Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-II
Lesson: Repetition Structures
Lesson Developer: Rakhi Saxena
College/Department: Deshbandhu College, University of Delhi

Table of Contents

3.1 Repetition Structures - I

You have learnt selection structures in the previous chapters. In this chapter, you will learn another programming construct – the repetition structure. Repetitive tasks means to do the same task multiple times. To solve certain problems in programming we need to perform some sequence of steps repeatedly. This is also called looping. Most programming languages provide repetition structures so that you can perform a set of statements repeatedly without having to write the same lines of code again and again. C++ provides two types of looping constructs – the pre-test and the post-test loops. In this chapter you will learn about the "while" loop – a pre-test loop construct and the – "do-while" loop – a post-test loop construct. These concepts will help you write programs with repetition structures.

3.1.1 Learning Objectives

After reading this chapter you should be able to:

3.1 Design and Develop programs with repetition structures.
- 3.1.1 Define pre-test and post-test loops.
- 3.1.2 Distinguish between pre-test and post-test loops.
- 3.1.3 Use the pre-test - "while" loop.
- 3.1.4 Use the post-test - "do-while" loop.
- 3.1.5 Construct sentinel controlled loops.
- 3.1.6 Apply loops for counting and for summation problems.

3.1.2 Looping

In real life you often do something repeatedly, for example, – for drinking a cup of tea – take one sip at a time again and again – until the tea is over; for walking – take a step one after another – until you reach your destination; for attending a course – go to the class, listen to the lecture everyday – until the semester is over. Consider another repetitive task such as reading a book, first you open the book, and then repeatedly - read a page; flip the page – until you get to the end of the book, then close the book.

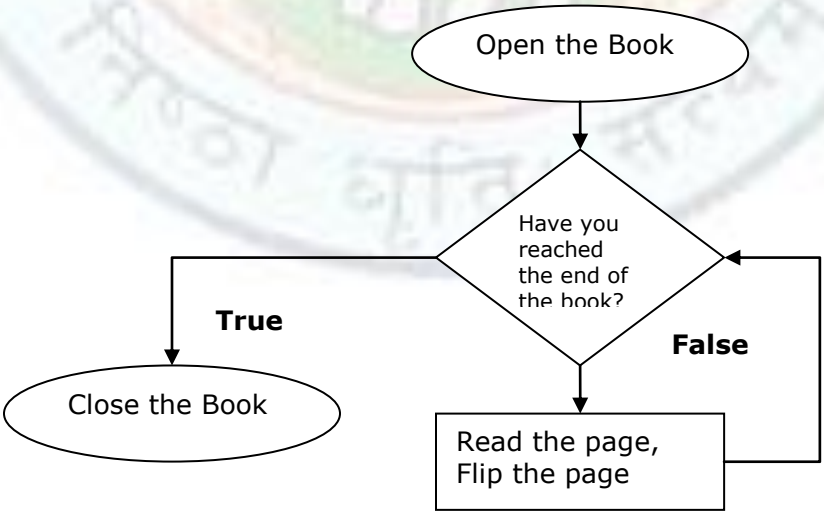


Figure 3.1 Repetition in daily life

Similarly, when writing programs, you might need to perform the same sequence of statements repeatedly until some condition is met.
The ability of a computer to perform the same set of actions again and again is called **looping**. All programming languages provide statements to facilitate writing loops. A loop is the basic component of a repetition structure.
The sequence of statements that is repeated again and again is called the **body** of the loop. The **test conditions** that determine whether a loop is entered or exited are usually written using relational or logical operators. A single pass through the loop is called **iteration**. For example, a loop that repeats the execution of the body three times goes through three iterations.
The repetition structures in programs are of two types – **pre-test loops** and **post-test loops**.

3.1.2.1 Pre-Test Loops – The "while" Loop

A pre-test loop is one where the test condition is evaluated before the body of the loop is executed.

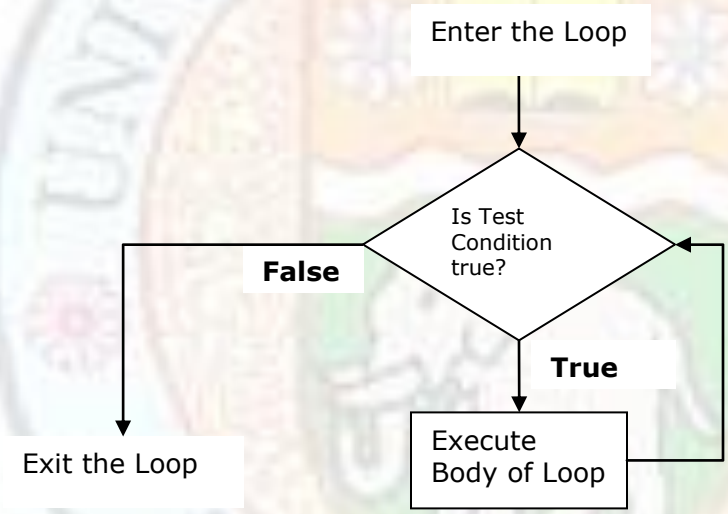


Figure 3.2 Pre-test Loop

The C++ construct for a pre-test loop is the **while** statement. Its syntax is as below:

```
while (expression)
{
        //Body of loop
}
```

Let us write a program that displays the squares of numbers entered by the user until the user enters a negative number. The negative number is not displayed.
The following steps need to be performed. The tasks that need to be performed repeatedly are in step 3 – these constitute the body of the loop. The test condition is in step 2.

1. Take as input a number from the user
2. Test if the number is a positive number
3. If yes, Display the square of the number;
   Take another number as input from the user;

Go back to step 2
Otherwise, exit the loop.

The corresponding C++ code is as below:

```
cin >> number;
while (number >=0 )
{
    cout << "Square of the number is = " << number * number << endl;
    cout <<"Enter another number: ";
    cin >> number;
 }
```

The test condition in this loop tests whether the number is a positive integer. If it is then the statements within the curly braces – the body of the loop is executed – otherwise the loop is exited.

Value addition: Source Code

Heading text – Squares of Numbers

```
/* This program prints squares of numbers entered by the user using a while loop. If the
number entered is negative, the loop is exited, otherwise the body of the loop is executed*/

#include <iostream>
using namespace std;

int main()
{
   int number;

   cout << "This program prints squares of numbers repeatedly."<< endl;
   cout << "Enter a negative number to quit." << endl<<endl;
   cout <<"Enter a number: ";
   cin >> number;
   while (number >=0 )            // test condition of the loop
   {
       cout << "Square of the number is = " << number * number << endl<<endl;
       cout <<"Enter another number: ";
       cin >> number;
    }
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – Squares of Numbers

```
C:\C++Programs\Squares.exe
This program prints squares of numbers repeatedly.
Enter a negative number to quit.

