5.7 — For statements

BY ALEX ON JUNE 25TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 7TH, 2018

By far, the most utilized looping statement in C++ is the *for statement*. The **for statement** (also called a **for loop**) is ideal when we know exactly how many times we need to iterate, because it lets us easily define, initialize, and change the value of loop variables after each iteration.

The *for statement* looks pretty simple in abstract:

```
for (init-statement; condition-expression; end-expression)
   statement
```

The easiest way to understand a *for loop* is to convert it into an equivalent *while loop*:

```
{ // note the block here
    init-statement;
    while (condition-expression)
    {
        statement;
        end-expression;
    }
} // variables defined inside the loop go out of scope here
```

The variables defined inside a *for loop* have a special kind of scope called *loop scope*. Variables with **loop scope** exist only within the loop, and are not accessible outside of it.

Evaluation of for statements

A *for statement* is evaluated in 3 parts:

1) The init-statement is evaluated. Typically, the init-statement consists of variable definitions and initialization. This statement is only evaluated once, when the loop is first executed.

2) The condition-expression is evaluated. If this evaluates to false, the loop terminates immediately. If this evaluates to true, the statement is executed.

3) After the statement is executed, the end-expression is evaluated. Typically, this expression is used to increment or decrement the variables declared in the init-statement. After the end-expression has been evaluated, the loop returns to step 2.

Let's take a look at a sample *for loop* and discuss how it works:

```
1 for (int count=0; count < 10; ++count)
2     std::cout << count << " ";
```

First, we declare a loop variable named count, and assign it the value 0.

Second, `count < 10` is evaluated, and since count is 0, `0 < 10` evaluates to true. Consequently, the statement executes, which prints 0.

Third, ++count is evaluated, which increments count to 1. Then the loop goes back to the second step.

Now, `1 < 10` is evaluated to true, so the loop iterates again. The statement prints 1, and count is incremented to 2.
`2 < 10` evaluates to true, the statement prints 2, and count is incremented to 3. And so on.

Eventually, count is incremented to 10, `10 < 10` evaluates to false, and the loop exits.

Consequently, this program prints the result:

```
0 1 2 3 4 5 6 7 8 9
```

For loops can be hard for new programmers to read -- however, experienced programmers love them because they are a very compact way to do loops of this nature. For the sake of example, let's uncompact the above *for loop* by converting it into an equivalent *while loop*:

```
{ // outer braces ensure loop scope
    int count = 0;
    while (count < 10)
    {
        std::cout << count << " ";
        ++count;
    }
}
```

That doesn't look so bad, does it? Note that the outer braces are necessary here, because count goes out of scope when the loop ends.

More for loop examples

Here's an example of a *for loop* used to calculate an exponentiation of integers:

```
// returns the value nBase ^ nExp
int pow(int base, int exponent)
{
    int total = 1;

    for (int count=0; count < exponent; ++count)
        total *= base;

    return total;
}
```

This function returns the value base^exponent (base to the exponent power).

This is a straightforward incrementing *for loop*, with count looping from 0 up to (but excluding) exponent.

If exponent is 0, the *for loop* will execute 0 times, and the function will return 1.
If exponent is 1, the *for loop* will execute 1 time, and the function will return 1 * base.
If exponent is 2, the *for loop* will execute 2 times, and the function will return 1 * base * base.

Although most *for loops* increment the loop variable by 1, we can decrement it as well:

```
for (int count = 9; count >= 0; --count)
    std::cout << count << " ";
```

This prints the result:

```
9 8 7 6 5 4 3 2 1 0
```

Alternately, we can change the value of our loop variable by more than 1 with each iteration:

```
for (int count = 9; count >= 0; count -= 2)
    std::cout << count << " ";
```

This prints the result:

```
9 7 5 3 1
```

Off-by-one errors

One of the biggest problems that new programmers have with *for loops* (and other kinds of loops) is off-by-one errors. **Off-by-one errors** occur when the loop iterates one too many or one too few times. This generally happens because the wrong relational operator is used in the conditional-expression (eg. > instead of >=). These errors can be hard to track down because the compiler will not complain about them -- the program will run fine, but it will produce the wrong result.

When writing *for loops*, remember that the loop will execute as long as the conditional-expression is true. Generally it is a good idea to test your loops using known values to make sure that they work as expected. A good way to do this is to test your loop with known inputs that cause it to iterate 0, 1, and 2 times. If it works for those, it will likely work for any number of iterations.

Rule: Test your loops with known inputs that cause it to iterate 0, 1, and 2 times.

