SESSION 8

Flow Control Commands

Session Checklist

- ✔ Controlling the flow through the program
- ✔ Executing a group of statements repetitively
- ✔ Avoiding “infinite loops”

30 Min.
To Go

The programs that have appeared so far in this book have been very simple. Each took a single set of values as input, output the result, and terminated. This is similar to instructing our computerized mechanic in how to take off a single lug nut with no option to loop around to the other lug nuts on the wheel or other wheels on the car.

What is missing in our programs is any form of flow control. We have no ability to make tests of any sort, much less make decisions based on those tests.

This chapter examines the different C++ flow control commands.

The Branch Command

The simplest form of flow control is the branch statement. This instruction enables the computer to decide which path to take through C++ instructions based on some logic condition. In C++, the branch statement is implemented using the `if` statement:

```
if (m > n)
{
    // instructions to be executed if
    // m is greater than n
}
else
{
    // ...instructions to be executed if not
}
```

First, the condition `m > n` is evaluated. If the result is true, then control passes to the instructions following the `{`. If `m` is not greater than `n`, then control passes to the brace immediately following the `else` clause.

The `else` clause is optional. If it is not present, C++ acts as if it is present, but empty.

Note

Actually the braces are optional if there is only one statement to be executed as part of the `while` loop; however, it is too easy to make a mistake that the C++ compiler can't catch without the braces as a guide marker. It is much safer to always include the braces. If your friends try to entice you into not using braces, just say "NO."

The following program demonstrates the `if` statement.

```
// BranchDemo - input two numbers. Go down one path of the
//              program if the first argument is greater than
//              the second or the other path if not
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{

    // input the first argument...
    int nArg1;
    cout << “Enter arg1: “;
    cin  > nArg1;

    // ...and the second
    int nArg2;
    cout << “Enter arg2: “;
    cin  > nArg2;

    // now decide what to do:
    if (nArg1 > nArg2)
    {
        cout << “argument 1 is greater than argument 2\n”;
    }
    else
    {
        cout << “argument 1 is not greater than argument 2\n”;
    }

    return 0;
}
```

Here the program reads to integers from the keyboard and branches accordingly. This program generates the following typical results:

```
Enter arg1: 10
Enter arg2: 8
argument 1 is greater than argument 2
```

Looping Commands

Branch statements enable you to control flow of a program’s execution down one path of a program or another. This is the C++ equivalent of allowing the computerized mechanic to decide whether to use a wrench or a screwdriver depending on the problem. This still doesn’t get us to the point where the mechanic can turn the wrench more than once, remove more than one lug nut, or handle more than one wheel on the car. For that, we need looping statements.

The while loop

The simplest form of looping statement is the `while` loop. The form of a `while` loop is as follows:

```
while(condition)
{
    // ...repeatedly executed as long as condition is true
}
```

The condition is tested. If it is true, then the statements within the braces are executed. Upon encountering the closed brace, control returns to the beginning and the process starts over. The net effect is that the C++ code between the braces is executed repeatedly as long as the condition is true.

The condition is only checked at the beginning of the loop. Even if the condition ceases to be true, control does not exit the loop until the beginning of the loop.

If the condition is true the first time, then what will make it false in the future? Consider the following example program:

```
// WhileDemo - input a loop count. Loop while
//              outputting astring nArg number of times.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{

    // input the loop count
    int nLoopCount;
    cout << “Enter nLoopCount: “;
    cin  > nLoopCount;

    // now loop that many times
    while (nLoopCount > 0)
    {
        nLoopCount = nLoopCount - 1;
        cout << “Only “ << nLoopCount << “ loops to go\n”;
    }
    return 0;
}
```

WhileDemo begins by retrieving a loop count from the user that it stores in the variable nLoopCount. That done, the program continues by testing nLoopCount. If nLoopCount is greater than 0, the program decrements nLoopCount by 1 and outputs the result to the display. The program then returns to the top of the loop to test whether nLoopCount is still positive.