Enter a number: 2
Square of the number is = 4

Enter another number: 3
Square of the number is = 9

Enter another number: 4
Square of the number is = 16

Enter another number: 5
Square of the number is = 25

Enter another number: -1
```

Source: Self made

3.1.2.2 Post-Test Loops – The "do-while" Loop

A post-test loop is one where the test condition is evaluated after the body of the loop has been executed.

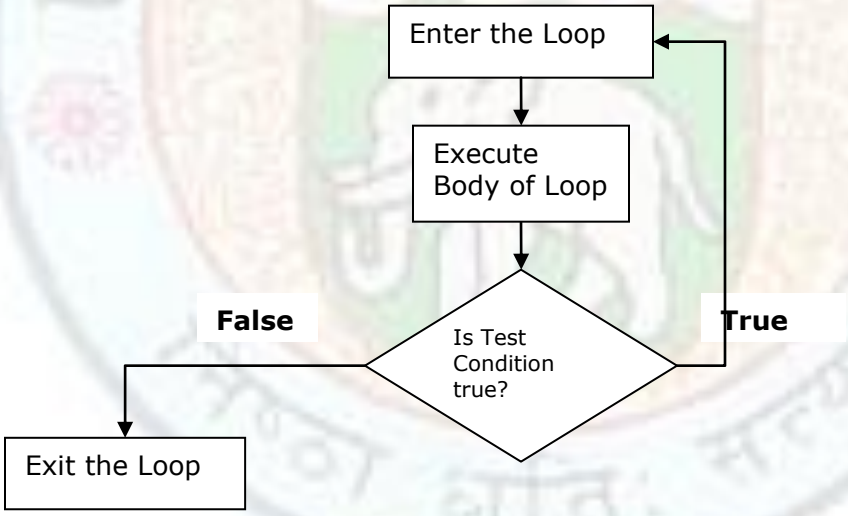


Figure 3.3 Post-test Loop

The C++ construct for a pre-test loop is the **do while** statement. Its syntax is as below:

```
do
{
        //Body of loop
}
while (expression);
```

Let us write the same program that displays the squares of numbers entered by the user until the user enters a negative number. This time we will use a do-while loop.

The following steps need to be performed. The tasks that need to be performed repeatedly are in steps 1 and 2 – these constitute the body of the loop. The test condition is in step 3.

1. Take as input a number from the user
2. Display the square of the number
3. Take another number as input from the user
4. Test if the number is positive. If yes, Go back to step 2
   Otherwise, Exit the loop

The corresponding C++ code is as below:

```
cin >> number;
do
{
    cout << "Square of the number is = " << number * number << endl<<endl;
    cout <<"Enter another number: ";
    cin >> number;
 }
 while (number >=0 );
```

First the body of the loop is executed. The test condition is then tested for whether the number is a positive integer. If it is then the statements within the curly braces – the body of the loop is executed again– otherwise the loop is exited.

Value addition: Source Code

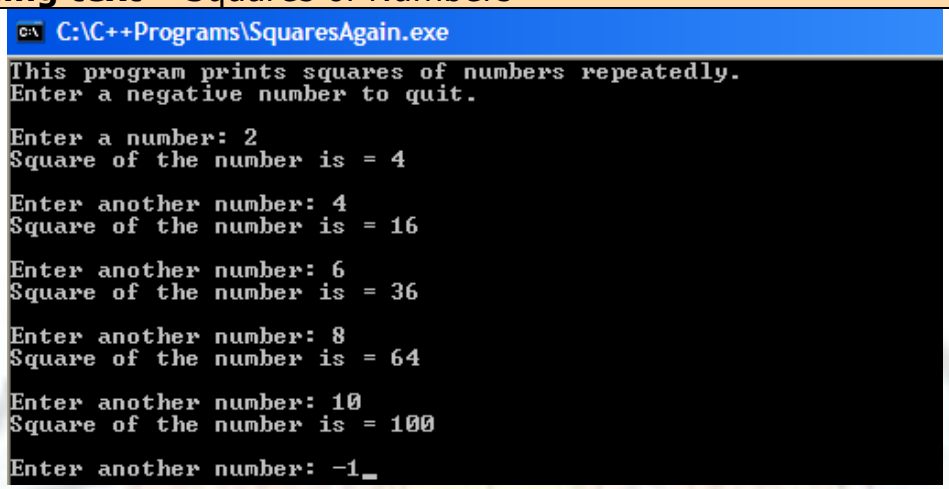
Heading text – Squares of Numbers Again

```
/* This program prints squares of numbers entered by the user
   using a do-while loop. If the number entered is negative,
   the loop is exited, otherwise the body of the loop is executed
*/

#include <iostream>
using namespace std;

int main()
{
   int number;

   cout << "This program prints squares of numbers repeatedly."<< endl;
   cout << "Enter a negative number to quit." << endl<<endl;
   cout <<"Enter a number: ";
   cin >> number;
   do
   {
        cout << "Square of the number is = " << number * number;
        cout << endl<<endl;
        cout <<"Enter another number: ";
        cin >> number;
    }
    while (number >=0 );            // test condition of the loop
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – Squares of Numbers



Source: Self made

3.1.2.3 Comparison of Pre-Test Loop and Post-Test Loop

As seen from the Squares of Numbers example, you can use either a pre-test or a post-test loop to accomplish the same task. However, some tasks are easier to do with the former and some with the latter. Use the while loop if you need the loop to run zero or more times. Use the do-while loop when you need the loop to run once or more times (that is, at least once).

Value addition: FAQ

Heading text – Difference between the pre-test and post-test loops

PRE-TEST LOOP	POST-TEST LOOP
A pre-test loop is an entry controlled loop – it tests for a condition prior to running a block of code	A post test loop is an exit control loop - it tests for a condition after running a block of code
Pretest loop runs zero or more times Body of loop may never be executed	Post test loop runs once or more times but at least once Body of loop is executed at least once
The variables in the test condition must be initialized prior to entering the loop structure.	It is not necessary to initialize the variables in the test condition prior to entering the loop structure.

while (condition)
{
// do something
}
	do {
	// do something
	} while (condition);

Source: Self made

Value addition: STYLE TIP

Heading text – Looping

1. Indent your loops. This will make the code easier to read and manage.

Indented Code	**Not Indented Code**

while (number >=0
{
cout << 2 * number;
cin >> number;
}
	while (number >=0
	{
	cout << 2 * number;
	cin >> number;
	}

2. If there is only one statement in the body of the loop, the set of curly braces enclosing the body can be omitted. For example,

```
while ( number < another_number)
        number++;
```

Source: Self made

Value addition: Common Coding Errors

Heading text – Looping

1. **Infinite Loops**

This type of error occurs when the loop runs forever, it is never exited. This happens when the test condition is always true.
For example,

```
number = 0;
while (number > 0)
{
    counter++;
 }
```

This loop will run forever since number never changes – it remains at 0.

2. **Syntax Errors**

- Do not forget the semi colon at the end of the test condition in a do-while loop.
- Do not place a semi colon at the end of the test condition in a while loop.
  For example,

```
int x = 5;
while( x > 0 );
    x--;
```

  This loop is an infinite loop - The semicolon after the while defines the statement to be repeated as the null statement (which does nothing). If the semi colon at the end is be removed, the loop will work as expected.

Source: Self made

3.1.3 Applications of Repetition Structures

In this section, we will write some programs that make use of repetitive constructs. Loops are often used to input a large amount of data into a program. The test condition of the loop should cause the loop to exit when all the data has been input. Usually, to force the loop to exit, a **sentinel** value is used – this sentinel is a special input value that cannot occur in the data. An input of this sentinel causes the loop to terminate. For example, if the input data -a set of marks scored by students is input using a loop, the sentinel value could be chosen as -1. No student will get a negative mark. An input of -1 will cause the loop to exit. The sentinel value is also referred to as "end of input marker".

3.1.3.1 Counting

Let us write a counting program that uses a loop. This program will take as input a set of numbers - marks obtained by students, and count the number of students that have passed in the exam. The program will stop when a sentinel value of -1 is input. For this program we will need a **counter**. Imagine the counter as an empty jar. Each time the input marks are more than the passing mark, we drop a marble to the jar – that is, we increase the contents of the jar by 1. When all the marks have been input, the number of marbles in the jar is the number of students that have passed.

