Chapter 5: Looping

Starting Out with C++
Early Objects
Eighth Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley
is an imprint of
PEARSON

Topics

5.1 Introduction to Loops: The **`while`** Loop

5.2 Using the **`while`** loop for Input Validation

5.3 The Increment and Decrement Operators

5.4 Counters

5.5 The **`do-while`** loop

5.6 The **`for`** loop

5.7 Keeping a Running Total

Topics (continued)

5.8 Sentinels

5.9 Deciding Which Loop to Use

5.10 Nested Loops

5.11 Breaking Out of a Loop

5.12 Using Files for Data Storage

5.13 Creating Good Test Data

5.1 Introduction to Loops: The **`while`** Loop

- Loop: part of program that may execute > 1 time (*i.e.*, it repeats)
- **`while`** loop format:

```
while (condition)
{   statement(s);
}
```

No ; here

- The **`{ }`** can be omitted if there is only one statement in the body of the loop

How the **`while`** Loop Works

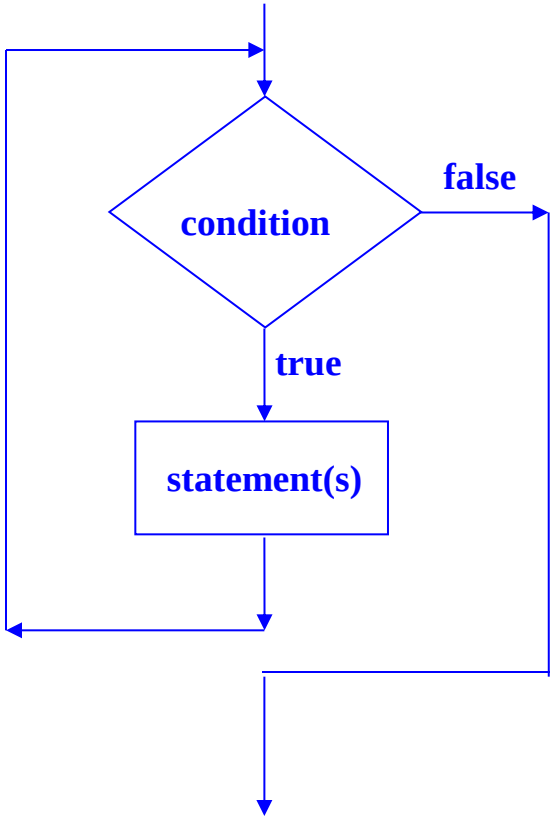
```
while (condition)
{   statement(s);
}
```

`condition` is evaluated

- if it is true, the ***`statement(s)`*** are executed, and then ***`condition`*** is evaluated again
- if it is false, the loop is exited

An iteration is an execution of the loop body

`while` Loop Flow of Control



`while` Loop Example

```
int val = 5;
while (val >= 0)
{    cout << val << "   ";
     val = val - 1;
}
```

- produces output:

```
 5   4   3   2   1   0
```

`while` Loop is a Pretest Loop

- **`while`** is a pretest loop (***`condition`*** is evaluated <u>before</u> the loop executes)
- If the condition is initially false, the statement(s) in the body of the loop are never executed
- If the condition is initially true, the statement(s) in the body will continue to be executed until the condition becomes false

Exiting the Loop

- The loop must contain code to allow ***condition*** to eventually become **false** so the loop can be exited
- Otherwise, you have an infinite loop (*i.e.*, a loop that does not stop)
- Example infinite loop:

```
x = 5;
while (x > 0)      // infinite loop because
   cout << x;      // x is always > 0
```

Common Loop Errors

- Don't put **;** immediately after ***(condition)***
- Don't forget the { } :

```
int numEntries = 1;
while (numEntries <=3)
    cout << "Still working … ";
    numEntries++; // not in the loop body
```

- Don't use = when you mean to use ==

```
while (numEntries = 3)  // always true
{
    cout << "Still working … ";
    numEntries++;
}
```

`while` Loop Programming Style

- Loop body statements should be indented

- Align { and } with the loop header and place them on lines by themselves

Note: The conventions above make the program more understandable by someone who is reading it. They have no effect on how the the program compiles or executes.

5.2 Using the `while` Loop for Input Validation

Loops are an appropriate structure for validating user input data

1. Prompt for and read in the data.
2. Use a `while` loop to test if data is valid.
3. Enter the loop only if data is <u>not</u> valid.
4. Inside the loop, display error message and prompt the user to re-enter the data.
5. The loop will not be exited until the user enters valid data.

