# Laboratory 5:
## *Implementing Loops and Loop Control Strategies*

**Overview:** C++ has three control structures that are designed exclusively for iteration: the ***while***, ***for*** and ***do*** statements. In today's lab, you will gain some practice using these repetition structures while experimenting with various strategies for controlling them. You will also see how *nested* looping structures operate.

**Objectives:** Before you leave lab today, your goals should include: understanding how all three repetition structures work, what the various loop control strategies are, and their relative strengths and weaknesses. You should feel comfortable with the logic behind nested loops. In addition, you should leave with an even better appreciation for the use of the Borland C++ debugging tools.

### 1. Setting up for the lab

Please perform the following activities as you wait for class to formally begin:

a. Boot up the machine and log on to the CS Department server, as described in Lab 1.

b. Put your lab diskette in the *A:* drive and make a subdirectory *LAB05* in the *LABS* directory.

c. Download the files from the class Web site to your new directory. The URL is: http://s9000.furman.edu/chealy/cs11/lab05.

### 2. Counter control

From the lecture and the reading you recall that C++ offers three mechanisms for repeating tasks—***while***, ***for*** and ***do***. These structures have fundamentally different properties that we're about to explore. Once we choose what kind of loop we want, we have to choose the *control strategy* for the loop—general event control or one of its special cases: counter control, interactive control or sentinel control.

In the case of the ***for*** structure, the choice of control strategy is simple. The whole point of the ***for*** loop is the automation of the counter control strategy.

*Using the **for** loop*

a. Start up Borland C++ and open the file *sum1.cpp*. Read through the code thoroughly to try to understand what it is going to do. Try to make sense of the ***for*** statement in particular. What role is played by each component—the *initialization expression*, the *test expression*, the *increment expression* and the *loop body*? If you're unsure, don't hesitate to ask me.

b. The goal is to trace the values assigned to key variables as the loop is processing. We will do this by tracing the values of expressions in the Watch window as we step through the program. Enter the following variables into the Watch window (using the technique you learned in the previous lab):

| | |
|---|---|
| *N* | (the number of values expected) |
| *number* | (the current value read from input) |
| *i* | (the counter) |
| *sum* | (the accumulating total) |

c. Compile (Build) the program and step through it line by line using the *F8* key. Choose any integer values to enter when prompted. Make a note of the values you choose so you can re-enter them in later tests. In the space below, write down the Watch window values as they change. Does the operation of the loop make sense to you based on these values?

d. Did you notice something wrong with the program? You should have. Fix the bug and re-run the program. √

As you know, the counting sequence for a loop may vary. The C++ ***for*** statement permits some flexibility with this sequence. We'll explore one change next.

e. Go to the **File** menu and choose *Save As*. Create a new copy of the program with the file name *sum2.cpp*.

f. Modify this new file so that the counting sequence is reversed. That is, instead of counting from 1 to N, have the loop count from N down to 1. Use watches for the same four variables and step through the program. Once again, write down the values as they change. Was there a difference in the final result? Why or why not?

g. Modify the program once again so that it displays the counter *i* each time through the loop. √

*The **while** loop*

Not all loops we write can be controlled with a simple counter mechanism. More general than the ***for*** loop is the C++ ***while*** loop. This kind of loop can be constructed to terminate when any specific event occurs, regardless of whether it happens at the end of a counting sequence or some other way.

Since the counter control strategy which the ***for*** loop automates is a special case of the event control strategy, it stands to reason that anything that can be written using a ***for*** loop can be rewritten using a ***while*** loop.

h. Change the ***for*** loop in the original *sum1.cpp* file into a ***while*** loop. All you have to do is explicitly initialize the counter before the loop, set the logical expression, and increment the counter within the loop body. Test your changes.

## 3. Interactive control

What we've done so far is all well and good. But what if we don't know exactly how many values will be read in to be summed? The counter control strategy is not available to use because there is no way to end the counting sequence. One idea would be to ask the user each time after the first entry whether he or she wishes to continue. This would mean managing the repetition using another special case of event control, called *interactive control*. Using the letters *Y* for "yes" and *N* for "no," the loop would terminate when the response equals *N*. An example dialogue with the user might be:

```
Enter an integer        : 235
Another number? (Y or N): Y
Enter an integer        : -49
Another number? (Y or N): Y
Enter an integer        : 22
Another number? (Y or N): N

The total for the values processed is: 208
```

Since the solution does not depend on counting, either the ***while*** or ***do*** looping mechanism will have to be used. That's the next thing we'll try.

a. Go to the **File** menu and choose *Save As*. Create a new copy of the program with the file name *sum3.cpp*.

b. Change the program to read an unspecified number of integers and find their sum, using the interactive strategy. Since there is always a first value entered, you know that there will always be at least one iteration. Which C++ structure would handle this more naturally and efficiently? Why?

c. Set up the Watch window in order to test the looping structure you have written. In this case, you will try to verify that it works properly—that is, after any syntax errors are removed. √

### 4. Sentinel control

Let's continue to assume that the number *N* of values to be summed is not known or specified by the user. Another way to control the number of iterations while retaining the interactive element is to employ a special value that signals when the process is done. This *sentinel control* strategy—another special case of event control—has the advantage of not having to prompt each time to continue.

Let's modify the problem again so that sentinel control can be employed. Suppose that only positive integers are to be summed. In this case, any value outside the legal range may be designated as the sentinel value.

a. Go to the **File** menu and choose *Save As*. Create a new copy of the program with the file name *sum4.cpp*.

b. Designate some negative value as the sentinel value. (Why did you choose what you chose?) Modify the loop so that it operates until the value is input by the user. Of course, the user should be informed as to how they can signal that the sequence of input values has ended. (You may do this with a simple message at the beginning of the sequence.) Since the sentinel value is entered like any other value, it may occur first. This means that it is possible that there will be no numbers to sum. Which C++ structure would best handle this situation? Why?

c. Set up the Watch window in order to test the looping structure. Don't forget to test it on a list that contains no numbers to sum.

d. Test your solution again and see what happens if you enter a negative value other than the sentinel. Does your solution screen out these exceptions? In other words, is it *robust*? If not, modify the solution again so that it does not count negatives in the sum, but will continue until the sentinel value is expressly entered.

**Note:** An important aspect of robustness is what the program does to help the user correct their mistake. In this case, that means the program should print a warning message to the user whenever a negative value other than the sentinel is entered.

e. Test the modified program until you're satisfied with its reliability. √

### 5. Nested loops

As was the case with ***if*** statements, looping statements of all types can contain other loops as part of their loop bodies. That is, loops can be *nested*. Sometimes the logic of such nested loops can be hard to follow.

a. Close all currently open files in Borland C++. Open the file *count1.cpp*. Study the two ***for*** loops contained in the file. Make a prediction as to what the values of ***counter1*** and ***counter2*** will be when the program is run.

b. Now open the file *count2.cpp*. This file also contains two ***for*** loops, very similar to those in the previous file, but with a critical difference—these are nested. Again, before running the program, analyze it and predict the final values of ***counter1*** and ***counter2***.

c. Now run each version of the program. Were your predictions correct? If you are stumped, ask me.

d. Modify *count2.cpp* so that it displays the following:

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

Be sure that you have spaces between the numbers. √

## 7. Finishing up

Close all windows and exit Borland C++.