

# Lecture 8 – Loops

J. Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

July 12, 2016

Acknowledgments: W.D. Bishop

Loops are another kind of control statement: **iteration** statements.

Iteration: the repetition of a group of statements.

Using a loop statement in code means a block of statements is repeated for some number of iterations.

This may be a fixed number or vary based on the state of the program.

Some examples:

A loop with fixed iterations might repeat a block of statements exactly 10 times.

A loop with a variable number of iterations might repeat a block of statements until the user enters the letter 'Q' to quit.

**Pretest** loops evaluate one or more expressions prior to executing the statement block.

**Posttest** loops execute the statement block once prior to evaluating one or more expressions.

Loops come in a few different varieties:

- Counter-controlled loops increment / decrement a counter until the counter reaches a threshold
- Logically-controlled loops evaluate an expression and terminate when the expression is false
- User-controlled loops can break out of the loop anywhere within the statement block

C++ provides four types of loops:

- The `while` loop
- The `do-while` loop
- The `for` loop
- The `foreach` loop (in C++11 and higher)

Let's just jump right in and look at the syntax.

This is the syntax for a `while` loop.

```
while ( condition )  
{  
    // Loop Body statements  
}
```

Like the `if`-statement, *condition* in this is a boolean expression.

The `while` loop is a pretest loop: the condition is evaluated first.

Like the if-statement, *condition* is evaluated and if it is true, the block of statements in the { } are executed.

That block of statements is referred to as the **loop body**.

If *condition* is false, the statements of the loop body are not executed.

It may happen that the body of the loop never executes.

# The while Loop: Repetition

What makes the `while` different from `if` is the **repetition**.

At the end of the statement body (the `}` character), control goes back to the `while` statement.

The condition is evaluated again.

Same applies: if true, the loop body is executed.

This continues until the condition evaluates to false.



# The while Loop: Countdown

Here's another example of a while loop:

```
int countdown = 10;

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

[Demo: output of this code.]

What happens if we forget the `countdown--;` statement in that loop?

The variable `countdown` remains at 10 and the `while` condition will always evaluate to true.

This will go on indefinitely (or until you get frustrated and close the program). The term for this is an **infinite loop**.

An infinite loop is very often an error condition.

Most programs should terminate at some point.

In some circumstances, however, the infinite loop is intended: the program should never terminate.

Example: the software in your router. On boot up it starts running its program, and continues, never ending (until the plug is pulled).

To create an infinite loop, write `while( true )`.

There are two special statements that can be written in the loop body that control the flow of execution.

The first of these is the `break` statement.

Yes, this is the same keyword as in the `switch` statement.

The context indicates it means something slightly different.

When the `break` statement executes, it means “exit the loop now”.

A `break` statement may be used to jump out of a loop, even an infinite one, in the middle of a statement block.

If possible, `break` statements should be avoided as they can result in code that is more difficult to debug.

The rule of thumb is that the `break` statement should be used if the code is clearer with it than it would be without it.

Multiple `break` statements can exist in a loop if multiple conditions for exiting the loop need to be evaluated.

```
int main ( ) {  
    int counter = 0;  
  
    while( counter < 500 ) {  
  
        counter++;  
        if ( counter > 4 ) {  
            break;  
        }  
        cout << counter <<endl;  
    }  
    return 0;  
}
```

[Demo: output of this program]

A `break` statement completely ends the loop, no matter if the loop condition is true or not.

If you write an infinite loop (`while( true )`) one way to use this properly is to have a condition inside that uses `break`.

```
int main ( ) {  
    cout << "Enter a negative number to exit." << endl;  
    int number = 0;  
  
    while (number >= 0) {  
        cout << "Enter a number: ";  
        int number;  
        cin >> number;  
    } // End of loop  
  
    cout << "Negative number entered." << endl;  
    return 0;  
}
```



## While loop with User Input & Break

Let's rewrite this with break:

```
int main ( ) {  
    cout << "Enter a negative number to exit." << endl;  
    int number = 0;  
  
    while (true) {  
        cout <<"Enter a number: ";  
        int number;  
        cin >> number;  
        if ( number < 0 ) {  
            break;  
        }  
    }  
    cout << "Negative number entered." << endl;  
    return 0;  
}
```

The other loop body statement that controls execution is the `continue` statement.

The `continue` statement works a lot like the `break` statement, except instead of exiting the loop, it means “go back to the start of the loop”.

In the `while` loop, the condition is tested, and if it's still true, the next iteration of the loop executes.

```
int main ( ) {  
    int counter = 0;  
  
    while( counter < 10 ) {  
        counter++;  
        if( counter == 4 )  
        {  
            continue;  
        }  
        cout << counter << endl;  
    }  
    return 0;  
}
```

[Demo: output of this program]

Like the `break` statement, `continue` should be avoided if possible, as they can result in code that is more difficult to debug.

Multiple `continue` statements can exist in a loop if multiple conditions for going to the next iteration of the loop exist.

If the loop condition is no longer true, use of `continue` takes us to testing the condition; it will evaluate to false, and the loop ends.

In this way, use of `continue` may have the same outcome as `break`, though it gets there by a different path.

A variant of the `while` loop that remains in the language for historical reasons is the `do-while` loop.

Its syntax is a lot like the `while` loop:

```
do
{
    // Loop body

} while ( condition );
```

Note the semicolon that appears after the condition's closing bracket.

Some important things to observe about the do-while loop.

The loop body is preceded by do to indicate the start of the loop.

The condition is checked at the end of the loop body, not beginning.  
This is a posttest loop.

Key observation: the loop body will execute at least once, even if the condition is false.

Let's compare this while loop:

```
int count;  
cin >> count;  
  
while ( count > 0 ) {  
    cout << count << endl;  
    count--;  
}
```

...with this one:

```
int count;  
cin >> count;  
do {  
    cout << count << endl;  
    count--;  
} while ( count > 0 );
```

It is still possible to write infinite loops with `do-while`.

Similarly, the `break` and `continue` statements work the same way.

The use of `do-while` is not recommended as it is really only in the language for historical reasons; use the `while` loop instead.