When executed, the program WhileDemo outputs the following results :

```
Enter nLoopCount: 5
Only 4 loops to go
Only 3 loops to go
Only 2 loops to go
Only 1 loops to go
Only 0 loops to go
```

When I enter a loop count of 5, the program loops five times, each time outputting the count down value.

If the user enters a negative loop count, the program skips the loop entirely. Because the condition is never true, control never enters the loop. In addition, if the user enters a very large number, the program loops for a long time before completing.

Note

A separate, seldom used version of the while loop, known as the do while, is identical to the while loop except that the condition isn't tested until the bottom of the loop:

```
do
{
    // ...the inside of the loop
} while (condition);
```

20 Min.
To Go

Using the autodecrement feature

The program decrements the loop count by using the assignment and subtraction statements. A more compact statement would use the autodecrement feature.

The following loop is slightly simpler than the one above.

```
    while (nLoopCount > 0)
    {
        nLoopCount--;
        cout << “Only “ << nLoopCount << “ loops to go\n”;
    }
```

The logic in this version is the same as the original — the only difference is the way that `nLoopCount` is decremented.

Because autodecrement both decrements its argument and returns its value, the decrement operation can actually be combined with either of the other statements. For example, the following version is the smallest loop yet.

```
    while (nLoopCount-- > 0)
    {
        cout << “Only “ << nLoopCount << “ loops to go\n”;
    }
```

Tip

This is the version that most C++ programmers would use.

This is where the difference between the predecrement and postdecrement operations arises.

Note

Both `nLoopCount--` **and** `--nLoopCount` **decrement** `nLoopCount`**; however, the former returns the value of** `nLoopCount` **before being decremented and the latter after.**

Do you want the loop to be executed when the user enters a loop count of 1? If you use the predecrement version, the value of `--nLoopCount` is 0 and the body of the loop is never entered. With the postdecrement version, the value of `nLoopCount--` is 1 and control enters the loop.

The dreaded infinite loop

Suppose that because of a coding error, the programmer forgot to decrement the variable `nLoopCount`, as in the loop example below. The result would be a loop counter that never changed. The test condition would either be always false or always true.

```
    while (nLoopCount > 0)
    {
        cout << “Only “ << nLoopCount << “ loops to go\n”;
    }
```

Because the value of nLoopCount never changes, the program executes in a never-ending or infinite loop. An execution path that continues forever is known as an *infinite loop*. An infinite loop occurs when the condition that would otherwise terminate the loop cannot occur — usually due to some coding error.

There are many ways of creating an infinite loop, most of which are much more difficult to spot than this one.

The for loop

A second form of loop is the for loop. The for loop has this format:

```
for (initialization; conditional; increment)
{
    // ...body of the loop
}
```

Execution of the for loop begins with the initialization clause. The initialization clause got this name because this is normally where counting variables are initialized. The initialization clause is only executed once, when the for loop is first encountered.

Execution continues to the conditional clause. In similar fashion to the while loop, the for loop continues to execute as long as the conditional clause is true.

After completing execution of the code in the body of the loop, control passes to the increment clause before returning to check the conditional clause, thereby repeating the process. The increment clause normally houses the autoincrement or autodecrement statements used to update the counting variables.

All three clauses are optional. If the initialization or increment clauses are missing, C++ ignores them. If the conditional clause is missing, C++ performs the for loop forever (or until something else breaks control outside of the loop).

The for loop is better understood by example. The following ForDemo program is nothing more than the WhileDemo program converted to use the for loop construct.

```
// ForDemo - input a loop count. Loop while
//           outputting a string nArg number of times.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // input the loop count
    int nLoopCount;
    cout << “Enter nLoopCount: “;
    cin  > nLoopCount;

// count up to the loop count limit
for (int i = 1; i <= nLoopCount; i++)
{
    cout << “We’ve finished “ << i << “ loops\n”;
}
```

This version loops the same as before; however, instead of modifying the value of `nLoopCount`, this version uses a counter variable.