Value addition: Animation
Heading text – A Counter

Source: Self made For clip art images www.pixmac.com/.../jar+of+marbles/000026969815 http://www.graphicsfactory.com/Clip_Art/People/Hands/marble700_158425.html http://mt.educarchile.cl/MT/jjbrunner/archives/answer.gif

When writing a program, we simulate the counter with a variable – it is initialized to 0 and each time we wish to increment the counter, we add one to the variable. Here's the part of code for counting the number of students that have passed:

```
cin >> marks;
while (marks != -1)              // loop will exit when sentinel (-1) is entered
{
        if (marks >= passing_marks)
           counter = counter + 1;
                // increment counter if marks are more than the passing_marks
         cout <<"Enter a student's marks(Enter a negative number to quit.): ";
         cin >> marks;
 }
```

cout << "Number of students that have passed are " << counter;

Value addition: Source Code

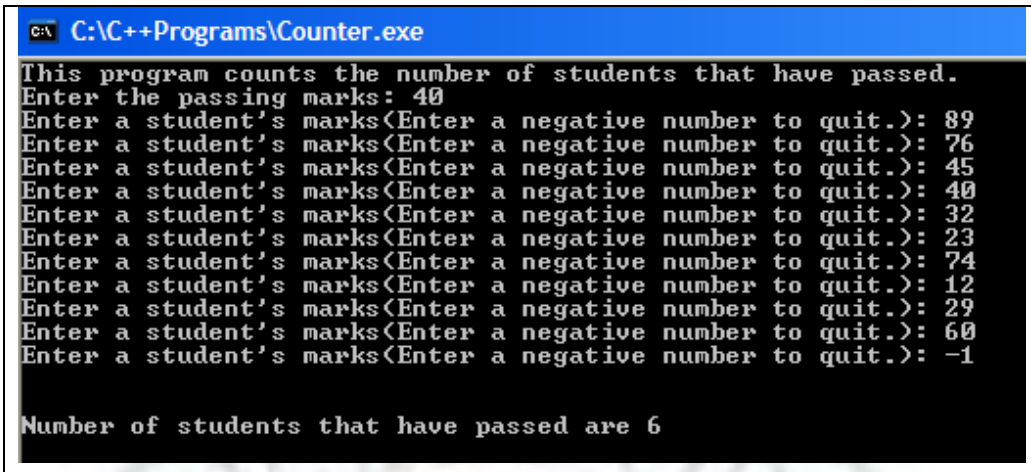
Heading text – Counting Program

```
/* This program counts the number of students that have passed. The passing marks and
the marks scored are entered by the user using a while loop. If the marks entered is -1,
the loop is exited, otherwise the body of the loop is executed. The counter is incremented
in the body if the marks are more than the passing marks.*/

#include <iostream>
using namespace std;

int main()
{
   int marks, passing_marks;
   int counter = 0;                //initialize counter to zero

   cout <<"This program counts the number of students that have passed."<< endl;
   cout <<"Enter the passing marks: ";
   cin >> passing_marks;
   cout <<"Enter a student's marks(Enter a negative number to quit.): ";
   cin >> marks;
   while (marks != -1)             // loop will exit when sentinel (-1) is entered
   {
        if (marks >= passing_marks)
            counter = counter + 1;  // increment counter if marks are more than the
                                    //passing_marks
        cout <<"Enter a student's marks(Enter a negative number to quit.): ";
        cin >> marks;
    }
    cout << endl<<endl;
    cout << "Number of students that have passed are " << counter;
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – Counting Program



Source: Self made

3.1.3.2 Summation of a Set of Numbers

Next, we will write a summation program that uses a loop. This program will take as input a set of numbers - marks obtained by students, and sum the marks of the students. The program will stop when a sentinel value of -1 is input.
For this program, we will need an **accumulator**. Continuing with the marble jar analogy, this time each time a mark is input; add as many marbles as the marks to the jar. For example, if the student scored 32 marks, we add 32 marbles to the jar. When all the marks have been input, the number of marbles "accumulated" in the jar is the sum of all the marks the students have scored.
Here's the part of code for increasing the accumulator to sum the marks of the students:

```
accumulator = accumulator + marks;
```

A counter can be used to compute the total number of students – the number of times the body of the loop is executed will tell you the number of students.
The statement

```
number_of_students ++;
```

executed in the body of the loop will give you the number of students. Don't forget to initialize the number_of_students variable to zero before the loop statement!

The sum of the marks can be used to compute the average marks of the class by dividing th accumulator value by the number of students.

Value addition: Source Code

Heading text- Summation Program

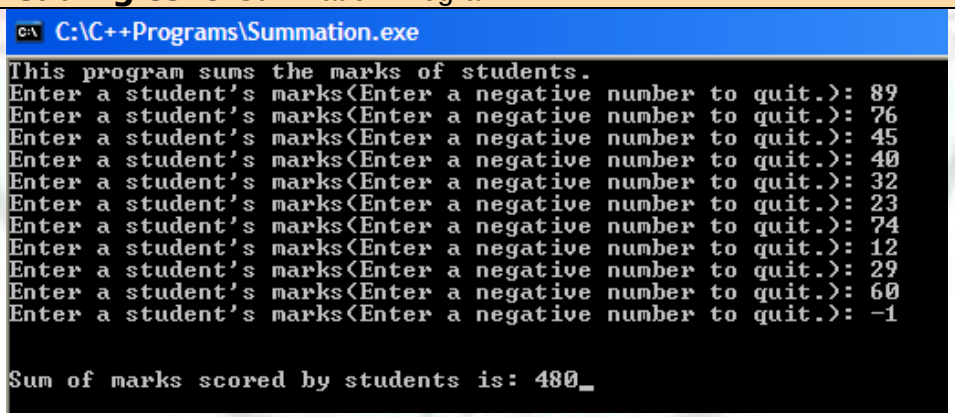
```
/* This program sums the marks of students.
   The marks scored are entered by the user in a while loop.
   If the mark entered is -1, the loop is exited,
   otherwise the body of the loop is executed -
   The accumulator is incremented by the input marks
*/
```

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

int main()
{
   int marks;
   int accumulator = 0;              //initialize accumulator to zero

   cout <<"This program sums the marks of students."<< endl;
   cout <<"Enter a student's marks(Enter a negative number to quit.): ";
   cin >> marks;
   while (marks != -1)       // loop will exit when sentinel (-1) is entered
   {
        accumulator = accumulator + marks;  // increment accumulator by
                                            //input marks
        cout <<"Enter a student's marks(Enter a negative number to quit.): ";
        cin >> marks;
    }
    cout << endl<<endl;
    cout << "Sum of marks scored by students is: " << accumulator;
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text- Summation Program



Source: Self made

3.2 Repetition Structures - II

You have learnt to write pre test and post test loops in the previous section. In this section, you will learn to apply another pre test loop – the counter controlled loop. C++ provides the "for" loop construct for implementing counter controlled loops. You will learn to apply the "for" loop for programming tasks.
These concepts will help you write programs with counter controlled repetition structures.

3.2.1 Learning Objectives

After reading this chapter you should be able to:

3.2 Design and develop programs with counter controlled loops.
- 3.2.1 Define event controlled and counter controlled loop.
- 3.2.2 Construct the pre-test counter controlled loop - "for" loop.
- 3.2.3 Create and use nested loops.
- 3.2.4 Apply loops for printing patterns.

3.2.2 Event Controlled and Counter Controlled Loops

In the examples of the previous chapter, the number of loop iterations was determined by user input. If the user enters a positive number the expression evaluates to true and the loop body is executed. If the user enters a negative number the expression evaluates to false and the loop is exited.

Such a loop where an event, such as user signaling end of input, changes the loop expression from true to false is called an **event controlled** loop. Event-controlled loops continue until some event occurs. The number of times the loop will execute is not known by the program when the loop starts.

Sometimes, the number of times a loop should execute is known prior to the execution of the loop. A **counter controlled loop**, a pre test loop, is executed a known fixed number of times.

You already know you can use a counter variable to count the loop iterations. The counter variable can also be used to control a counter controlled loop. It forces the loop to execute a predefined number of times in the following way:

1. Define and initialize the counter with an initial_value.
2. Test if the counter has counted up to the desired loop iterations, a limit_value.
3. If yes, exit the loop, otherwise, update the counter with a step_value and repeat the loop.

The loop counter is also called the **loop index**.

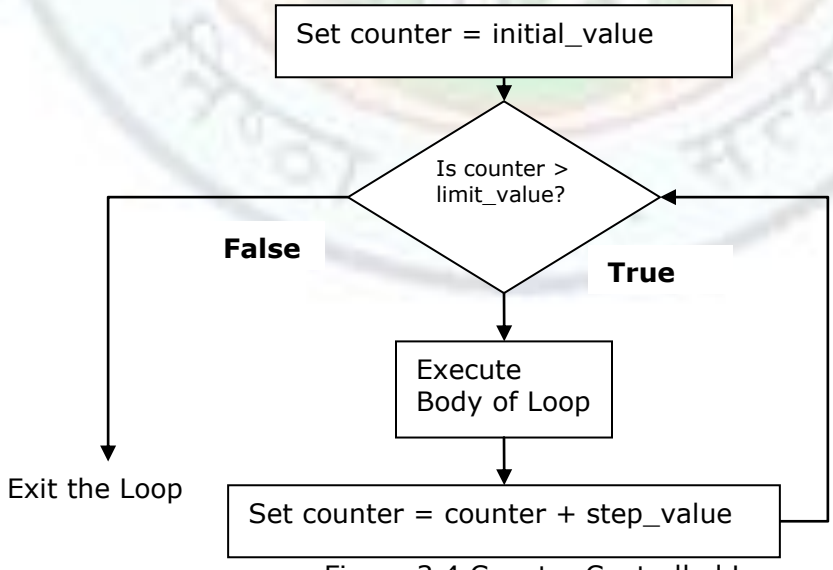


Figure 3.4 Counter Controlled Loop

The C++ construct for a pre-test counter controlled loop is the "**for"** statement.

3.2.2.1 Counter Controlled Loop - The "for" Loop

The "for" loop has the following syntax:

