Chapter 7

Repetition

Decision-making via if statements is the first of the really powerful programming constructs needed to write useful programs. The second programming construct we need is repetition. Humans are bad at doing things over and over again; we get bored and tired and eventually start making mistakes. Computers, however, never get tired or bored and can perform the same tasks hours on end without every making even the slightest mistake. In this section we'll study how to perform repetition in C++, also called looping, using the while loop statement.

The while Loop

One means of performing repetition in C++ is using the while loop. The while loop examines a condition, much like an if statement does, executing a set of statements repeatedly while the condition is true. The process of executing statements multiple times is called *iteration*.

The general form of the while loop looks like this:

while (condition)
{
  statement₁;
  statement₂;
  statementₙ;
}

The statements inside the block are called the *loop body*.

There are two necessary ingredients of a while loop: 1) a condition to be tested; and 2) a statement that causes the condition to eventually become false. The second condition is important because if the condition controlling the loop never becomes false, the loop will run forever (or at least until the computer runs out of resources). This situation is called an *infinite loop*.

Let's look at two simple examples of while loops. The first example displays the phrase “Hello, World!” ten times. Here is the code:

```
int number = 1;
while (number <= 10)
{
```

```
    cout << "Hello, World!" << endl;
    ++number;
}
```

This example is trivial, but it points out the important components necessary for a successful while loop. The variable number is called the l*oop-control* variable. The value of this variable controls the iterations of the loop. Generally, the loop-control variable is assigned a value outside of the loop. Then the loop-control variable is tested in the condition. Because we want to display “Hello, World!” ten times, the value of number must be less than or equal to 10 for each iteration of the loop. It is initialized to 1 so that the condition will be true and the loop body entered.

After the variable is initialized, the while loop condition is tested. Since the value of number currently is 1, and 1 is less than or equal to 10, the condition is true and the loop body is entered. The phrase “Hello, World!” is displayed to the console and in the last line, the value of number is incremented to 2.

At this point, the program moves back to the loop condition, which is tested again. Since 2 is less than equal to 10, the condition is true and another iteration is performed. This continues until the value of number becomes 11. When the condition is tested, the value is false because 11 is not less than or equal to 10, it is greater, so control falls to the statement after the closing curly brace.

This next example also doesn't do anything very important but it again demonstrates the common components of while loops. We want to write a program that sums the integers 1 through 10. Our loop-control variable will again be named number. It is intialized to the value 1 and is incremented by 1 each iteration. We also need to initialize a second variable (sum) outside of the loop to hold the sum of the numbers as we work through them.

Here is the program:

```
// WhileLoop.cpp : main project file.

#include "stdafx.h"
using namespace System;

#include <iostream>
using namespace std;

int main(array<System::String ^> ^args)
{
      int number, sum;
      number = 1;
      sum = 0;
      while (number <= 10)
      {
            sum += number;
            ++number;
      }
      cout << "The sum of 1 - 10 is " << sum << endl;
      Console::ReadKey();
```

```
    return 0;
}
```

The output from this program is:

The sum of 1 – 10 is 55

This program also demonstrates how the loop-control variable can be used for something other than just controlling the iterations of the loop, since the values of number are used to compute the sum.

Reading Data From Files

There are two major types of while loops in most C++ programs – counter-controlled loops and sentinel-controlled loops. The examples discussed above were both counter-controlled loops because the loop's condition was based on reaching a certain count, either in iterations of the loop in the “Hello, World!” example, or on reaching a certain number in the compute the sum example.

Sentinel-controlled loops, on the other hand, are controlled by a specific situation, such as reaching a particular value, or when a resource runs out, such as reaching the end of a file. This value is called a sentinel value and sentinel-controlled loops stop when this sentinel value is obtained.