Omitted expressions

It is possible to write *for loops* that omit any or all of the expressions. For example, in the following example, we'll omit the init-statement and end-expression:

```
int count=0;
for ( ; count < 10; )
{
    std::cout << count << " ";
    ++count;
}
```

This *for loop* produces the result:

```
0 1 2 3 4 5 6 7 8 9
```

Rather than having the *for loop* do the initialization and incrementing, we've done it manually. We have done so purely for academic purposes in this example, but there are cases where not declaring a loop variable (because you already have one) or not incrementing it (because you're incrementing it some other way) are desired.

Although you do not see it very often, it is worth noting that the following example produces an infinite loop:

```
for (;;)
    statement;
```

The above example is equivalent to:

```
while (true)
    statement;
```

This might be a little unexpected, as you'd probably expect an omitted condition-expression to be treated as "false". However, the C++ standard explicitly (and inconsistently) defines that and omitted condition-expression in a for loop should be treated as "true".

We recommend avoiding this form of the for loop altogether and using while(true) instead.

Multiple declarations

Although for loops typically iterate over only one variable, sometimes *for loops* need to work with multiple variables. When this happens, the programmer can make use of the comma operator in order to assign (in the init-statement) or change (in the end-statement) the value of multiple variables:

```
    int iii, jjj;
    for (iii=0, jjj=9; iii < 10; ++iii, --jjj)
        std::cout << iii << " " << jjj << endl;
```

This loop assigns values to two previously declared variables: iii to 0, and jjj to 9. It iterates iii over the range 0 to 9, and each iteration iii is incremented and jjj is decremented.

This program produces the result:

```
0 9
1 8
2 7
3 6
4 5
5 4
```

```
6 3
7 2
8 1
9 0
```

This is the only place in C++ where the comma operator typically gets used.

Note: More typically, we'd write the above loop as:

```
    for (int iii=0, jjj=9; iii < 10; ++iii, --jjj)
        std::cout << iii << " " << jjj << endl;
```

In this case, the comma in the init-statement is part of the variable definition syntax, not a use of the comma operator. But the effect is identical.

For loops in old code

In older versions of C++, variables defined as part of the init-statement did not get destroyed at the end of the loop. This meant that you could have something like this:

```
for (int count=0; count < 10; ++count) // count defined here
    std::cout << count << " ";

// count is not destroyed in older compilers

std::cout << "\n";
std::cout << "I counted to: " << count << "\n"; // so you can still use it here
```

This use has been disallowed, but you may still see it in older code.

Nested for loops

Like other types of loops, for loops can be nested inside other loops. In the following example, we're nesting a for loop inside another for loop:

```
#include <iostream>

int main()
{
    for (char c = 'a'; c <= 'e'; ++c) // outer loop on letters
    {
        std::cout << c; // print our letter first

        for (int i = 0; i < 3; ++i) // inner loop on all numbers
            std::cout << i;

        std::cout << '\n';
    }

    return 0;
}
```

For each iteration of the outer loop, the inner loop runs in its entirety. Consequently, the output is:

```
a012
b012
c012
d012
e012
```

Here's some more detail on what's happening here. The outer loop runs first, and char c is set to 'a'. Then the loop body executes, with c set to 'a'. This prints 'a', executes the inner loop entirely (which prints '0', '1', and '2'). Then a newline is printed. Now the loop body is finished, so the outer loop returns to the top and the loop condition is re-evaluated. Since the loop condition is true, variable c (with value 'a') is incremented (to value 'b'), and the next iteration of the outer loop begins. This prints ("b012\n"). And so on.

Conclusion

For statements are the most commonly used loop in the C++ language. Even though its syntax is typically a bit confusing to new programmers, you will see *for loops* so often that you will understand them in no time at all!

Quiz

1) Write a *for loop* that prints every even number from 0 to 20.

2) Write a function named sumTo() that takes an integer parameter named value, and returns the sum of all the numbers from 1 to value.

For example, sumTo(5) should return 15, which is 1 + 2 + 3 + 4 + 5.

Hint: Use a non-loop variable to accumulate the sum as you iterate from 1 to the input value, much like the pow() example above uses the total variable to accumulate the return value each iteration.

3) What's wrong with the following for loop?

```
// Print all numbers from 9 to 0
for (unsigned int count=9; count >= 0; --count)
    std::cout << count << " ";
```

Quiz solutions

1) **Hide Solution**

```
for (int count=0; count <= 20; count += 2)
    std::cout << count << std::endl;
```

2) **Hide Solution**

```
int sumTo(int value)
{
    int total(0);
    for (int count=1; count <= value; ++count)
        total += count;

    return total;
}
```

3) **Hide Solution**

This for loop executes as long as count >= 0. In other words, it runs until count is negative. However, because count is unsigned, count can never go negative. Consequently, this loop will run for-ever (ha ha)! Generally, it's a good idea to avoid looping on unsigned variables unless necessary.


5.8 -- Break and continue


Index


5.6 -- Do while statements