```
for (counter = initial_value; test_condition; counter = counter + step)
{
  // body of the loop
}
```

- The *initial_value* initializes the value of the loop counter.
- The *test_condition* tests whether the loop should be executed again. The loop is exited when the test condition fails.
- The *step* updates the counter in each loop iteration.

For example, consider the following loop that prints "Hello World" three times:

```
int count;
for ( count = 1; count <= 3; count = count + 1)
{
        cout << "Hello World" << endl;
}
```

When the loop is entered, the loop counter: count is set to 1. Then the *test_condition* tests whether the loop variable, count <= 3. If it is, the body of the loop is executed – a "Hello World" is displayed. The *step* then increments count by 1. The *test_condition* is tested again to decide whether to execute the body. When count exceeds 3, the loop is exited.

In English, we read the loop above as – for count is 1; count is less than or equal to 3; execute the body of the loop; increment count; loop back to the test condition.

The loop given above counts up the loop index – it is an **incrementing** loop. We can also count down in a loop for a **decrementing** loop. The following decrementing loop will execute 5 times.

```
for ( int count = 5; count >= 1; count = count - 1)
{
        //body of the loop
}
```

Usually the counter variable is defined within the for loop and the increment (or decrement) operators are used for changing the counter:

```
for ( int count = 1; count <= 5; count++)
{
        //body of the loop
}
```

The loop index can be incremented or decremented by any value, not just 1. The following loop increments the loop index by 2. It displays odd numbers between 1 and 20.

```
for ( int count = 1; count <= 20; count = count +2)
{
        cout << count << endl;
}
```

The loop index can begin with any value, not necessarily 1. The following loop also iterates 5 times.

```
for ( int count = 0; count < 5; count++)
{
        //body of the loop
}
```

The counter in the loop iterates with values – 0, 1,2,3,4 – for a total of five times. Notice that the test condition is count < 5 and not count <= 5. If it was the latter, the loop would have iterated 6 times for loop index as – 0, 1, 2, 3, 4, 5.

Value addition: Did you Know?

Heading text – Converting a while loop to a for loop

The following code fragment is a *while* loop that prints "Hello World" three times:

```
count = 1;
while (count <= 3)
{
      cout << "Hello World" << endl;
      count++;
}
```

The following code fragment is an equivalent "for" loop:

```
for(count = 1; count <= 3; count++)
     cout << "Hello World" << endl;
```

Source: Self made

Value addition: STYLE TIP

Heading text – Looping

1. If there is only one statement in the body of the loop, the set of curly braces enclosing the body can be omitted. For example,
   ```
   while ( number < another_number)
         number++;
   ```
2. Use indentation with the alignment of the body of the loop.
3. You can use multiple items in the initialization and the updation part of the loop by separating them with the comma operator.
   For example,
   ```
   for ( x = 0, y = 10; x < 10; x++, y--)
         cout << x <<"  " << y << endl;
   ```

Source: Self made

Value addition: Common Coding Errors

Heading text – The for Loop

1. Initial value is greater than the limit value and the loop increment is positive.
   For example,
   ```
   for ( count = 5; count <= 1; count++)
   ```
   In this case, body of the loop will never be executed

2. Initial value is lesser than the limit value and the loop increment is negative.
   For example,
   ```
   for ( count = 1; count >= 5; count--)
   ```
   In this case, body of the loop will never be executed.

3. Placing a semicolon at the end of a for statement:
   ```
   for ( count = 1; count <= 5; count++);
   {
       //body of loop
   }
   ```
   This has the effect of defining the body of the loop to be empty. The statements within the curly braces will be executed only once and not as many times as expected.

4. Executing the loop either more or less times than the desired number of times.

   For example, the following loop iterates 4 times and not the intended 5 times because it exits when count = 5.

   ```
   for ( count = 1; count < 5; count ++)
   ```

   The correct way to loop five times would be to test for count <= 5.
   Such errors are known as off by one errors.

5. Using a loop index declared within the loop outside the loop.
   ```
   for ( int count = 1; count < 5; count ++)
   {
       cout << count;
   }
   cout << count;              //error!!
   ```
   The scope of the variable count is only within the body of the loop. It is not visible outside the loop.

Source: Self made

3.2.2.2 Printing a multiplication table

Let us apply the for loop to print a multiplication table of a number taken as user input.
To print the table, start from count = 1, print count * number, increment count by 1 and repeat this until count exceeds 10.

The loop to do the printing task is:

```
for ( int count = 1; count <= 10; count++)
     cout << number<< " X "<<count<<" = " << count * number << endl;
```

Note: In this code fragment, the variable count has a dual purpose – it is used as a loop counter and is also used in the output statement.

Value addition: Source Code

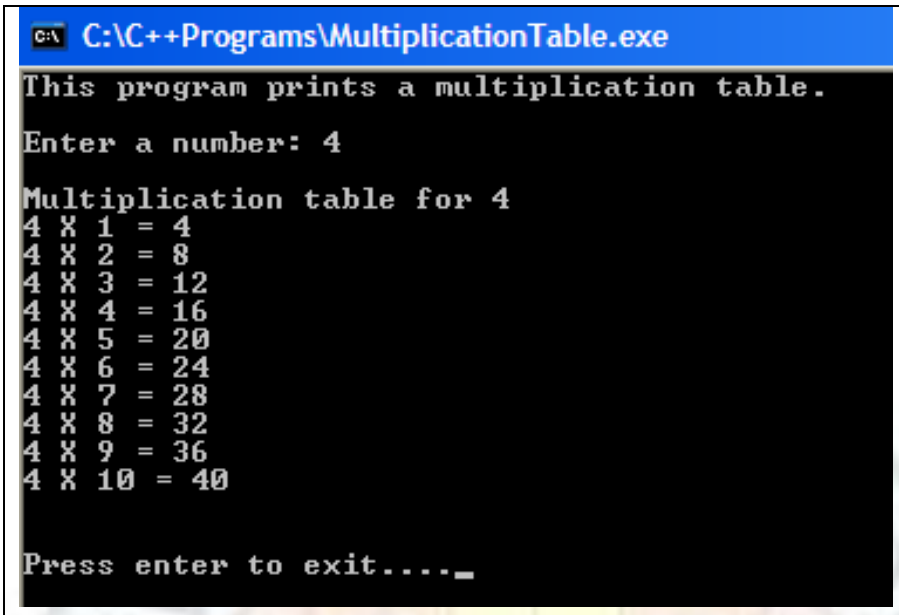
Heading text – MultiplicationTable Program