Control begins by declaring a variable `i` and initializing it to t1. The `for` loop then checks the variable `i` to make sure that it is less than or equal to the value of `nLoopCount`. If this is true, the program executes the output statement, increments `i`, and starts over.

The `for` loop is also convenient when you need to count from 0 up to the loop count value rather than from the loop count down to 0. This is implemented by a simple change to the `for` loop:

```
// ForDemo - input a loop count. Loop while
//           outputting a string nArg number of times.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{

    // input the loop count
    int nLoopCount;
    cout << “Enter nLoopCount: “;
    cin  > nLoopCount;

    // count up to the loop count limit
    for (int i = 1; i <= nLoopCount; i++)
    {
        cout << “We’ve finished “ << i << “ loops\n”;
    }
    return 0;
}
```

Rather than begin with the loop count, this version starts with 1 and “loops up” to the value entered by the user.

Note

The use of the variable `i` for `for` loop increments is historical (stemming from the early days of the Fortran programming language). It is for this reason that such loop variables don’t follow the standard naming convention.

Tip

When declared within the initialization portion of the `for` loop, the index variable is only known within the `for` loop itself. Nerdy C++ programmers say that the scope of the variable is the `for` loop. In the example above, the variable `i` is not accessible from the `return` statement because that statement is not in the loop.

Special loop controls

It can happen that the condition for terminating the loop occurs neither at the beginning nor at the end of the loop. Consider the following program, which accumulates a number of values entered by the user. The loop terminates when the user enters a negative number.

The challenge with this problem is that the program can’t exit the loop until after the user has entered a value, yet it has to exit the loop before the value is added to the sum.

For these cases, C++ defines the `break` command. When encountered, the `break` causes control to exit the current loop immediately; that is, control passes from the `break` statement to the statement immediately following the closed brace.

The format of the `break` commands is as follows:

```
while(condition) // break works equally well in for loop
{
    if (some other condition)
    {
        break;   // exit the loop
    }
}                // control passes here when the
                 // program encounters the break
```

Armed with this new `break` command, my solution to the accumulator problem appears as the program BreakDemo.

```
// BreakDemo - input a series of numbers.
//             Continue to accumulate the sum
//             of these numbers until the user
//             enters a 0.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{

    // input the loop count
    int nAccumulator = 0;
    cout << “This program sums values entered “
         << “by the user\n”;
    cout << “Terminate the loop by entering “
         << “a negative number\n”;

    // loop “forever”
    for(;;)
    {
        // fetch another number
        int nValue = 0;
        cout << “Enter next number: “;
        cin  > nValue;

        // if it’s negative...
        if (nValue < 0)
        {
            // ...then exit
            break;
        }

        // ...otherwise add the number to the accumulator
        nAccumulator = nAccumulator + nValue;
    }

    // now that we’ve exited the loop
    // output the accumulated result
    cout << “\nThe total is “
         << nAccumulator
```

```
                << "\n";

            return 0;
    }
```

After explaining the rules to the user (entering negative number to terminate, and so forth), the program enters an infinite `for` loop.

Tip

A `for` loop with no condition loops forever.

Note

The loop is not truly infinite because it contains a `break` internally; however, the loop command is still referred to as infinite because the break condition doesn't exist in the command itself.

Once in the loop, BreakDemo retrieves a number from the keyboard. Only after the program has read a number can it test to determine whether the number read matches the exit criteria. If the input number is negative, control passes to the `break`, causing the program to exit the loop. If the input number is not negative, control skips over the `break` command to the expression that sums the new value to the accumulator.

