Review

Increment and Decrement Operator

To execute many algorithms we need to be able to add or subtract 1 from a given integer quantity. For example:

```
count = count + 1;     // what would happen if we used ==
                       // instead of = ?
count += 1;
```

Both of these statements **increment** the value of `count` by 1. If we replace "`+`" with "`-`" in the above code, then both statements **decrement** the value of `count` by 1. C++ also provides an **increment operator `++`** and a **decrement operator** -- to perform these tasks. There are two modes that can be used:

```
count++;    // increment operator in the postfix mode
count--;    // decrement operator in the postfix mode

++count;    // increment operator in the prefix mode
--count;    // decrement operator in the prefix mode
```

The two increment statements both execute exactly the same. So do the decrement operators. What is the purpose of having postfix and prefix modes? To answer this, consider the following code:

```
int age = 49;
if (age++ > 49)
  cout << "Congratulations - You have made it to the half-century"
       << " mark !" << endl;
```

In this code, the `cout` statement will not execute. The reason is that in the postfix mode the comparison between `age` and 49 is made *first*. Then the value of `age` is incremented by one. Since 49 is not greater than 49, the `if` conditional is false. Things are much different if we replace the postfix operator with the prefix operator:

```
int age = 49;
if (++age > 49)
  cout << " Congratulations - You have made it to the half-century"
       << " mark !" << endl;
```

In this code `age` is incremented first. So its value is 50 when the comparison is made. The conditional statement is true and the `cout` statement is executed.

The `while` Loop

Often in programming one needs a statement or block of statements to repeat during execution. This can be accomplished using a **loop**. A loop is a control structure that causes repetition of code within a program. C++ has three types of loops. The first we will consider is the **`while` loop**. The syntax is the following:

```
while (expression)
{
      statement_1;
      statement_2;
            :
      statement_n;
}
```

If there is only one statement, then the curly braces can be omitted. When a `while` loop is encountered during execution, the expression is tested to see if it is true or false. The block of statements is repeated as long as the expression is true. Consider the following:

Sample Program 5.1:

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

int main()
{

   int num = 5;
   int numFac = 1;

   while (num > 0)
   {
        numFac = numFac * num;
        num—;          // note the use of the decrement operator
   }

   cout << " 5! = " << numFac << endl;

   return 0;
}
```

This program computes 5! = 5 * 4 * 3 * 2 * 1 and then prints the result to the screen. Note how the `while` loop controls the execution. Since `num = 5` when the while loop is first encountered, the block of statements in the body of the loop is executed at least once. In fact, the block is executed 5 times because of the decrement operator which forces the value of `num` to decrease by one every time the block is executed. During the fifth **iteration** of the loop `num` becomes 0, so the next time the expression is tested `num > 0` is false and the loop is exited. Then the `cout` statement is executed.

What do you think will happen if we eliminated the decrement operator `num--` in the above code? The value of `num` is always 5. This means that the expression `num > 0` is always true! If we try to execute the modified program, the result is an **infinite loop**, i.e., a block of code that will repeat forever. One must be very cautious when using loops to ensure that the loop will terminate. Here is another example where the user may have trouble with termination.

Sample Program 5.2:

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

int main()
{
    char letter = 'a';

    while (letter != 'x')
    {
        cout << "Please enter a letter"      << endl;
        cin  >> letter;
        cout << "The letter your entered is " << letter << endl;
    }

    return 0;
}
```

Note that this program requires input from the user during execution. Infinite loops can be avoided, but it would help if the user knew that the `'x'` character terminates the execution. Without this knowledge the user could continually enter characters other than `'x'` and never realize how to terminate the program. In the lab assignments you will be asked to modify this program to make it more user friendly.

Counters

Often a programmer needs to control the number of times a particular loop is repeated. One common way to accomplish this is by using a **counter**. For example, suppose we want to find the average of five test scores. We must first input and add the five scores. This can be done with a **counter-controlled** loop as shown in Sample Program 5.3. Notice how the variable named `test` works as a counter. Also notice the use of a constant for the number of tests. This is done so that the number of tests can easily be changed if we want a different number of tests to be averaged.

Sample Program 5.3:

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

const int NUMBEROFTESTS = 5;

int main()
{
      int score ;            // the individual score read in
      float total = 0.0;     // the total of the scores
      float average;         // the average of the scores
      int test = 1;          // counter that controls the loop

      while (test <= NUMBEROFTESTS)     // Note that test is 1 the first time
                                        // the expression is tested
      {
            cout << "Enter your score on test " << test << ": " << endl;
            cin >> score;

            total = total + score;
```

```
            test++;
      }

      average = total / NUMBEROFTESTS;

      cout  << "Your average based on " << NUMBEROFTESTS
            << " test scores is " << average << endl;

      return 0;
}
```

Sample Program 5.3 can be made more flexible by adding an integer variable called `numScores` that would allow the user to input the number of tests to be processed.

Sentinel Values

We can also control the execution of a loop by using a **sentinel value** which is a special value that marks the end of a list of values. In a variation of the previous program example, if we do not know exactly how many test scores there are, we can input scores which are added to total until the sentinel value is input. Sample Program 5.4 revises Sample Program 5.3 to control the loop with a sentinel value. The sentinel in this case is -1 since it is an invalid test score. It does not make sense to use a sentinel between 0 and 100 since this is the range of valid test scores. Notice that a counter is still used to keep track of the number of test scores entered, although it does not control the loop. What happens if the first value the user enters is a -1?

Sample Program 5.4:

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

int main()
{
   int score ;              //    the individual score read in
   float total = 0.0;       //    the total of the scores
   float average;           //    the average of the scores
   int test = 1;            //    counter that controls the loop

   cout << "Enter your score on test " << test
        << " (or -1 to exit): " << endl;

   cin >> score;            // Read the 1st score

   while (score != -1)      // While we have not entered the sentinel
                            // (ending) value, do the loop
   {
     total = total + score;
     test++;
     cout << "Enter your score on test " << test
          << " (or -1 to exit): " << endl;
     cin  >> score;                          // Read the next score
   }
```

```
   if (test > 1)      // If test = 1, no scores were entered
   {
      average = total / (test - 1);

      cout << "Your average based on " << (test - 1)
           << " test scores is " << average << endl;
   }

   return 0;
}
```

Notice that the program asks for input just before the `while` loop begins and again as the last instruction in the `while` loop. This is done so that the `while` loop can test for sentinel data. Often this is called **priming the read** and is frequently implemented when sentinel data is used to end a loop.

Data Validation

One nice application of the `while` loop is data validation. The user can input data (from the keyboard or a file) and then a `while` loop tests to see if the value(s) is valid. The loop is skipped for all valid input but for invalid input the loop is executed and prompts the user to enter new (valid) input. The following is an example of data validation.

```
cout << "Please input your choice of drink "
     << "(a number from 1 to 4 or 0 to quit)" << endl;
cout << " 1 -Coffee" << endl
     << " 2 -Tea" << endl
     << " 3 -Coke" << endl
     << " 4 -Orange Juice" << endl << endl
     << " 0 -QUIT" << endl << endl;

cin >> beverage;

while (beverage < 0 || beverage > 4)
{
      cout << "Valid choices are 0 - 4. Please re-enter: ";
      cin  >> beverage;
}
```

What type of invalid data does this code test for? If `beverage` is an integer variable, what happens if the user enters the character '$' or the float 2.9?

The `do-while` Loop

The while loop is a **pre-tes**t or **top test** loop. Since we test the expression before entering the loop, if the test expression in the `while` loop is initially false, then no iterations of the loop will be executed. If the programmer wants the loop to be executed at least once, then a **post-test** or **bottom test** loop should be used. C++ provides the **`do-while` loop** for this purpose. A `do-while` loop is similar to a while loop except that the statements inside the loop body are executed *before* the expression is tested. The format for a single statement in the loop body is the following:

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

Note that the statement must be executed once even if the expression is false. To see the difference between these two loops consider the code

```
int num1 = 5;
int num2 = 7;

while (num2 < num1)
{
      num1 = num1 + 1;
      num2 = num2 - 1;
}
```

Here the statements `num1=num1+1` and `num2=num2-1` are never executed since the test expression `num2 < num1` is initially false. However, we get a different result using a `do-while` loop:

```
int num1 = 5;
int num2 = 7;

do
{
      num1 = num1 + 1;
      num2 = num2 - 1;
} while (num2 < num1);
```

In this code the statements `num1=num1 + 1` and `num2=num2-1` are executed exactly once. At this point `num1=6` and `num2=6` so the expression `num2 < num1` is false. Consequently, the program exits the loop and moves to the next section of code. Also note that since we need a block of statements in the loop body, curly braces must be placed around the statements. In Lab 5.2 you will see how `do-while` loops can be useful for programs that involve a repeating menu.

The `for` Loop

The **`for` loop** is often used for applications that require a counter. For example, suppose we want to find the average (mean) of the first *n* positive integers. By definition, this means that we need to add 1 + 2 + 3 + . . . + *n* and then divide by *n*. Note this should just give us the value in the "middle" of the list 1, 2, ..., *n*. Since we know exactly how many times we are performing a sum, the `for` loop is the natural choice.

The syntax for the `for` loop is the following:

```
for (initialization; test; update)
{
    statement_1;
    statement_2;
     :
    statement_n;
}
```

Notice that there are three expressions inside the parentheses of the `for` statement, separated by semicolons.

1. The **initialization expression** is typically used to initialize a counter that must have a starting value. This is the first action performed by the loop and is done only once.
2. The **test expression**, as with the `while` and `do-while` loops, is used to control the execution of the loop. As long as the test expression is true, the body of the `for` loop repeats. The `for` loop is a pre-test loop which means that the test expression is evaluated before each iteration.
3. The **update expression** is executed at the end of each iteration. It typically increments or decrements the counter. Now we are ready to add the first *n* positive integers and find their mean value.

Sample Program 5.5:

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

int main()
{
   int value;
   int total = 0;
   int number;
   float mean;

   cout << "Please enter a positive integer" << endl;
   cin >> value;

   if (value > 0)
   {
     for (number = 1; number <= value; number++)
     {
          total = total + number;
     }                                  // curly braces are optional since
                                        // there is only one statement

     mean = static_cast<float>(total) / value;    // note the use of the typecast
                                                  // operator

     cout << "The mean average of the first " << value
          << " positive integers is " << mean << endl;
   }
   else
     cout << "Invalid input - integer must be positive" << endl;

   return 0;
}
```

Note that the counter in the `for` loop of Sample Program 5.5 is `number`. It increments from 1 to `value` during execution. There are several other features of this code that also need to be addressed. First of all, why is the typecast operator needed to compute the mean? What do you think will happen if it is removed?

Finally, what would happen if we entered a float such as 2.99 instead of an integer? Lab 5.3 will demonstrate what happens in these cases.

Nested Loops

Often programmers need to use a loop within a loop, or **nested loops**. Sample Program 5.6 below provides a simple example of a nested loop. This program finds the averagenumber of hours per day spent programming by each student over a three-day weekend. The outer loop controls the number of students and the inner loop allows the user to enter the number of hours worked each of the three days for a given student. Note that the inner loop is executed three times for each iteration of the outer loop.

Sample Program 5.6:

```
// This program finds the average time spent programming by a student each
// day over a three day period.

#include <iostream>
using namespace std;

int main()
{
    int numStudents;
    float numHours, total, average;
    int count1 = 0, count2 = 0;   // these are the counters for the loops

    cout << "This program will find the average number of hours a day"
        << " that each given student spent programming over a long weekend"
        << endl << endl;
    cout << "How many students are there ?" << endl << endl;
    cin  >> numStudents;

    for (count1 = 1; count1 <= numStudents; count1++)
    {
         total = 0;

         for (count2 = 1; count2 <= 3; count2++)
         {
           cout << "Please enter the number of hours worked by student "
               << count1 << " on day " << count2 << "." << endl;
           cin  >> numHours;

           total = total + numHours;
         }

         average = total / 3;

        cout << endl;
         cout << "The average number of hours per day spent programming by"
             << " student " << count1 <<" is " << average
             << endl << endl << endl;
    }

   return 0;
}
```

In Lab 5.4 you will be asked to modify this program to make it more flexible.

Review #5 CS 151 SEC# _____ Name _____

Fill-in-the-Blank Questions

1) A block of code that repeats forever is called _____.

2) To keep track of the number of times a particular loop is repeated, one can use a(n) _____.

3) An event controlled loop that is always executed at least once is the _____

4) An event controlled loop that is not guaranteed to execute at least once is the _____

5) In the conditional `if(++number < 9)`, the comparison `number < 9` is made _____ and `number` is incremented _____. (Choose first or second for each blank.)

6) In the conditional `if(number++ < 9)`, the comparison `number < 9` is made _____ and `number` is incremented _____. (Choose first or second for each blank.)

7) A loop within a loop is called a _____.

8) To write out the first 12 positive integers and their cubes, one should use a(n) _____ loop.

9) A(n) _____ value is used to indicate the end of a list of values. It can be used to control a `while` loop.

10) In a nested loop the _____ loop goes through all of its iterations for each iteration of the _____ loop. (Choose inner or outer for each blank.)