# 8.8 — Introduction to loops and while statements

ALEX SEPTEMBER 11, 2023

## Introduction to loops

And now the real fun begins -- in the next set of lessons, we'll cover loops. Loops are control flow constructs that allow a piece of code to execute repeatedly until some condition is met. Loops add a significant amount of flexibility into your programming toolkit, allowing you to do many things that would otherwise be difficult.

For example, let's say you wanted to print all the numbers between 1 and 10. Without loops, you might try something like this:

```
#include <iostream>

int main()
{
    std::cout << "1 2 3 4 5 6 7 8 9 10";
    std::cout << " done!\n";
    return 0;
}
```

While that's doable, it becomes increasingly less so as you want to print more numbers: what if you wanted to print all the numbers between 1 and 1000? That would be quite a bit of typing! But such a program is writable in this way because we know at compile time how many numbers we want to print.

Now, let's change the parameters a bit. What if we wanted to ask the user to enter a number and then print all the numbers between 1 and the number the user entered? The number the user will enter isn't knowable at compile-time. So how might we go about solving this?

## While statements

The **while statement** (also called a **while loop**) is the simplest of the three loop types that C++ provides, and it has a definition very similar to that of an if-statement:

```
while (condition)
    statement;
```

A `while statement` is declared using the **while** keyword. When a while-statement is executed, the expression condition is evaluated. If the condition evaluates to `true`, the associated statement executes.

However, unlike an if-statement, once the statement has finished executing, control returns to the top of the while-statement and the process is repeated. This means a while-statement will keep looping as long as the condition continues to evaluate to `true`.

Let's take a look at a simple while loop that prints all the numbers from 1 to 10:

```
#include <iostream>

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

    std::cout << "done!\n";

    return 0;
}
```

This outputs:

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

Let's take a closer look at what this program is doing.

First, we define a variable named `count` and set it to `1`. The condition `count <= 10` is `true`, so the statement executes. In this case, our statement is a block, so all the statements in the block will execute. The first statement in the block prints `1` and a space, and the second increments `count` to 2. Control now returns back to the top of the while-statement, and the condition is evaluated again. `2 <= 10` evaluates to `true`, so the code block is executed again. The loop will repeatedly execute until `count` is `11`, at which point `11 <= 10` will evaluate to `false`, and the statement associated with the loop will be skipped. At this point, the loop is done.



While this program is a bit more code than typing all the numbers between 1 and 10, consider how easy it would be to modify the program to print all the numbers between 1 and 1000: all you'd need to do is change `count <= 10` to `count <= 1000`.

---

### While-statements that evaluate to false initially

Note that if the condition initially evaluates to `false`, the associated statement will not execute at all. Consider the following program:

```
#include <iostream>

int main()
{
    int count{ 15 };
    while (count <= 10)
    {
        std::cout << count << ' ';
        ++count;
    }

    std::cout << "done!\n";

    return 0;
}
```

The condition `15 <= 10` evaluates to `false`, so the associated statement is skipped. The program continues, and the only thing printed is `done!`.

---

### Infinite loops

On the other hand, if the expression always evaluates to `true`, the while loop will execute forever. This is called an **infinite loop**. Here is an example of an infinite loop:

```
#include <iostream>

int main()
{
    int count{ 1 };
    while (count <= 10) // this condition will never be false
    {
        std::cout << count << ' '; // so this line will repeatedly execute
    }

    std::cout << '\n'; // this line will never execute

    return 0; // this line will never execute
}
```

Because `count` is never incremented in this program, `count <= 10` will always be true. Consequently, the loop will never terminate, and the program will print `1 1 1 1 1`... forever.



---

### Intentional infinite loops

We can declare an intentional infinite loop like this:

```
while (true)
{
  // this loop will execute forever
}
```

The only way to exit an infinite loop is through a return-statement, a break-statement, an exit-statement, a goto-statement, an exception being thrown, or the user killing the program.

Here's a silly example demonstrating this:

```
#include <iostream>

int main()
{

    while (true) // infinite loop
    {
        std::cout << "Loop again (y/n)? ";
        char c{};
        std::cin >> c;

        if (c == 'n')
            return 0;
    }

    return 0;
}
```

This program will continuously loop until the user enters `n` as input, at which point the if-statement will evaluate to `true` and the associated `return 0;` will cause function `main()` to exit, terminating the program.

It is common to see this kind of loop in web server applications that run continuously and service web requests.

**Best practice**

Favor `while(true)` for intentional infinite loops.

## Loop variables and naming

A **loop variable** is a variable that is used to control how many times a loop executes. For example, given `while (count <= 10)`, `count` is a loop variable. While most loop variables have type `int`, you will occasionally see other types (e.g. `char`).

Loop variables are often given simple names, with `i`, `j`, and `k` being the most common.

**As an aside...**

The use of `i`, `j`, and `k` for loop variable names arose because these are the first three shortest names for integral variables in the Fortran programming language. The convention has persisted since.

However, if you want to know where in your program a loop variable is used, and you use the search function on `i`, `j`, or `k`, the search function will return half of the lines in your program! For this reason, some developers prefer loop variable names like `iii`, `jjj`, or `kkk`. Because these names are more unique, this makes searching for loop variables much easier, and helps them stand out as loop variables. An even better idea is to use "real" variable names, such as `count`, `index`, or a name that gives more detail about what you're counting (e.g. `userCount`).