A classic example of a sentinel-controlled loop is a loop that reads the contents of a text file. Reading data files are usually controlled by sentinel values because we don't know when we are going to reach the end of the file. A specific value can't be entered, such as the number of lines to read, unless we know in advance how many lines are in the file, and this is rarely the case. We'll discuss how to develop sentinel-controlled loops by reading the contents of a text file.

Working With Text Files

To read data from a text file, we need to use some special functions and objects that are part of an external library, the fstream library. This library, called the “file stream” library, allows us to open text files as a stream, much like we use the iostream library to work with the console and the keyboard.

The first step is to include a preprocessor directive for the fstream library:

```
#include <fstream>
```

This directive is placed in the same group as the iostream library directive.

The next step is to create a file object that represents the file we want to read. Because we are reading data from a file, the data type for the file object is ifstream. If we were writing data to a file, we would use the ofstream (output) data type. The line of code to create a file object is:

```
ifstream myFile;
```

We also need a string variable to store the data as its read:

```
string line;
```

Then we need to associate our file object with a physical file. We do this with the open() function, passing in the complete path to our file as the argument to the function. The function is called directly from the file object, like this:

```
myFile.open("c:\\data\\example.txt");
```

The argument to the open() function must be the complete path to the file unless the file is located in the same directory as the program, in which case we can just use the name of the file:

```
myFile.open("example.txt");
```

When the file is open, we're ready to read its contents. This requires a while loop that will run until there is no more data in the file. This is an example of a sentinel-controlled loop and the sentinel we use for reading data is a true/false value given by the fstream function eof(). This function returns false while there is more data in the file to read and returns true

when there is no more data. Here is the code:

```
while (!myFile.eof())
{
```

Note that we use the ! (not) operator here because the eof() function returns false when there is data to read so we have to negate that false value in order for the loop to continue running.

The first line in the body of the loop reads a line of text and stores it in a variable:

```
getline(myFile, line);
```

The next and last line displays the text read from the file:

```
   cout << line << endl;
}
```

Finally, any time we open a file we should close it and there is a close() function for just that purpose:

```
myFile.close();
```

Here's the complete program:

```
// textFile.cpp : main project file.
#include "stdafx.h"
using namespace System;
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main(array<System::String ^> ^args)
{
     ifstream myFile;
     string line;
     myFile.open("example.txt");
     while (!myFile.eof())
     {
          getline(myFile, line);
          cout << line << endl;
     }
     myFile.close();
     Console::ReadKey();
     return 0;
```

```
}
```

This example demonstrates how to work with a text file, but it doesn't really solve a problem. Let's write a program that reads a set of daily high temperatures from a file and computes the average high temperature. This is an interesting problem because we will have to keep track of how many temperatures we've read in order to compute the average. First, create a text file with a list of temperatures. Try not to pay attention to how many temperatures you put in the file but include at least fifteen to twenty temperatures.

Here is the program to compute the average temperature:

```
// readTemps.cpp : main project file.

#include "stdafx.h"

using namespace System;
#include <iostream>
#include <fstream>
using namespace std;

int main(array<System::String ^> ^args)
{

      ifstream temps;
      int temp, totalTemps, numTemps;
      double average;
      totalTemps = numTemps = 0;
      temps.open("c:\\data\\temperatures.txt");
      while (!temps.eof())
      {
            temps >> (int)temp; // cast to int
            totalTemps += temp;
            ++numTemps;
      }



      average = totalTemps / numTemps;
      cout << "The average temperature is: "<< average
           << endl;
      Console::ReadKey();
     return 0;
}
```

There are some new things in this program that you haven't seen before. The line:

```
totalTemps = numTemps = 0;
```

initializes both variables to 0 in one statement. We could have done the same thing in two lines but this is a simple shortcut that ends up saving more time and space than you first realize.

The argument to the open() function specifies a path that is not in the current working directory. To do this we have to use two backslashes instead of one when specifying a folder name or the file name. If we didn't do this, C++ will try to read the backslash and next character as a single character, such as \n, leading to confusion. Using two backslashes remedies this problem.

