# CHAPTER 4

## REPETITION CONTROL STRUCTURE / LOOPING

# Chapter's Outcome

At the end of this chapter, student should be able to:

a) Explain the importance of loops in programs

b) Use ***for*** loops

c) Use ***while*** loops

d) Use ***do while*** loops

e) Use the ***break*** and ***continue*** statements with loops appropriately

f) Use ***nested loops*** effectively when appropriate

g) Use ***flag variables*** and ***sentinel values*** to control indeterminate loops

# Chapter’s Outline

1) Importance of loops in programs
2) For loops
3) While loops
4) Do while loops
5) Break and continue statements
6) Nested loops
7) Flag variables and sentinel values

# The importance of loops in programs

- In order to write a non-trivial computer program, you almost always need to use one or more loops.

- **Loops allow your program to repeat groups of statements a specified number of times.** Loops are classified as a type of iteration (or repetition) structure.

- **It is important to be aware of how many iterations a given loop is expected to perform. If a loop does not iterate a finite number of times, it is considered to be an infinite loop.** Rarely would a programmer ever intentionally incorporate an infinite loop into a program, however it is easy to do so by mistake.

# Components of Looping

- C++ provides the for and while statements as forms for implementing loops.

- The **`for`** statements are usually for **counter-controlled (counting) loops** and the **`while`** statements are generally for **sentinel-controlled loops**.

- All loops are typical of most **Looping Control Structures** in that they have the following **components**:

1) **Loop control variable (LCV)**
2) **A starting point / Initialization of the LCV**
3) **An ending point / Testing the loop repetition condition**
4) **Updating the LCV, and**
5) **the Loop body**

# For Loop

- A **`for` loop** always executes a specific number of iterations.
- **Use a for loop when you know exactly how many times a set of statements must be repeated.**
- A for loop is called a **determinate** or **definite loop** because the programmer knows exactly how many times it will iterate.
- You can mathematically determine the number of iterations by desk checking the logic of the loop.

# For Loop (cont…)

- The keyword, **for**, a set of parentheses is necessary with three expressions as parameters.

```
for (initializing expression; control expression; step expression)
{
            statement or statement block
}
```

- The **initializing expression** sets the counter variable for the loop to its initial value.
- The **control expression** ends the loop at the specified moment.
- The **step expression** (or incrementing expression) changes the counter variable, effectively determining the number of iterations.
- The counter variable often increments by one, but it may increment, decrement, or count in other ways.

# For Loop (cont…)

**Example:**

```
for (i = 1; i <= 3; i++ )
// the counter variable, i, is initialized to the value 1
{
cout << i ;
// the loop will iterate while i is less than or equal to 3

cout << '\n';
// the counter variable, i, increments by 1 on each iteration

}
// two statements are executed in the body of the loop on each      iteration
```

# For Loop (cont…)

- It is possible to initialize the loop variable within the initializing expression.
- But it is NOT recommended for students to initialize loop variables within the loop.
- Students are required to declare loop variables at the top of the function (e.g. main function) where they are used.

```
for (int num = 1; num <= 9; num = num + 3)
{
            cout << num << endl;
}
```

# For Loop (cont…)

- **You can increment the loop variable by more than 1** if you wish. In the following loop, the loop variable is incremented by 3 on each iteration:

```
for (i = 1; i<= 9; i = i + 3)
{
            cout << num << endl;
}
```

- Of course, the compound operator could be used with i += 3 replacing the i = i + 3 incrementing expression.

- It is a **common error** for students to try to use:

```
for (i = 1; i <= 9 ; i + 3)
```

  instead of the **required**

```
for (i = 1; i <= 9; i = i + 3)
```

# For Loop (cont…)

- Some programmers use a for loop like,

```
for (;;)
{
     cout << "Enter a number: ";
     cin >> userInput;

     if (userInput != -99)
     {
          runningTotal += userInput;
     }
     else
     {
          break;
     }

}
```

- **In which case the infinite for loop will run until the break condition is met.**

- Note that technically the initializing, control, and incrementing expressions of a for loop are optional. However, **this is poor coding** since it can lead to logical errors and you **should avoid using break statements** and infinite loops when possible.

# Example of Program

```
 // Purpose - to illustrate the use of the for loop
#include <iostream.h>

int main()
{
        int loopCounter = 0;  // loop variable

        for (loopCounter = 1; loopCounter <= 10; loopCounter++)
        {
                cout << loopCounter << endl;
        }
        // The values 1 through and including 10 are displayed.
        // Notice the use of blank lines above and below the for loop for    // readability.

        for (loopCounter = 1; loopCounter < 10; loopCounter++)
        {
                cout << loopCounter << endl;
        }
        // The values 1 through and including 9 are displayed BUT
        // loopCounter's final value is 10. Can you explain why?
```

# Example of Program (cont…)

```
        for (int i = 0; i != 10; i++)
        {
        cout << i << endl;
        }
        // The loop variable i is declared and initialized to the
        // starting value of 0 within the initializing expression
        // of the for loop. While this is valid syntax, I recommend
        // against it.
        // The control expression i != 10 is very
        // dangerous here. The loop does stop iterating when i
        // increments to the value of 10 but if, for whatever reason,
        // i "skipped over" the value of 10, the loop would iterate
        // forever. In that case, the for loop would be an infinite
        // loop and the C++ program would never stop executing!
        // Does the value of 10 print in this loop?

        return 0;
}  // end of main
```

# `While` Loop

- A **`while`** loop **does not necessarily iterate a specified number of times**.
- Rather, as long as its **control expression** is true, a while loop will continue to iterate.
- This type of loop is very useful in situations where a for loop would be ineffective, particularly when the user is given the opportunity to interact with the program.
- A **`while`** loop is considered to be an **indeterminate** or **indefinite loop** **because usually only at run-time can it be determined how many times it will iterate.**
- A while loop is also considered to be a **top-checking** (or pretest) loop, since the **control expression is located on the first line of code with the while keyword**. That is, if the control expression initially evaluates to FALSE, the loop will not iterate even once.

# `While` Loop (cont...)

- After the keyword, **`while`**, a control expression is necessary.

```
while (control expression)
{
            statement or statement block
}
```

- **The control expression must evaluate to TRUE in order for the while loop to iterate.**

# Example of Program

**Example 1:**

```
        while (num > 100 )
        // the control expression is TRUE if the variable num is     // greater than 100
        {
         cout << num ;
         num = num / 2;
                // the variable num is reassigned a new value during each           //
iteration
        }
```

# Example of Program

**Example 2:**

```
// Purpose - to illustrate the use of the while and do while loops
#include <iostream.h>

int main()
{
int loopVariable = 0;   // loop variable

while (loopVariable < 10)
{
cout << loopVariable << endl;
loopVariable += 1;
}
// The values of 0 through and including 9 are displayed.
// What is the final value of loopVariable?
```

# Example of Program (cont…)

**Example 2:**

```
        loopVariable = 100;

        do
        {
        loopVariable--;
        cout << loopVariable << endl;

        }       while (loopVariable >= 0);

        // Predict what values will be displayed.
        // Notice the use of the decrementing operator.
        // Also notice the necessary semicolon after the
        // the control expression.

        return 0;
} // end of main
```

# Do…While Loop

- As with a while loop, a **do while** loop **does not necessarily iterate a specified number of times.**
- However, it is guaranteed that a **do while** **loop will iterate at least one time because its control expression is placed at the end of the loop.**
- This loop is useful when you want to guarantee at least one iteration of the loop.
- Like a while loop, a **do while** loop is considered to be **an indeterminate or indefinite loop**.
- A **do while** loop is considered to be a **bottom-checking (or post-test) loop**, **since the control expression is located after the body of the loop after the while keyword.** A do while loop is guaranteed to iterate at least once even if the control expression evaluates to FALSE.

# Do…While Loop (cont…)

- The **do** keyword is placed on a line of code by itself at the top of the loop.
- A block of statements follows it with a control expression after the keyword while at the bottom of the loop.

```
do
{
            // body statements would be placed here
}  while (control expression);
```

- **The control expression must evaluate to TRUE in order for the do while loop to iterate after the first time.**

# Example of Program

**Example:**

```
do
{
cout << "Enter a number (0 to quit): ";
cin >> num;
sum = sum + num;
cout << "Current sum is " << sum << '\n';

} while (num != 0 );
```

***Don't forget to include the required semicolon (;) after the control expression.

# The `break` statements with loops

- A **`break`** statement is used to stop the execution of a loop immediately and to continue by executing the statement that comes directly after the loop.
- Only use the **`break`** statement when it is not practical to control the execution of a loop with its control expression. That is, only use a **`break`** statement when absolutely necessary.

```
while (num != -99)
{
     cin >> num;

     if (num == -99)
     {
            break;
     }

     sum += num;
}
```

# The `break` statements with loops (cont…)

- However it would be better to write this loop to **avoid the use of** a **`break`** statement if possible since **`break`** statements found within a loop can be difficult to pick out by those who read over your code.

- **This version** of the same loop **would be better for clarity**.

```
num = 0;

while (num != -99)
{
sum += num;
    cin >> num;
}
```

# The `continue` statements with loops

- A **`continue`** statement is used to stop the execution of the statements in the loop's body on that particular iteration and to continue by starting the next iteration of the loop.

- In the following example, all inputted positive numbers less than or equal to 100 are subtotaled. If a negative value is inputted, it is not added to the subtotal but rather the loop continues with the next iteration. Any value greater than 100 could be entered to terminate the loop.

```
while (num <= 100)
{
     cin >> num;

     if (num < 0)
     {
            continue;
     }

     sum += num;
}
```

# Nested loops

- **A loop** of any kind may be placed **in another loop** (of any kind).
- One must be sure though to entirely encapsulate the inner loop inside of the outer loop, otherwise an error is sure to occur.
- Two loops are considered to be nested loops if one is enclosed within the other.

**Example:**

```
for (i = 0; i <= 10; i++ )
{
     cout << "i is " << i << endl;

     for (j = 10; j >= 0; j-- )
     {
                    cout << "j is " << j << endl;
     }
}
```

# Example of Program

**Example:**

```
// Purpose - to illustrate the use of determinant, indeterminant, and nested loops.
#include <iostream.h>

int main()
{
        int loopVariable = 0;   // loop variable
        int i = 0;  // loop variable

        while (loopVariable < 10)
        {
        cout << loopVariable << endl;
        loopVariable++;

        if (loopVariable == 5)
        {
        break;
        }
        }
        // What values will be printed by the loop above?
```

# Example of Program (cont…)

```
for (i = 1; i < 10; i++)
{
          loopVariable = 1;

          do
          {
                    cout << (loopVariable + i) << endl;
                    loopVariable += 2;

                    if (loopVariable >= 5)
                    {
                              break;
                    }

          } while (loopVariable < 1000);
}
// This is an example of nested loops with a do loop inside
// of a for loop.
// Predict what values will be printed out. Realize though
// that the break command only breaks out of the inner do loop
// each time that it applies.

return 0;
} // end of main
```

# Sentinel variable to control indeterminate loops

- A **sentinel** value is a special value that is used to cause an indeterminate loop to terminate. Often **sentinel** values are inputted by a user to indicate the end of input.

- For example, the **sentinel** value of -99 is used in the loop below to terminate the loop which calculates a bowling average. It should be noted that the -99 is being used as a **sentinel** value since it is impossible in the sport of bowling to bowl a score of -99. Therefore the **sentinel** value will not mistakenly be entered as an inputted bowling score.

```
cout << "Enter a bowling score (-99 to quit): ";
cin >> score;

while (score != -99)
{
     sum = sum + score;
     numGames++;
     cout << "Enter a bowling score (-99 to quit): ";
     cin >> score;
}
cout << "Your average is " << double (sum) / numGames << endl;
```

# Flag variable to control indeterminate loops

- A **flag** variable is a special variable that is used to control an indeterminant loop. A **flag** variable is usually an int that is assigned the values 0 or 1 to indicate false or true respectively.
- Often though a variable of the bool data type is used for **flag** variables.

```
bool choseBook= false;
bool choseCD = false;

do
{
      cout << "1 Book" << endl;
      cout << "2 CD" << endl;
      cout << "3 Exit" << endl;
      cout << "Enter a menu option: ";
      cin >> choice;

      if (choice == 1)
      {
      cout << "You chose book." << endl;
      choseBook = true;
      }
      else if (choice == 2)
      {
               cout << "You chose cd." << endl;
               choseCD = true;
      }

} while (choice != 3 && choseBook != true && choseCD != true);
```

# Example of Program

**Example:**

```
// Purpose - to explore various of methods for using a definite loop with
// user input even though an indefinite loop is more suited for the task.
#include <iostream>
using namespace std;

int main()
{
          int userInput = 0;
          int sum = 0;

          for (;;)
          {
          cout << "Enter an integer (-99 to quit): ";
          cin >> userInput;

          if (userInput != -99)
          sum += userInput;
          else
                    break;
          }
          cout << "The total is " << sum << endl;
          // ****************************************
```

# Example of Program (cont…)

```
sum = 0;

for (userInput = 0; userInput != -99; sum += userInput)
{
          cout << "Enter an integer (-99 to quit): ";
          cin >> userInput;
}
cout << "The total is " << sum + 99 << endl;
// ****************************************
sum = 0;

for (userInput = 0; userInput != -99; sum += userInput)
{
          cout << "Enter an integer (-99 to quit): ";
          cin >> userInput;

          if (userInput == -99)
          break;
}
cout << "The total is " << sum << endl;
return 0;
} // end of main
```