The most common type of loop variable is called a **counter**, which is a loop variable that counts how many times a loop has executed. In the examples above, the variable `count` is a counter.

## Integral loop variables should be signed

Integral loop variables should almost always be signed, as unsigned integers can lead to unexpected issues. Consider the following code:

```
#include <iostream>

int main()
{
    unsigned int count{ 10 }; // note: unsigned

    // count from 10 down to 0
    while (count >= 0)
    {
        if (count == 0)
        {
            std::cout << "blastoff!";
        }
        else
        {
            std::cout << count << ' ';
        }
        --count;
    }

    std::cout << '\n';

    return 0;
}
```

Take a look at the above example and see if you can spot the error. It's not very obvious if you haven't seen this before.

It turns out, this program is an infinite loop. It starts out by printing `10 9 8 7 6 5 4 3 2 1 blastoff!` as desired, but then loop variable `count` overflows, and starts counting down from `4294967295` (assuming 32-bit integers). Why? Because the loop condition `count >= 0` will never be false! When count is `0`, `0 >= 0` is true. Then `--count` is executed, and count wraps around back to `4294967295`. And since `4294967295 >= 0` is `true`, the program continues. Because `count` is unsigned, it can never be negative, and because it can never be negative, the loop won't terminate.

**Best practice**

Integral loop variables should generally be a signed integral type.

---

## Doing something every N iterations



Each time a loop executes, it is called an **iteration**.

Often, we want to do something every 2nd, 3rd, or 4th iteration, such as print a newline. This can easily be done by using the remainder operator on our counter:

```
#include <iostream>

// Iterate through every number between 1 and 50
int main()
{
    int count{ 1 };
    while (count <= 50)
    {
        // print the number (pad numbers under 10 with a leading 0 for formatting purposes)
        if (count < 10)
        {
            std::cout << '0';
        }

        std::cout << count << ' ';

        // if the loop variable is divisible by 10, print a newline
        if (count % 10 == 0)
        {
            std::cout << '\n';
        }

        // increment the loop counter
        ++count;
    }

    return 0;
}
```

This program produces the result:

```
01 02 03 04 05 06 07 08 09 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

## Nested loops

It is also possible to nest loops inside of other loops. In the following example, the nested loop (which we're calling the inner loop) and the outer loop each have their own counters. Note that the loop expression for the inner loop makes use of the outer loop's counter as well!

```
#include <iostream>

int main()
{
    // outer loops between 1 and 5
    int outer{ 1 };
    while (outer <= 5)
    {
        // For each iteration of the outer loop, the code in the body of the loop executes once

        // inner loops between 1 and outer
        int inner{ 1 };
        while (inner <= outer)
        {
            std::cout << inner << ' ';
            ++inner;
        }

        // print a newline at the end of each row
        std::cout << '\n';
        ++outer;
    }

    return 0;
}
```

This program prints:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Nested loops tend to be hard for new programmers to understand, so don't be discouraged if you find this a bit confusing. For each iteration of the outer loop, the body of the outer loop will execute once. Because the outer loop body contains an inner loop, the inner loop is executed for each iteration of the outer loop.

Let's examine how this works in more detail.

First, we have an outer loop (with loop variable `outer`) that will loop 5 times (with `outer` having values `1`, `2`, `3`, `4`, and `5` successively).

On the first iteration of the outer loop, `outer` has value `1`, and then the outer loop body executes. Inside the body of the outer loop, we have another loop with loop variable `inner`. The inner loop iterates from `1` to `outer` (which has value `1`), so this inner loop will execute once, printing the value `1`. Then we print a newline, and increment `outer` to `2`.

On the second iteration of the outer loop, `outer` has value `2`, and then the outer loop body executes. Inside the body of the outer loop, `inner` iterates from `1` to `outer` (which now has value `2`), so this inner loop will execute twice, printing the values `1` and `2`. Then we print a newline, and increment `outer` to `3`.

This process continues, with the inner loop printing `1 2 3`, `1 2 3 4`, and `1 2 3 4 5` on successive passes. Eventually, `outer` is incremented to `6`, and because the outer loop condition (`outer <= 5`) is then false, the outer loop is finished. Then the program ends.

If you're still finding this confusing, stepping through this program in a debugger line-by-line and watching the values of `inner` and `outer` is a good way to get a better understanding of what's happening.

---

## Quiz time

### Question #1

In the above program, why is variable `inner` declared inside the while block instead of immediately following the declaration of `outer`?

Show Solution (javascript:void(0))

---

### Question #2

Write a program that prints out the letters a through z along with their ASCII codes. Use a loop variable of type `char`.

Show Hint (javascript:void(0))

Show Solution (javascript:void(0))

---

### Question #3

Invert the nested loops example so it prints the following:

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

Show Solution (javascript:void(0))

**Question #4**

Now make the numbers print like this:

```
        1
      2 1
    3 2 1
  4 3 2 1
5 4 3 2 1
```

Hint: Figure out how to make it print like this first:

```
X X X X 1
X X X 2 1
X X 3 2 1
X 4 3 2 1
5 4 3 2 1
```

Show Solution (javascript:void(0))


**Next lesson**
8.9 Do while statements

**Back to table of contents**

**Previous lesson**
8.7 Goto statements