The last new piece of programming is in the line where a temperature is read from the file:

```
temps >> (int)temp; // cast to int
```

When data is stored in a text file, it is stored as a string. So each temperature we read into our program is being read as a string, not as an integer. If we want to perform arithmetic on the temperature once its read into our program, we have to somehow convert it from a string to an integer. We accomplish this with what is know as a *type cast*. To cast one type to another type, you place the type you want to convert to in front of the wrong type value, surrounded in parentheses, and when the value "passes through" the type cast, it is converted to the new data type. So when we write:

```
temps >> (int) temp;
```

we are saying to read the next data item in the file, convert it to integer, and store it in the variable temp.

Another Sentinel-Controlled Loop Example

Using a loop to read data from a file is one situation where you need a sentinel-controlled loop; getting a set of data from the user is another such situation. This scenario occurs when you want to have the user enter data but you don't know in advance how many items the user is going to enter. The program has to recognize a sentinel value that indicates there is no more input and the program can continue with its processing.

Here's an example. We want a program that averages a set of test scores but we don't know how many test scores will be entered. In this type of program, a numerical value will have to do as the sentinel since the program will be reading numbers from the keyboard. For test scores, a good sentinel value is -1, since the lowest test score possible is a 0 (unless it's a really, really hard test).

A program that solves this problem follows:

```
// AverageGrade.cpp : main project file.

#include "stdafx.h"

using namespace System;
#include <iostream>
using namespace std;

int main(array<System::String ^> ^args)
{
```

```
      int testScore, total, count;
      double average;
      total = 0;
      count = 0;
      cout << "Enter a test score (-1 to quit): ";
      cin >> testScore;
      while (testScore != -1)
      {
            total += testScore;
            ++count;
            cout << "Enter a test score (-1 to quit): ";
            cin >> testScore;
      }
      average = total / count;
      cout << endl <<  "The average test score is: "
           << average << endl;
      Console::ReadKey();
      return 0;
}
```

Here is a sample run of the program:

Enter a test score (-1 to quit): 87

Enter a test score (-1 to quit): 98

Enter a test score (-1 to quit): 76

Enter a test score (-1 to quit): 88

Enter a test score (-1 to quit): 92

Enter a test score (-1 to quit): 83

Enter a test score(-1 to quit): -1

The average test score is: 87

The program prompts the user to enter a test score continually until a -1 is entered. Entering -1 makes the loop condition false and the loop is exited. Notice that the first prompt for a test score has to be outside the loop in order for the loop body to be executed properly. This is sometimes called “priming the pump.” If you don't prime the pump and put the prompt inside the loop, like this:

```
while (testScore != -1)
{
      cout << "Enter a test score (-1 to quit): ";
      cin >> testScore;
      total += testScore;
      ++count;
}
```

The sentinel value will be included in the total, causing the average to be miscalculated.

for Loops

The second construct used for repetition is the for loop. The for loop is primarily used in situations where you want to iterate over a set of statements a set number of times, as opposed to a while loop, which is used primarily when you want to iterate over a set of statements until a specific condition occurs.

Working With for Loops

The general form of the for loop looks like this:

for(control-var declaration; condition; control-variable modification)
{
statement1;
statement2;
statementn;
}

A concrete example will explain how the for loop works. Here's a simple program that displays the numbers 1 through 10:

```
// ForLoop.cpp : main project file.

#include "stdafx.h"
using namespace System;

#include <iostream>
using namespace std;

int main(array<System::String ^> ^args)
{
     for(int i = 1; i <= 10; ++i)
     {
          cout << i << " ";
     }
     Console::ReadKey();
     return 0;
}
```

The output from this program is:

1 2 3 4 5 6 7 8 9 10

The first thing that happens is the loop control variable, i, is declared and initialized to 1. Next, the condition is tested. The condition for this loop says to iterate through the loop while the value of i is less than or equal to 10. Since i was just initialized to 1, the condition is true and the loop body is entered.