```
/* This program prints the multiplication table for a number entered by the user.*/

#include <iostream>
using namespace std;

int main()
{
   int number;

   cout <<"This program prints a multiplication table."<< endl<<endl;
   cout <<"Enter a number: ";
   cin >> number;
   cout << endl << "Multiplication table for " <<number << endl;
   for ( int count = 1; count <= 10; count++)
   {
        cout << number<< " X "<<count<<" = " << count * number << endl;

    }
    cout << endl<<endl<<"Press enter to exit....";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – MultiplicationTable Program



Source: Self made

3.2.3 Nested Loops

Sometimes, you will need to write a loop contained entirely within another loop in a program. Such loops are called **nested loops**. The larger loop is called the **outer loop** and the one contained inside is called the **inner loop**.
The inner loop is executed for each iteration of the outer loop. The outer loop index is updated only after the inner loop is completely finished.
A loop can contain any number of loop statements in itself.

The following is a nested loop:

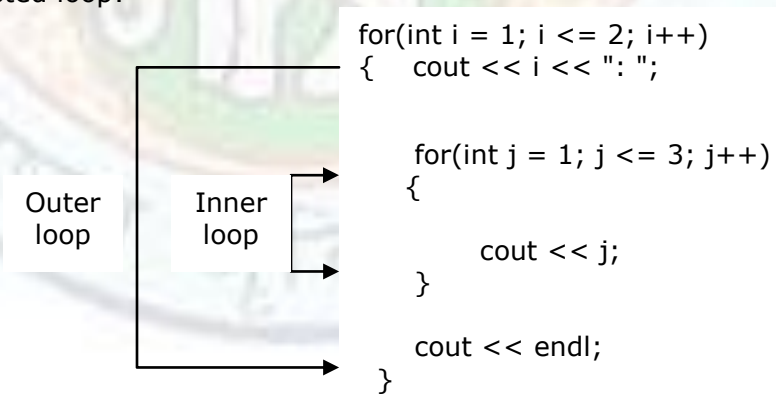


Figure 3.5 Nested Loop

The outer loop variable i varies from 1 up to 2. The inner loop variable j varies from 1 up to 3 for each value of i.
The inner loop displays a row of numbers - 123. The outer loop prints the outer loop index and causes the inner loop to repeat 2 times. The effect is to display the following:

1: 123
2: 123

Each row is printed in a newline as the outer loop prints a newline after every inner loop is executed. Let us look at a trace of the two nested loops – how the loop indices change.

Loop Indices		Display Screen
i	**j**	
1	1	1: 1
	2	1: 12
	3	1: 123
	End of inner loop	Newline is printed
2	1	1:123 2:1
	2	1:123 2:12
	3	1:123 2:123
	End of inner loop	Newline is printed
End of outer loop		

Outer Loop Index

Inner Loop Index

Figure 3.6 Trace of Nested Loop

3.2.3.1 Printing a larger multiplication table

In the last section, we printed a multiplication table. We will now write a program that calculates and prints a multiplication table for all values between 1 and 9 (inclusive).

The inner loop prints the table for a number in one row with each value separated by tabs:

```
for ( int count = 1; count <= 10; count++)
     cout << count * number << "\t";
```

We need to repeat this loop for values of number from 1 to 9. For this we enclose the inner loop in an outer loop with loop index varying from 1 to 9. We also need to print a newline at the end of each row.

```
for ( int number = 1; number <= 9; number++ )
{
     for ( int count = 1; count <= 10; count++)
        cout << count * number << "\t";
     cout << endl;
```

}

Value addition: Source Code

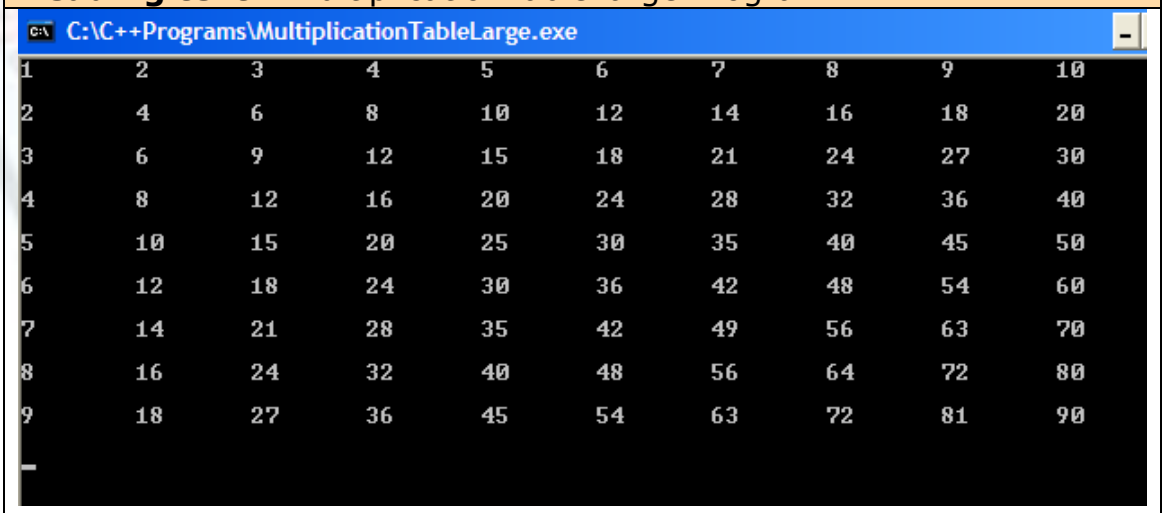
Heading text – MultiplicationTableLarge Program

```
/* This program calculates and prints a multiplication table
for all numbers between 1 and 9 (inclusive).
*/

#include <iostream>
using namespace std;

int main()
{
  for ( int number = 1; number <= 9; number++ )
   {
     for ( int count = 1; count <= 10; count++)
         cout << count * number << "\t";
     cout << endl;
   }
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – MultiplicationTableLarge Program



Source: Self made

3.2.3.2 Printing Patterns

Nested loops can also be used to print patterns. Let us write a program that prints a right triangle in the following pattern:

*

```
**
***
****
*****
```

Notice that there are five rows. In the first row, 1 star is printed, in the second row, 2 stars, in the third row, 3 stars and so on - The number of stars to be printed is the same as the row number!
To print this pattern, we can use a nested loop. The outer loop will iterate as many times as the number of rows. If 5 rows are to be printed, the outer loop index (i) should vary from 1 to 5.
The inner loop will iterate as many numbers of times as the row number – the inner loop should run once for the first row, twice for the second row, and so on. So, we can vary the inner loop index (j) from 1 up to **i**.

The corresponding code fragment is:

```
for ( int i = 1; i <= 5; i++ )
{
    for ( int j = 1; j <= i; j++ )
        cout << "*";
    cout << endl;
}
```

Let us look at a trace of the two nested loops – how the loop indices change for the first three rows.

Loop Indices		**Display Screen**
i	**j**	
1 (1st Row)	1	*
	End of inner loop	Newline is printed
2 (2nd Row)	1	* *
	2	* **
	End of inner loop	Newline is printed
3 (3rd Row)	1	* ** *
	2	* ** **
	3	* ** ***
	End of inner loop	Newline is printed

End of outer loop	

Figure 3.7 Trace of Nested Loop

3.3 Repetition Structures - III

For some programming tasks it is necessary to terminate or interrupt loop iteration before its normal conclusion. In this section you will learn to apply jump statements that allow you to make early exits from loops. In this section, you will also learn to apply repetition structures in some more applications.

3.3.1 Learning Objectives

After reading this chapter you should be able to:

3.3 Terminate and Interrupt loop iterations early.
  3.3,1 Define and apply jump statements – break, continue and return.
3.4 Apply loops to data input validation problems.
3.5 Apply loops to find the maximum and minimum from a list of numbers.
3.6 Apply loops to compute factorial of a number.

3.3.2 Exiting a Loop Iteration Early

In the previous sections, you have learnt to write

- Event controlled loops – Loops that terminate on occurrence of a specific event
- Counter controlled loops – Loops that terminate after a predefined number of iterations.

For solving certain programming tasks, however, you might wish to terminate a loop or a loop iteration before its natural conclusion. For example, to handle an error or some other condition that the conditional expression itself is not testing for.

C++ constructs for interrupting or terminating loops are

- break
- continue
- return

These statements are also called **jump statements** – they allow control to jump to some other part of the code.

3.2.1.1 break

You have already seen the use of a break statement in a switch statement. In a loop, the break statement is used to exit the loop before the exit conditions are met. A break causes the program to jump over all the statements in the loop's block and continue with the statement immediately after the loop.

Let us consider a program that determines whether a number entered as input is a prime number or not.

How do we know if a number is prime? If it is not divisible by any number less than itself, then the number is a prime number. How do we check for divisibility? If x % y is equal to zero, then x is divisible by y (remainder of division x/y is 0).

So, to check for primality, apply a loop to check if the number is divisible by any integer from 2 to number-1.

Also notice that if the number is divisible by an integer, we do not need to continue checking for divisibility – we can break out of the loop.

The C++ fragment of code for checking if number is prime is as below:

```
for (div = 2; div < number; div++)
      if (number % div == 0)
         break;

if ( div == number)
      cout << "The number is prime." <<endl;
else
      cout << "The number is not prime." <<endl;
```

If the loop terminates normally then div became equal to number; this means number is prime.
If the loop terminates because break was executed, then div will not be equal to number; this means number is not prime.

Value addition: Source Code

Heading text – CheckIfPrime

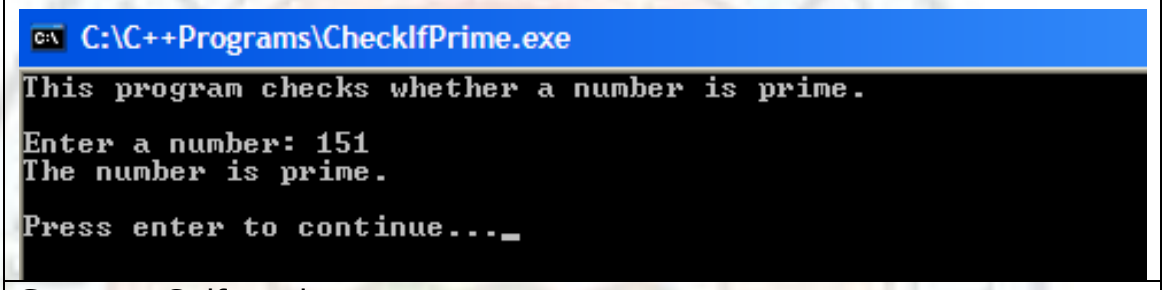
```
/* A not very efficient program that checks whether a number is prime */

#include <iostream>
using namespace std;

int main()
{
   int number;
   int div;

   cout << "This program checks whether a number is prime."<<endl<<endl;

   cout << "Enter a number: ";
   cin >> number;

   if (number == 2){
      cout << "The number is prime." <<endl;
      return 0;
   }

   for (div = 2; div < number; div++)
      if (number % div == 0)
```

```
            break;          // exit  the loop
    if ( div == number)
        cout << "The number is prime." <<endl;
    else
        cout << "The number is not prime." <<endl;

    cout << endl<<"Press enter to continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text - CheckIfPrime



Source: Self made

In a nested loop, break escapes out of the nearest enclosing loop.

```
while (expression1)
{
        //statement1;
        while (expression2)
        {
          …
          break;
          …
        }
        //statement2;
}
```

The execution of break will force execution to exit the inner while loop and resume execution at statement2 in the outer loop.

3.2.1.2 continue

The continue statement forces return to the next test condition of the loop before all the statements in the body of the loop are executed. It serves to bypass certain sections of a loop.
The difference between break and continue statements is that the break statement exits control from the loop but continue statement keeps continuity in loop without executing the statements written after the continue statement.

Consider the program that finds the sum of 5 numbers entered by a user. If the number is not between 1 and 100, it is not added to the sum - the remaining part of the loop is skipped using the continue statement.

```
for (int i = 1; i <= n; i++)
{
  cout << "Enter a number : ";
  cin >> number;
  if( (number < 0) || ( number > 100))
              //check whether the number is between 1 and 100
  {
      cout << endl<< "You have entered a number out of range ..."<<endl <<endl;
      continue; // starts with the beginning of the loop;
                  //does not add number to the sum
   }
   sum = sum + number; // add and store sum to num
 }
```

Value addition: Source Code

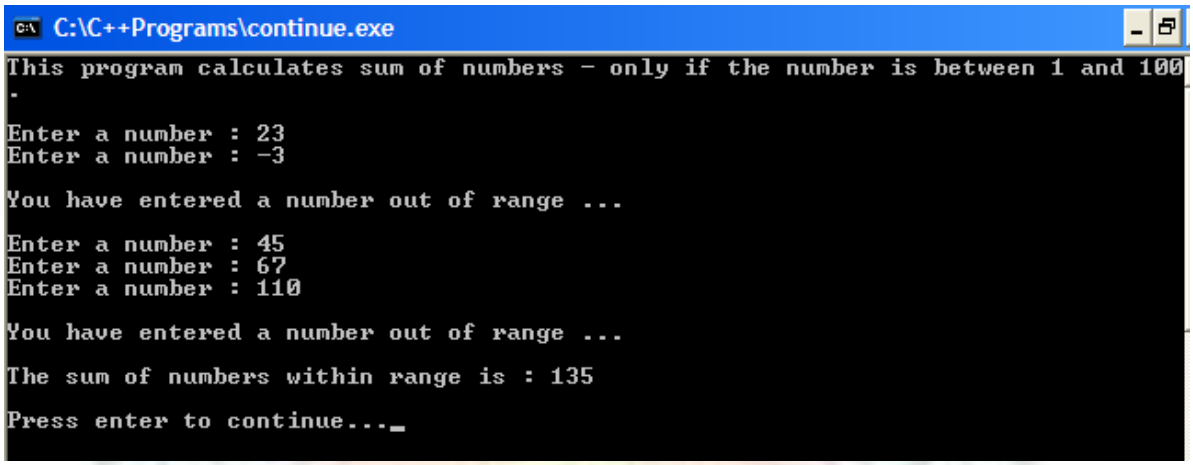
Heading text – ContinueProgram

```
/*This program calculates sum of numbers- only if numbers are between 1 and
100 */

#include <iostream>
using namespace std;

int main()
{
   int n = 5, number, sum = 0;

   cout << "This program calculates sum of numbers - ";
   cout << "only if the number is between 1 and 100." << endl<<endl;

   for (int i = 1; i <= n; i++)
   {
       cout << "Enter a number : ";
       cin >> number;
       if( (number < 0) || ( number > 100))
                   //check whether the number is between 1 and 100
       {
           cout << endl<< "You have entered a number out of range ..."<<endl;
           continue; // starts with the beginning of the loop
        }
        sum = sum + number; // add and store sum to num
    }
    cout << "The sum of numbers within range is : " << sum << endl;
    cout << endl<<"Press enter to continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – ContinueProgram



Source: Self made

3.2.1.3 return

It is also possible to break out of loop prior to its normal termination by using an "extra" return statement.

```
int main()
{
        …
        while (expression)
        {
                …
                if (condition)
                   return 0;
                …
        }
        …
        return 0;
}
```

The return statement in the body of the loop will cause the function (in which the loop is placed) to return immediately to the calling function.

Value addition: Did you Know?

Heading text – The goto statement

Besides the break, continue, and the return statements, there is another statement – the goto statement that can cause control to jump to a specific location anywhere in your code - backward or forward.
The syntax of this statement is:

goto *labelname*;

where *labelname* is a place in the code that is marked with a label name followed by a colon.

For example,

```
cout << "Enter a number : ";
cin >> value;
while (value < 100)
{
   if (value > 90)
        goto end;      // jump to the statement marked with label - end
   cout << "value is =" << value;
   cout << "Enter another number:  ";
   cin >> value;
}
end:  cout << "done";      // statement labeled – end
```

However, the indiscriminate use of goto statements causes difficult to understand and impossible to read programs. Because of this, use of the goto statement is never advised. Use of goto is a sign of bad program design.

Source: Self made

3.2.2 More Applications of Repetition Structures

In this section we will look at some more applications of loops.

3.2.2.1 Validating Data

Whenever you write programs, you must make sure that the data is valid, that is it is of the correct type or format. In most programs, you will need to take in some user input.
For example, you might be writing a program that takes as input the number of students in a class. You should therefore include statements that check the number input and request a new number if the first number is not valid. You can validate input with a post-test loop as below:

```
do {
        cout << "Enter the number of students : ";
        cin >> num_students;
} while (num_students <= 0);
```

The loop will be executed again and again until the user enters a valid input, which in this case is a positive number.
The same validation can also be performed with a post-test loop. The advantage is that if the user enters an invalid input, an error message can be displayed to the user.

```
cout << "Enter the number of students : ";
cin >> num_students;
while (num_students <= 0) {
      cout << "Number of students cannot be zero or negative"<<endl;
      cout << "Please enter a positive number : ";
      cin >> num_students;
 }
```

Data validation helps you avoid pitfalls in your program. For example, if you were calculating average marks of students, ensuring number of students is not zero avoids division by zero in your program.

3.2.2.2 Finding Maximum and Minimum

Finding the maximum or the minimum from a list of user input is also a repetitive task and a loop structure suits this application.
Consider that we have a stack of cards with some integers written on them and we are asked to find out the card with the maximum number. Assume that we can see only one card at a time. To solve this problem, we can take an empty card. This card is going to hold the maximum value – so let it be called the maximum card.
The steps to find the maximum are:

- Initialization: Write the smallest possible value on the maximum card – zero.
- Repetition: Take a card. Compare its value with the one on the maximum card. If the value on the maximum card is smaller, then the value on the newly read card is the new partial maximum. Overwrite the previous maximum value with the new maximum. Repeat this step until all the cards in the stack are exhausted.
- Conclusion: The value on the maximum card is the largest value in the stack of cards.

Similarly, the minimum can be computed. The minimum card is initialized with the largest possible value. The value on the minimum card is overwritten if the newly read number is lesser than the minimum.
The maximum and minimum cards in a program can be simulated with a variable. The corresponding C++ loop to find out the maximum is thus:

```
int maximum = 0, minimum = 999, n = 10;
for (int i = 1; i <=n ; i++)
{
     cout << "Please enter a positive number : ";
     cin >> number;
     if (number > maximum)
        maximum = number;
     if (number < minimum)
        minimum = number;
}
cout << "The largest number in the input is : " << maximum;
cout << "The smallest number in the input is : " << minimum;
```

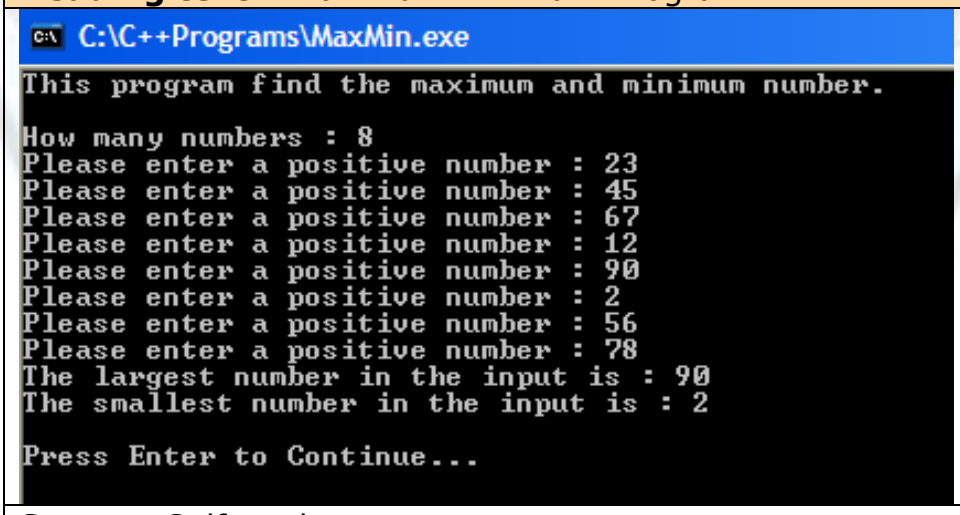
Value addition: Source Code
Heading text – MaximumMinimum Program
/* This program find the maximum and the minimum number from a list of numbers entered as user input. */ #include <iostream> using namespace std;

```
int main()
{
    int n, number, maximum = 0, minimum = 999;

    cout << "This program find the maximum and minimum number." << endl;

    cout << "How many numbers : ";
    cin >> n;
    for (int i = 1; i <=n ; i++)
     {
       cout << "Please enter a positive number : ";
        cin >> number;
        if (number > maximum)
             maximum = number;
       if (number < minimum)
             minimum = number;

    }
    cout << "The largest number in the input is : " << maximum<<endl;
    cout << "The smallest number in the input is : " << minimum<<endl;
    cout << endl<< "Press Enter to Continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – MaximumMInimum Program



Source: Self made

3.2.2.3 Computing Factorial

Recall, that the factorial of an integer is defined as the product of the numbers less than the given integer up to 1. For example, to find the factorial of 6 (written as 6!) we compute it as follows:

6! = 1 * 2 * 3 * 4 * 5 * 6 = 720

The program to compute the factorial is like the summation program, except that instead of accumulating the sum, this time we have to accumulate the products. Since a computer can multiply only two numbers at a time, we need to apply a loop to multiply the numbers from 1 up to the number for which we want the factorial.
The loop will be as follows:

```
for (int index = 1; index <= number ; index++)
    factorial = factorial * index;
```

The variable *index* begins at 1 and is incremented by 1 at every loop iteration until it reaches number.

The variable *factorial* is used to accumulate the partial products – by multiplying it each time with the *index*. Just like in the summation program, we used a variable *sum* initialized to 0, in the factorial program, the variable *factorial* initialized to 1. If *factorial* were initialized to 0, it would cause the factorial of any number to be 0!

