5.5 — While statements

BY ALEX ON JUNE 22ND, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 13TH, 2018

The **while statement** is the simplest of the four loops that C++ provides, and it has a definition very similar to that of an *if statement*:

```
while (expression)
    statement;
```

A while statement is declared using the **while** keyword. When a while statement is executed, the expression is evaluated. If the expression evaluates to true (non-zero), the 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.

Let's take a look at a simple while loop. The following program prints all the numbers from 0 to 9:

```
#include <iostream>

int main()
{
    int count = 0;
    while (count < 10)
        {
        std::cout << count << " ";
        ++count;
        }
    std::cout << "done!";

    return 0;
}
```

This outputs:

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

Let's take a closer look at what this program is doing. First, count is initialized to 0. `0 < 10` evaluates to true, so the statement block executes. The first statement prints 0, and the second increments count to 1. Control then returns back to the top of the while statement. `1 < 10` evaluates to true, so the code block is executed again. The code block will repeatedly execute until count is 10, at which point `10 < 10` will evaluate to false, and the loop will exit.

It is possible that a while statement executes 0 times. Consider the following program:

```
#include <iostream>

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

    return 0;
}
```

The condition `15 < 10` immediately evaluates to false, so the while statement is skipped. The only thing this program prints 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 = 0;
    while (count < 10) // this condition will never be false
        std::cout << count << " "; // so this line will repeatedly 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 "0 0 0 0 0 ..." forever.

We can declare an intentional infinite loop like this:

```
while (1) // or 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.

Programs that run until the user decides to stop them sometimes intentionally use an infinite loop along with a return, break, or exit statement to terminate the loop. It is common to see this kind of loop in web server applications that run continuously and service web requests.

Loop variables

Often, we want a loop to execute a certain number of times. To do this, it is common to use a **loop variable**, often called a **counter**. A loop variable is an integer variable that is declared for the sole purpose of counting how many times a loop has executed. In the examples above, the variable *count* is a loop variable.

Loop variables are often given simple names, such as i, j, or k. However, naming variables i, j, or k has one major problem. 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 your program! Many words have an i, j, or k in them. Consequently, a better idea is to use iii, jjj, or kkk as your loop variable names. 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, or a name that gives more detail about what you're counting.

It is best practice to use signed integers for loop variables. Using unsigned integers can lead to unexpected issues. Consider the following code:

```
#include <iostream>

int main()
{
    unsigned int count = 10;

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

    return 0;
}
```

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

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 goes off the rails, and starts counting down from 4294967295. 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 overflows back to 4294967295. And since 4294967295 is >= 0, the program continues. Because count is unsigned, it can never be negative, and because it can never be negative, the loop won't terminate.

Rule: Always use signed integers for your loop variables.

Iteration

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

Because the loop body is typically a block, and because that block is entered and exited with each iteration, any variables declared inside the loop body are created and then destroyed with each iteration. In the following example, variable x will be created and destroyed 5 times:

```
#include <iostream>

int main()
{
    int count = 1;
    int sum = 0; // sum is declared up here because we need it later (beyond the loop)

    while (count <= 5) // iterate 5 times
    {
        int x; // x is created here with each iteration

        std::cout << "Enter integer #" << count << ':';
        std::cin >> x;

        sum += x;

        // increment the loop counter
        ++count;
    } // x is destroyed here with each iteration

    std::cout << "The sum of all numbers entered is: " << sum;

    return 0;
}
```

For fundamental variables, this is fine. For non-fundamental variables (such as structs and classes) this may cause performance issues. Consequently, you may want to consider defining non-fundamental variables before the loop. This is another one of the cases where you might declare a variable well before its first actual use.

Note that variable count is declared outside the loop. This is necessary because we need the value to persist across iterations (not be destroyed with each iteration).

Often, we want to do something every n iterations, such as print a newline. This can easily be done by using the modulus 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" << count << " ";
        else
            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 inner loop and outer loops each have their own counters. However, note that the loop expression for the inner loop makes use of the outer loop's counter as well!

```
#include <iostream>

// Loop between 1 and 5
int main()
{
    int outer = 1;
    while (outer <= 5)
    {
        // loop between 1 and outer
        int inner = 1;
        while (inner <= outer)
            std::cout << 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
```

Quiz

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

Hide Solution

Variable inner is declared inside the while block so that it is recreated (and reinitialized to 1) each time the outer loop executes. If variable inner were declared before the outer while loop, its value would never be reset to 1, or we'd have to do it with an assignment

statement. Furthermore, because variable inner is only used inside the outer while loop block, it makes sense to declare it there. Remember, declare your variables in the smallest scope possible!

2) Write a program that prints out the letters a through z along with their ASCII codes. Hint: to print characters as integers, you have to use a static_cast.

Hide Solution

```
#include <iostream>

int main()
{
    char mychar = 'a';
    while (mychar <= 'z')
    {
        std::cout << mychar  << " " << static_cast<int>(mychar) << "\n";
        ++mychar;
    }

    return 0;
}
```

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
```

Hide Solution

```
#include <iostream>

// Loop between 5 and 1
int main()
{
    int outer = 5;
    while (outer >= 1)
    {
        // loop between inner and 1
        int inner = outer;
        while (inner >= 1)
            std::cout << inner-- << " ";

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

    return 0;
}
```

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
```

Hide Solution

```cpp
// Thanks to Shiva for this solution
#include <iostream>

int main()
{
    // There are 5 rows, we can loop from 1 to 5
    int outer = 1;

    while (outer <= 5)
    {
        // Row elements appear in descending order, so start from 5 and loop through to 1
        int inner = 5;

        while (inner >= 1)
        {
            // The first number in any row is the same as the row number
            // So number should be printed only if it is <= the row number, space otherwise
            if (inner <= outer)
                std::cout << inner << " ";
            else
                std::cout << "  "; // extra spaces purely for formatting purpose

            --inner;
        }

        // A row has been printed, move to the next row
        std::cout << "\n";

        ++outer;
    }

    return 0;
}
```



5.6 -- Do while statements


Index


5.4 -- Goto statements

Share this: