaluate to value 5
five()          // evaluates to the return value of function five()
```

As you can see, literals evaluate to their own values. Variables evaluate to the value of the variable. Operators (such as `operator+`) use their operands to evaluate to some other value. We haven't covered function calls yet, but in the context of an expression, function calls evaluate to whatever value the function returns.

Expressions involving operators with side effects are a little more tricky:

```
x = 5           // x = 5 has side effect of assigning 5 to x, evaluates to x
x = 2 + 3       // has side effect of assigning 5 to x, evaluates to x
std::cout << x  // has side effect of printing value of x to console, evaluates to
std::cout
```

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 two literals and an operator).

Key insight

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

Expression statements

Certain expressions (like `x = 5`) are useful for their side effects (in this case, to assign the value `5` to the variable `x`). However, we mentioned above that expressions cannot be executed by themselves -- they must exist as part of a statement. So how can we use such expressions?

Fortunately, it's easy to convert any expression into an equivalent statement. An **expression statement** is a statement that consists of an expression followed by a semicolon. When the expression statement is executed, the expression will be evaluated.

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

When an expression is used in an expression statement, any return value generated by the expression is discarded (because it is not used).

Useless expression statements

We can also make expression statements that compile but have no effect. For example, the expression statement (`2 * 3;`) is an expression statement whose expression evaluates to the result value of *6*, which is then discarded. While syntactically valid, such expression statements are useless. Some compilers (such as gcc and Clang) will produce warnings if they can detect that an expression statement is useless.

Subexpressions, full expressions, and compound expressions

We occasionally need to talk about specific kinds of expressions. For this purpose, we will define some related terms.

Consider the following expressions:

```
2               // 2 is a literal that evaluates to value 2
2 + 3           // 2 + 3 uses operator + to evaluate to value 5
x = 4 + 5       // 4 + 5 evaluates to value 9, which is then assigned to variable x
```

Simplifying a bit, a **subexpression** is an expression used as an operand. For example, the subexpressions of `x = 4 + 5` are `x` and `4 + 5`. The subexpressions of `4 + 5` are `4` and `5`.

A **full expression** is an expression that is not a subexpression. All three expressions above (`2`, `2 + 3`, and `x = 4 + 5`) are full expressions.

In casual language, a **compound expression** is an expression that contains two or more uses of operators. `x = 4 + 5` is a compound expression because it contains two uses of operators (`operator=` and `operator+`). `2` and `2 + 3` are not compound expressions.

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:

```
foo(); // foo is a function
```

Show Solution

e) 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.

```
#include <iostream>

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

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

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

	return 0;
}
```

Show Solution