Value addition: Source Code

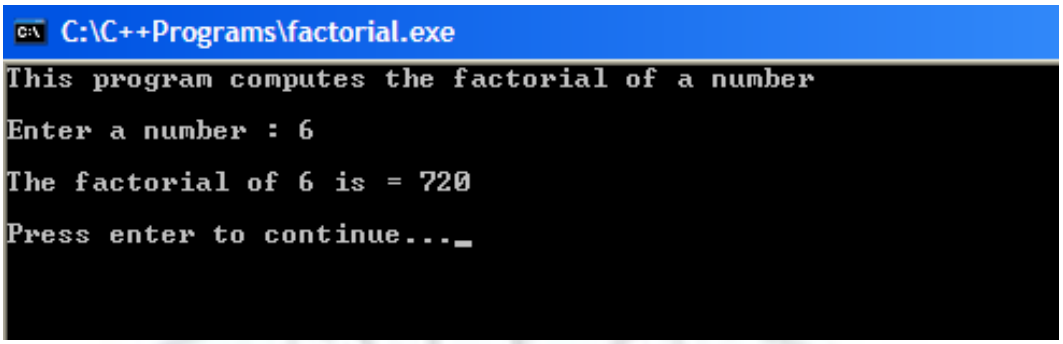
Heading text – Factorial Program

```
/* This program computes the factorial of a number */

#include <iostream>
using namespace std;

int main()
{
    int number, factorial = 1;

    cout << "This program computes the factorial of a number"<<endl<<endl;
    cout << "Enter a number : ";
    cin >> number;
    cout << endl;

    for (int index = 1; index <= number ; index++)
        factorial = factorial * index;

    cout << "The factorial of " << number << " is = "<< factorial;
    cout <<endl<<endl;

    cout << "Press enter to continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in action

Heading text – Factorial Program



Source: Self made

Summary

- Repetition structures or loops contain a block of statements that are executed repeatedly.
- The sequence of statements that is repeated again and again is called the **body** of the loop.
- The **test conditions** that determine whether a loop is entered or exited are written using relational or logical operators.
- A single pass through the loop is called **iteration**.
- There are two basic types of loops: the **pre test loop** and the **post test loop**.
- The pre test loop is an entry controlled loop where the test condition is evaluated before the body of the loop is executed. The pre test loop can be implemented in C++ with the **while** statement.
- The post test Loop is an exit controlled loop where the test condition is evaluated after the body of the loop is executed. The post test loop can be implemented in C++ with the **do-while** statement.
- The body of a pre test loop may never be executed if the test condition is false initially. The body of a post test loop is always executed at least once.
- Applications of loops include:
    - Inputting data until the user enters a sentinel value
    - Counting the number of times a value occurs
    - Computing sums and averages

- A counter controlled loop is a pre test loop that is executed a known fixed number of times.
- The C++ construct for a counter controlled loop is the "for" loop. A counter variable controls the number of iterations of a for loop.
- Its syntax is:

```
for ( count = initial_value; count compared to limit_value; count = count + step)
{
    // body of the loop
}
```

- If the loop increment *step* is positive the body of the for loop is executed until the value of the counter exceeds the loop's limit value.
- If the loop increment *step* is negative the body of the for loop is executed until the value of the counter becomes less than the loop's limit value.
- In **nested looping** a loop is contained entirely within another loop.
- The larger loop in nested looping is called the **outer loop** and the one contained inside is called the **inner loop**. The inner loop is executed for each iteration of the outer loop.
- The break statement in a loop is used to exit the loop before the exit conditions are met. A break causes the program to jump over all the statements in the loop's block and continue with the statement immediately after the loop.
- The continue statement forces return to the next test condition of the loop before all the statements in the body of the loop are executed. It serves to bypass certain sections of a loop.
- The return statement in the body of the loop will cause the function in which the loop is placed to return immediately to the calling function.
- The goto statement causes control to jump to a specific location anywhere in the source code - backward or forward.
- Repetition structures can be applied to solve varied problems like – validating input data, finding the maximum and minimum in a list of numbers and computing the factorial.

Exercises

3.1.1 Differentiate between a "while" and a "do-while" loop.

3.1.2 Predict the output for each of the following:

i)
```
int number = 1;
do
{
        cout << 2*number;
        number = number + 1;
} while (number <= 3);
```

ii)
```
int number = 1;
while (number <= 3)
{
        cout << 2*number;
        number = number + 1;
}
```

3.1.3 The following program fragment is supposed to input numbers from the user and display them until a 0 is entered, but a statement is missing. What is the missing statement and where should it be inserted?

```
cin >> number;
while (number != 0)
{
   cout << number;
}
```

3.1.4 Replace the while loop in the Counting program and the Summation program with the do-while loop.

3.1.5 Write a program to count the number of students when a set of student marks are entered as input.

3.1.6 Write a program to find the average marks scored by a set of students.

3.2.1 Define a counter controlled loop with an example.
3.2.2 Predict the output:

1. for (int count = 10; count < 6; count = count -2)
   cout << count << endl;

2. for (int count = 0; count > 6; count--)
   cout << count << endl;

3.2.3 Write a program that prints multiples of 10 upto 100.
3.2.3 Write a program that prints the following pattern:

55555
4444
333
22
1

3.3.1 What are the differences between break, continue and return?
3.3.2 Predict the output

- ```
  for (i = 10; i > 0; i--)
  {
      cout << i << endl;

      if (i == 4)
          break;
  }
  cout << "done"<< endl;
  ```

- ```
  for (int n=10; n>0; n--)
  {
       if (n==5) continue;
       cout << n << ", ";
  }
  cout << "done" << endl;
  ```

3.3.3 Write a program that accepts a character input and validates that the character is only between a to z.
3.3.4 Write a program to find the second largest number from a list of numbers entered as input.
3.3.5 Write a program to compute C(n,m) where C(n,m) = n! / (m!) * (n-m)! given n and m as input.

Glossary

Accumulator: A variable that stores/ accumulates partial results of arithmetic operations to it.

break: A break causes the program to jump over all the statements in the loop's block and continue with the statement immediately after the loop.
continue: The continue statement forces return to the next test condition of the loop before all the statements in the body of the loop are executed.
Counter: A variable that is incremented in each pass of a loop every time or on satisfying certain conditions – it keeps track of the number of passes around a loop.
Counter Controlled loop: A pre test loop that is executed a known fixed number of times.
Event Controlled Loop: A loop where an event, such as user signaling end of input, controls the number of times the loop is executed.
goto: The goto statement causes control to jump to a specific location anywhere in the code - backward or forward.
Inner loop: The loop that is completely contained inside an outer loop in a nested loop.
Loop index: A counter variable that controls the number of iterations of a for loop.
Nested loop: A loop contained entirely within another loop.
Outer loop: The larger loop in a nested loop that contains another loop.
Post-test Loop: A loop where the test condition is evaluated after the body of the loop is executed.
Pre-test Loop: A loop where the test condition is evaluated before the body of the loop is executed.
return: The return statement causes a function to return immediately to the calling function.
Sentinel: A special input value that cannot occur in the data. It is usually used to mark the end of input. An input of this sentinel causes the loop to terminate.
Variable: A named storage location in memory. Its value can change during program execution. A variable has a data type and must be declared and defined in a program before it can be used.

References

1. Works Cited

2. Suggested Reading

1. B. A. Forouzan and R. F. Gilberg, Computer Science, A structured Approach using C++, Cengage Learning, 2004.

2. R.G. Dromey, How to solve it by Computer, Pearson Education

3. E. Balaguruswamy, Object Oriented Programming with C++ , 4th ed., Tata McGraw Hill

4. G.J. Bronson, A First Book of C++ From Here to There, 3rd ed., Cengage Learning.

5. Graham Seed, An Introduction to Object-Oriented Programming in C++, Springer
6. J. R. Hubbard, Programming with C++ (2nd ed.), Schaum's Outlines, Tata McGraw Hill
7. D S Malik, C++ Programming Language, First Indian Reprint 2009, Cengage Learning
8. R. Albert and T. Breedlove, C++: An Active Learning Approach, Jones and Bartlett India Ltd.

3. Web Links

1. http://www.gurus4pcs.com/CPP/Lesson15.html
2. http://www.functionx.com/cpp/Lesson09.htm
3. http://www.cprogramming.com/tutorial/lesson3.html
4. http://fringe.davesource.com/Fringe/Computers/Languages/CC/Tutorial/tutorial1.html
5. http://www.functionx.com/cpp/Lesson10.htm