7.9 — For statements

ALEX AUGUST 30, 2021

By far, the most utilized loop statement in C++ is the `for statement`. The **for statement** (also called a **for loop**) is preferred when we have an obvious loop variable because it lets us easily and concisely define, initialize, test, and change the value of loop variables.

As of C++11, there are two different kinds of `for loops`. We'll cover the classic `for statement` in this lesson, and the newer `range-based for statement` in a future lesson (10.19 -- For-each loops) once we've covered some other prerequisite topics, such as arrays and iterators.

The `for statement` looks pretty simple in abstract:

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

The easiest way to initially understand how a `for statement` works is to convert it into an equivalent `while statement`:

```
{ // note the block here
    init-statement; // used to define variables used in the loop
    while (condition)
    {
        statement;
        end-expression; // used to modify the loop variable prior to reassessment of the condition
    }
} // variables defined inside the loop go out of scope here
```

Evaluation of for statements

A `for statement` is evaluated in 3 parts:

First, the init-statement is executed. This only happens once when the loop is initiated. The init-statement is typically used for variable definition and initialization. These variables have "loop scope", which really just is a form of block scope where these variables exist from the point of definition through the end of the loop statement. In our while-loop equivalent, you can see that the init-statement is inside a block that contains the loop, so the variables defined in the init-statement go out of scope when the block containing the loop ends.

Second, for each loop iteration, the condition is evaluated. If this evaluates to true, the statement is executed. If this evaluates to false, the loop terminates and execution continues with the next statement beyond the loop.

Finally, after the statement is executed, the end-expression is evaluated. Typically, this expression is used to increment or decrement the loop variables defined in the init-statement. After the end-expression has been evaluated, execution returns to the second step (and the condition is evaluated again).

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

```
#include <iostream>

int main()
{
    for (int count{ 1 }; count <= 10;
++count)
        std::cout << count << ' ';

    return 0;
}
```

First, we declare a loop variable named `count`, and initialize it with the value `1`.

Second, `count <= 10` is evaluated, and since count is `1`, this evaluates to `true`. Consequently, the statement executes, which prints `1` and a space.

Finally, `++count` is evaluated, which increments `count` to `2`. Then the loop goes back to the second step.

Now, `count <= 10` is evaluated again. Since `count` has value `2`, this evaluates `true`, so the loop iterates again. The statement prints `2` and a space, and count is incremented to `3`. The loop continues to iterate until eventually `count` is incremented to `11`, at which point `count <= 10` evaluates to `false`, and the loop exits.

Consequently, this program prints the result:

```
1 2 3 4 5 6 7 8 9 10
```

For the sake of example, let's convert the above `for loop` into an equivalent `while loop`:

```
#include <iostream>

int main()
{
    { // the block here ensures block scope for
count
        int count{ 1 }; // our init-statement
        while (count <= 10) // our condition
        {
            std::cout << count << ' '; // our
statement
            ++count; // our end-expression
        }
    }
}
```

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.

`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 with a counter, with all of the necessary information about the loop variables, loop conditions, and loop count modifiers are presented up front. This helps reduce errors.

More for loop examples

Here's an example of a function that uses a `for loop` to calculate integer exponents:

```
// returns the value base ^ exponent -- watch out for
overflow!
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:

```
#include <iostream>

int main()
{
    for (int count{ 9 }; count >= 0; --count)
        std::cout << count << ' ';

    return 0;
}
```

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:

```
#include <iostream>

int main()
{
    for (int count{ 9 }; count >= 0; count -= 2)
    std::cout << count << ' ';

    return 0;
}
```

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 loops that utilize counters) are `off-by-one errors`. **Off-by-one errors** occur when the loop iterates one too many or one too few times to produce the desired result.

Here's an example:

```
#include <iostream>

int main()
{
    // oops, we used operator< instead of operator<=
    for (unsigned int count{ 1 }; count < 5; ++count)
    {
        std::cout << count << ' ';
    }

    return 0;
}
```

This program is supposed to print `1 2 3 4 5`, but it only prints `1 2 3 4` because we used the wrong relational operator.

Although the most common cause for these errors is using the wrong relational operator, they can sometimes occur by using pre-increment or pre-decrement instead of post-increment or post-decrement, or vice-versa.

Omitted expressions

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

```
#include <iostream>

int main()
{
    int count{ 0 };
    for ( ; count < 10; ) // no init-statement or end-expression
    {
        std::cout << count << ' ';
        ++count;
    }

    return 0;
}
```

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 in the end-expression (because you're incrementing it some other way) is 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 an 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.

For loops with multiple counters

Although `for loops` typically iterate over only one variable, sometimes `for loops` need to work with multiple variables. To assist with this, the programmer can define multiple variables in the init-statement, and can make use of the comma operator to change the value of multiple variables in the end-expression:

```
#include <iostream>

int main()
{
    for (int x{ 0 }, y{ 9 }; x < 10; ++x, --y)
        std::cout << x << ' ' << y << '\n';

    return 0;
}

```

This loop defines and initializes two new variables: `x` and `y`. It iterates `x` over the range `0` to `9`, and after each iteration `x` is incremented and `y` 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 about the only place in C++ where defining multiple variables in the same statement, and use of the comma operator is considered an acceptable practice.

Best practice

Defining multiple variables (in the init-statement) and using the comma operator (in the end-expression) is acceptable inside a `for statement`.

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 initialized to `'a'`. Then `c <= 'e'` is evaluated, which is `true`, so the loop body executes. Since `c` is set to `'a'`, this first prints `a`. Next the inner loop executes entirely (which prints `0`, `1`, and `2`). Then a newline is printed. Now the outer loop body is finished, so the outer loop returns to the top, `c` is incremented to `'b'`, and the loop condition is re-evaluated. Since the loop condition is still `true` 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!

`For statements` excel when you have a counter variable. If you do not have a counter, a `while statement` is probably a better choice.

Best practice

Prefer `for loops` over `while loops` when there is an obvious loop variable.
Prefer `while loops` over `for loops` when there is no obvious loop variable.

Quiz time

Question #1

Write a `for loop` that prints every even number from 0 to 20.

Show Solution

Question #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.

Show Solution

Question #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 << ' ';
```

Show Solution

Next lesson
7.10 Break and continue

Back to table of contents

Previous lesson
7.8 Do while statements

@ Email*

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

Notify me about replies: POST COMMENT

DP N N FOUT

Newest