Once the program exits the loop, it outputs the accumulated value and exits. Here's the output of a sample session:

```
This program sums values entered by the user
Terminate the loop by entering a negative number
Enter next number: 1
Enter next number: 2
Enter next number: 3
Enter next number: 4
Enter next number: -1

The total is 10
```

Tip

When performing an operation on a variable repeatedly in a loop, make sure that the variable was initialized properly before entering the loop. In this case, the program zeros `nAccumulator` before entering the loop where `nValue` is added to it.

Nested Control Commands

10 Min. To Go

The three loop programs in this chapter are the moral equivalent of instructing the mechanic in how to remove a lug nut: continue to turn the wrench until the nut falls off. What about instructing the mechanic to continue removing lug nuts until the wheel falls off? For this we need to implement nested loops.

A loop command within another loop command is known as a **nested loop**.

As an example, let's modify the BreakDemo program into a program that accumulates any number of sequences. In this NestedDemo program, the inner loop sums numbers entered from the keyboard until the user enters a negative number. The outer loop continues accumulating sequences until the sum is 0.

```
// NestedDemo - input a series of numbers.
//              Continue to accumulate the sum
//              of these numbers until the user
//              enters a 0. Repeat the process
//              until the sum is 0.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // the outer loop
    cout << "This program sums multiple series\n"
         << "of numbers. Terminate each sequence\n"
         << "by entering a negative number.\n"
         << "Terminate the series by entering two\n"
         << "negative numbers in a row\n";

    // continue to accumulate sequences
    int nAccumulator;
    do
    {
        // start entering the next sequence
        // of numbers
        nAccumulator = 0;
        cout << "\nEnter next sequence\n";

        // loop forever
        for(;;)
        {
            // fetch another number
            int nValue = 0;
            cout << “Enter next number: “;
            cin  > nValue;

            // if it’s negative...
            if (nValue < 0)
            {
                // ...then exit
                break;
            }

            // ...otherwise add the number to the accumulator
            nAccumulator = nAccumulator + nValue;
        }

        // output the accumulated result...
        cout << “\nThe total is “
            << nAccumulator
            << “\n”;

        // ...and start over with a new sequence
        // if the accumulated sequence was not zero
    } while (nAccumulator != 0);
    cout << “Program terminating\n”;
    return 0;
}
```

Can We switch *to a Different Subject?*

One last control statement is useful in a limited number of cases. The `switch` statement is like a compound `if` statement in that it includes a number of different possibilities rather than a single test:

```
switch(expression)
{
    case c1:
        // go here if the expression == c1
        break;
    case c2:
        // go here if expression == c2
        break;
    else
        // go here if there is no match
}
```

The value of expression must be an integer (`int`, `long`, or `char`). The case values `c1`, `c2`, and `c3` must be constants. When the `switch` statement is encountered, the expression is evaluated and compared to the various case constants. Control branches to the case that matches. If none of the cases match, control passes to the `else` clause.

The `break` statements are necessary to exit the `switch` command. Without the `break` statements, control "falls through" from one case to the next.

Review

The simple `if` statement enables the programmer to send the flow of control down one path or another based on the value of an expression. The looping commands add the capability to execute a block of code repeatedly until an expression is false. Finally, the `break` commands provide an extra level of control by allowing program control to exit a loop at any point.

- The `if` statement evaluates an expression. If the expression is not 0 (that is, it is `true`), control passes to the block immediately following the `if` clause. If not, control passes to the block of code following the `else` clause. If there is no `else` clause, control passes directly to the statement following the `if`.
- The looping commands, `while`, `do while`, and `for` execute a block of code repetitively until a condition is no longer true.
- The `break` statement allows control to pass out of a loop prematurely.

In Session 9, we look at ways to simplify C++ programs by using functions.

QUIZ YOURSELF

1. Is it an error to leave the `else` off of an `if` command? What happens? (See “The Branch Command.”)
2. What are the three types of loop commands? (See “Looping Commands.”)
3. What is an infinite loop? (See “The Dreaded Infinite Loop.”)
4. What are the three “clauses” that make up a `for` loop? (See “The for Loop.”)