Lecture 9 — Loops II

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 12, 2016

Acknowledgments: D.W. Harder, W.D. Bishop

Loops, Continued

Last time, we examined the `while` and `do-while` loops.

We also mentioned, but did not examine, the third kind of loop:
the `for` loop.

Let's start by re-examining a `while` loop.

Countdown Loop

```
int countdown = 10;
while ( countdown > 0 )
{
    cout << countdown << endl;
    countdown--;
}
```

This is a counted loop. Observe three statements:

1. `int countdown = 10;`
   Declares and initializes the counter variable.
2. `while (countdown > 0)`
   The loop test condition.
3. `countdown--;`
   Decrement the counter variable.

Counted Loops

Counted loops are very common. The `for` loop syntax puts all three of those statements in one line. The proper names for these elements:

1. `int countdown = 10;`
   Initialization
2. `while (countdown > 0)`
   Condition
3. `countdown--;`
   Modification

Now we can see the syntax of the for loop.

The `for` Loop

for (*initialization*; *condition*; *modification*)

Using the countdown example:

```
for ( int countdown = 10; countdown > 0; countdown-- )
```

And as always, the declaration is followed by the { and } braces to enclose the loop body (statement block).

Rewriting the Countdown Loop

Let's rewrite the countdown loop using `for`:

```
for ( int countdown = 10; countdown > 0; countdown-- )
{
    cout << countdown << endl;
}
```

Important note: there is no semicolon after the modification expression. A `for` loop has exactly two semicolons.

(Technical explanation: the three elements are expressions and not statements, and that's why no semicolon is required.)

Execution Order of the `for` Loop

Let's examine the execution order of this loop:

```
for ( int i = 0; i < 1; ++i )
{
    // Loop Body
}
```

1. The initialization: `int i = 0;`
2. The condition is checked: `i < 1;`
3. The condition is true, so the loop body executes
4. The modification: `++i`
5. Control goes back to the start of the loop; condition is checked
6. The condition is false, so the loop body is skipped

For Loop Execution Comments

Like the `while` loop, the `for` loop is a pretest loop.

If the condition is not satisfied, the loop body does not execute.
This may mean the loop executes zero times.

Optional Parts of the `for` Loop

Technically, each of the 3 elements of the `for` statement are optional.

In this example, initialization is missing:

```
int countdown;
cin >> countdown;
for ( ; countdown > 0; countdown-- )
{
    cout << countdown << endl;
}
```

The semicolon is still needed after the spot where the initialization would otherwise go.

Optional Parts of the `for` Loop

In this example, the modification expression is elsewhere in the loop:

```
for ( int countdown = 10; countdown > 0; )
{
    cout << countdown << endl;
    countdown--;
}
```

Optional Parts of the `for` Loop

Let's combine the last two examples:

```
int countdown;
cin >> countdown;

for ( ; countdown > 0; ) {
    cout << countdown << endl;
    countdown--;
}
```

This is also acceptable. If you use only the middle expression (condition) of the `for` loop, it works just like the `while` loop.

(In fact, any `while` loop can be trivially converted to a `for` loop by using this syntax.)

Optional Parts of the `for` Loop

And if we delete the condition, too?

```
int countdown;
cin >> countdown;
for ( ; ; ) {
    cout << countdown << endl;
    countdown--;
    if ( countdown < 0 ) {
        break;
    }
}
```

If no condition is specified, it's an infinite loop; the same as if we wrote `while (true)`.

For Loop Flexibility

The elements of the `for` loop are pretty flexible. The initialization and modification expressions may do more than one thing.

The expressions are separated by commas.

```
for ( int x = 10, int y = 0; x > 0; x--, y++ )
{
    // Loop Body
}
```

This is uncommon and may be confusing, especially to non-experts. It is not recommended as a good practice.

Pitfall: Extra Semicolon

Potential pitfall: there is an extra semicolon after the closing bracket.

```
for ( int x = 10; x > 0; x-- );
    cout << "Hello World" << endl;
```

The for loop will run, but the `cout` statement will execute only once.

The semicolon creates an "empty statement" and that forms the body of the loop. That empty statement executes with each iteration.

Solution: like with `if`, use the { and } braces to prevent this problem.

Use of `break` and `continue`

The `break` and `continue` statements can also appear in the body of the `for` loop as they do in the `while` loop.

As before, `break` exits from the loop and takes us to the next statement after the loop.

`continue` still means "go back to the start of the loop", but the modification statement is executed before the condition is checked.

Use of `continue` in a `for` Loop

In this example, numbers that are multiples of 8 are not printed.

```
int numberPrinted = 0;
for ( int i = 0; i < 100; ++i ) {
    if ( i % 8 == 0 ) {
        continue;
    }
    cout <<  i;
    numberPrinted += 1;
}
cout << numberPrinted << " numbers printed in total." << endl;
```

If `i` is a multiple of 8, the next iteration of the loop is started.

Confusing `continue` and `break`

What if instead of using `continue` we used `break` instead?

```
int numberPrinted = 0;
for ( int i = 0; i < 100; ++i ) {
    if ( i % 8 == 0 ) {
        break;
    }
    cout <<  i;
    numberPrinted += 1;
}
cout << numberPrinted << " numbers printed in total." << endl;
```

Nested Loops

It is legal to nest one loop statement inside another.
    Even different types (e.g., a `for` inside a `while`).

When nesting `for` loops, they should use different counter variables.
    Common choices are `i`, `j`, and `k`.
    Choosing more descriptive names can be helpful.

A `break` or `continue` statement applies to the innermost loop statement containing that statement.

(We can also put a `switch` statement inside a loop; a `break` statement at the end of an option applies to the `switch`, not the loop.)

Nested Loop Example

```
for ( int i = 0; i < 10; i++ ) {
    for (int j = 0; j < 10; j++ ) {
        cout << "i = ";
        cout << i;
        cout << ", j = ";
        cout << j;
        cout << endl;
    }
}
```

Nested Loop with break Example

```
for ( int i = 0; i < 10; i++ ) { // Outer loop
    for (int j = 0; j < 10; j++ ) { // Inner loop
        if ( j == 9 ) {
            break; // End inner loop
        }
        cout << "i = ";
        cout << i;
        cout << ", j = ";
        cout << j;
        cout << endl;
    }
}
```

Comments on Iteration Statements

As you've seen, it doesn't matter what iteration statement you choose; It's possible to rewrite any loop as another kind.

Some tips about when to use each of the loops:

- The `for` statement is a slightly better choice for deterministic loops (i.e., loops that require a fixed number of iterations)
- The `while` statement is slightly better for non-deterministic loops (i.e., loops that require a flexible number of iterations)
- The `do-while` statement is slightly better for statement blocks that must execute at least once (but is still not recommended)