As with a while loop, the loop body of a for loop is a block. The body for this particular loop just has one statement, so the value of i is written to the console. Since this is the first and last statement in the loop body, the loop control modification step occurs next, with the value of i being incremented by 1.

Now the value of i is 2 and the condition is tested again. The condition is true so the value of i, 2, is written to the console and i is incremented by 1 again. The for loop continues in this fashion until the value of i is incremented to 11. Then the condition is tested, the result is false, since 11 is not less than or equal to 10, and the loop body is exited.

More for Loop Examples

One use of the for loop is to determine the value of an investment at the end of some time interval. Say you want to invest $10,000 at 2% annual interest for 10 years. What will the value of that investment be at the end of that period?

The algorithm for solving this problem involves multiplying the principal by the rate and adding that value back to the principal 10 times. We can use a for loop to solve this problem. Here is the code:

```
// ForLoop.cpp : main project file.

#include "stdafx.h"
using namespace System;

#include <iostream>
using namespace std;

int main(array<System::String ^> ^args)
{
      double rate = 1.02;
      double principal = 10000;
      for(int i = 1; i <=10; ++i)
      {
            principal *= rate;
      }
      cout << "After 10 years, $10,000 invested at 2%"
           << " annually will be worth: $"
           << principal << endl;
      Console::ReadKey();
      return 0;
}
```

Let's look at an example that doesn't involve numbers. If we want to encrypt a sentence using a simple encryption algorithm, we might want to know how many vowels are in a sentence. To do this we have to examine each character in the string and compare it with the list of vowels. We can use a for loop to control our movement through the string.

To write this program, we need to use a couple of string functions we haven't seen before. The first of these is the substr() function. Called the “substring function,” this function returns a substring from a string. The arguments to the function are the character position you want to start the substring, and the number of characters you want for the substring. Here's an example:

```
string word = "Hello";
cout << word.substr(1,3);  // returns "ell"
```

The characters in a string are numbered beginning at position 0. So, the letter “H” in “Hello” is at position 0. The “e” is at position 1. So, if we write word.substr(1,3), we are saying we want to return 3 characters from the string word starting at position 1.

The second string function we need for this example is the compare() function. The usual relational operators don't work with strings in C++, so we have to use some other means of making comparisons between strings. The compare() function takes a string as an argument and compares it to the string object the function is called from. The function returns either 0, 1, or -1, depending on the value of the string argument. For example, if we write:

```
string fruit = "banana";
cout << word.compare("apple");
```

the value 1 is returned because “banana” is greater than “apple”. If we compare “pear” to the variable fruit we get the value -1, because “apple” is less than “pear.” If we compare “banana” to “banana” we get 0, since they are equal.

Now we can write the algorithm for the vowel-counting program. Using a for loop, we examine each letter in the string and compare it to a list of vowels. If the letter is a vowel, increment a counter variable, and otherwise do nothing.

Here is the program:

```
// ForLoop.cpp : main project file.

#include "stdafx.h"
using namespace System;

#include <iostream>
#include <string>
using namespace std;


int main(array<System::String ^> ^args)
{
      string sentence = "Now is the time for all good"
       sentence += " people to come to the aid of "
      sentence += " their party.";
       int vowelCount = 0;
       string letter = "";
       int strLen = sentence.size();
       for(int i = 0; i < strLen; ++i)
       {
             letter = sentence.substr(i,1);
             if ((letter.compare("a") == 0) ||
                   (letter.compare("e") == 0) ||
                   (letter.compare("i") == 0) ||
                   (letter.compare("o") == 0) ||
                   (letter.compare("u") == 0))
                  ++vowelCount;
```

```
      }
      cout << "Number of vowels in the sentence is: "
           << vowelCount << endl;
      Console::ReadKey();
      return 0;
}
```

The output from this program is:

Number of vowels in the sentence is: 23

Nested for Loops

There are many programming problems where the data is structured in a table format. For example, if an instructor wants to compute the average grade for each student in his or her class, the data will be structured something like this:

	Test 1	Test 2	Test 3	Test 4
Student 1	84	78	93	98
Student 2	78	82	84	89
Student 3	93	89	98	100

The program will start with the first student, moving across the columns creating a sum from each test grade, then dividing by the number of grades to display the average, and then moving to the next student. This program will need two for loops, an outer for loop to move from student to student, and an inner for loop to move across the columns of grades. Two for loops working in this manner are called nested for loops.

We can't use nested for loops to solve the problem above yet because we haven't learned how to store data in arrays yet (we will get to arrays very soon). However, we can demonstrate how to use nested for loops for a simpler problem – displaying multiplication tables.

First, let's look at the layout for the multiplication table for 1 x 1 through 5 x 5:

1 x 1 = 1	1 x 2 = 2	1 x 3 = 3	1 x 4 = 4	1 x 5 = 5
2 x 1 = 2	2 x 2 = 4	2 x 3 = 6	2 x 4 = 8	2 x 5 = 10
3 x 1 = 3	3 x 2 = 6	3 x 3 = 9	3 x 4 = 12	3 x 5 = 15
4 x 1 = 4	4 x 2 = 8	4 x 3 = 12	4 x 4 = 16	4 x 5 = 20
5 x 1 = 5	5 x 2 = 10	5 x 3 = 15	5 x 4 = 20	5 x 5 = 25

We can clearly see that that the rows can be controlled by one for loop (the inner loop) and the columns by another for loop (the outer loop). Now let's look at the program:

```
// ForLoop.cpp : main project file.

#include "stdafx.h"
using namespace System;

#include <iostream>
#include <string>
using namespace std;

int main(array<System::String ^> ^args)
{
    int product;
     for(int leftOp = 1; leftOp <= 5; ++leftOp)
     {
          for(int rightOp = 1; rightOp <= 5; ++ rightOp)
          {
               product = leftOp * rightOp;
               cout << leftOp << " x " << rightOp << " = "
                    << product << "\t";
          }
          cout << endl;
     }
     Console::ReadKey();
    return 0;
}
```

Here is the output from the program:

```
1 x 1 = 1    1 x 2 = 2    1 x 3 = 3    1 x 4 = 4    1 x 5 = 5
2 x 1 = 2    2 x 2 = 4    2 x 3 = 6    2 x 4 = 8    2 x 5 = 10
3 x 1 = 3    3 x 2 = 6    3 x 3 = 9    3 x 4 = 12   3 x 5 = 15
4 x 1 = 4    4 x 2 = 8    4 x 3 = 12   4 x 4 = 16   4 x 5 = 20
5 x 1 = 5    5 x 2 = 10   5 x 3 = 15   5 x 4 = 20   5 x 5 = 25
```

The spacing in each row is controlled by the tab character (\t).

In all our previous for loops we used a single-letter variables for the loop control variable. This is acceptable since it is easy for someone reading the code to understand how the variable is being used. For this program, however, we decided to use more meaningful names for the loop control variables, leftOp and rightOp, since it helps the reader more easily understand the program.

Exercises

1. Write a program that asks for the user to enter a number and then calculates the sum of the squares from 1 to that number. For example, if the user enters 4, the sum of the squares is 30 (1 + 4 + 9 + 16). The output should include the number entered by the user and the result.
2. Write a program that reads a set of numbers from a file and displays the number of even numbers and the number of odd numbers in the file. Hint: use the % operator to determine if a number is even or odd.
3. Write a program that computes the even numbers in the range from 16 to 36. Use a for loop in your program.

4. Using a while loop and a for loop, write a program that reads a set of numbers from a file and prints a string of asterisks for each number. The output should look like this:

```
5 -    *****
10 -   **********
7 -    *******
```