Input Validation Loop Example

```
cout << "Enter a number (1-100) and"
     << " I will guess it. ";
cin  >> number;

while (number < 1 || number > 100)
{  cout << "Number must be between 1 and 100."
        << " Re-enter your number. ";
   cin  >> number;
}
// Code to use the valid number goes here.
```

5.3 The Increment and Decrement Operators

- **Increment – increase value in variable**

  `++` adds one to a variable

  `val++;` is the same as **`val = val + 1;`**

- **Decrement – reduce value in variable**

  `--` subtracts one from a variable

  `val--;` is the same as **`val = val – 1;`**

- can be used in prefix mode (before) or postfix mode (after) a variable

Prefix Mode

- **`++val`** and **`--val`** increment or decrement the variable, *then* return the new value of the variable.
- It is this returned new value of the variable that is used in any other operations within the same statement

Prefix Mode Example

```
int x = 1, y = 1;

x = ++y;            // y is incremented to 2
                    // Then 2 is assigned to x
cout << x
  << "  " << y; // Displays 2  2

x = --y;            // y is decremented to 1
                    // Then 1 is assigned to x
cout << x
  << "  " << y; // Displays 1 1
```

Postfix Mode

- **`val++`** and **`val--`** return the old value of the variable, *then* increment or decrement the variable
- It is this returned old value of the variable that is used in any other operations within the same statement

Postfix Mode Example

```
int x = 1, y = 1;

x = y++;            // y++ returns a 1
                    // The 1 is assigned to x
                    // and y is incremented to 2
cout << x
  << "  " << y; // Displays 1  2

x = y--;            // y-- returns a 2
                    // The 2 is assigned to x
                    // and y is decremented to 1
cout << x
  << "  " << y; // Displays 2 1
```

Increment & Decrement Notes

- Can be used in arithmetic expressions

```
result = num1++ + --num2;
```

- Must be applied to something that has a location in memory. Cannot have

```
result = (num1 + num2)++; // Illegal
```

- Can be used in relational expressions

```
if (++num > limit)
```

- Pre- and post-operations will cause different comparisons

5.4 Counters

- Counter: variable that is incremented or decremented each time a loop repeats
- Can be used to control execution of the loop (loop control variable)
- Must be initialized before entering loop
- May be incremented/decremented either inside the loop or in the loop test

Letting the User Control the Loop

- Program can be written so that user input determines loop repetition
- Can be used when program processes a list of items, and user knows the number of items
- User is prompted before loop. Their input is used to control number of repetitions

User Controls the Loop Example

```
int num, limit;
cout << "Table of squares\n";
cout << "How high to go? ";
cin  >> limit;
cout << "\n\nnumber square\n";
num = 1;
while (num <= limit)
{  cout << setw(5) << num << setw(6)
        << num*num << endl;
   num++;
}
```

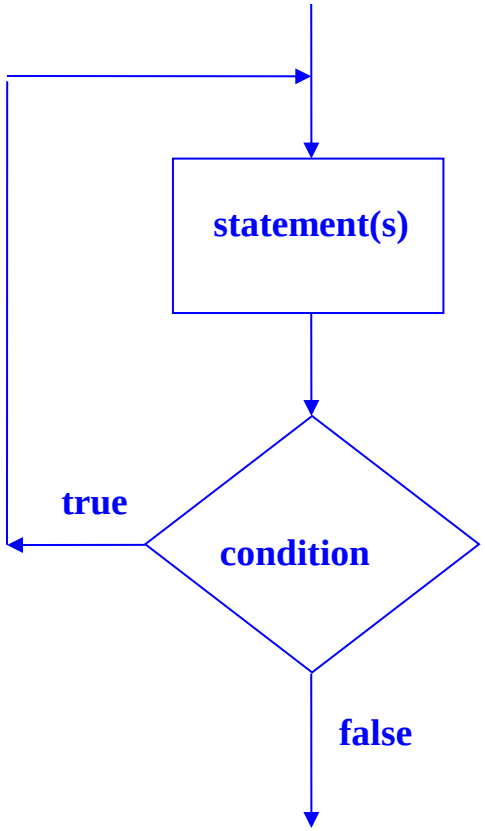
5.5 The **do-while** Loop

- **do-while**: a post test loop (***condition*** is evaluated after the loop executes)
- Format:

```
do
{    1 or more statements;
} while (condition);
```

Notice the required ;

`do-while` Flow of Control



`do-while` Loop Notes

- Loop always executes at least once
- Execution continues as long as ***`condition`*** is **`true`**; the loop is exited when ***`condition`*** becomes **`false`**
- { } are required, even if the body contains a single statement
- **;** after ***`(condition)`*** is also required

`do-while` and Menu-Driven Programs

- do-while can be used in a menu-driven program to bring the user back to the menu to make another choice
- To simplify the processing of user input, use the **`toupper`** ('to upper') or **`tolower`** (to lower') function

Menu-Driven Program Example

```
do {
   // code to display menu
   // and perform actions
   cout << "Another choice? (Y/N) ";
} while (choice ==Y'||choice =='y');
```

|| means OR

The condition could be written as

```
(toupper(choice) == 'Y');
```

or as

```
(tolower(choice) == 'y');
```

makes the input go to upper case or lower case whichever is the test.

5.6 The `for` Loop

- Pretest loop that executes zero or more times
- Useful for counter-controlled loop

- Format:

```
for( initialization; test; update )
{    1 or more statements;
}
```

Required ;

No ; goes here

`for` Loop Mechanics

Step 1: Perform the initialization expression.

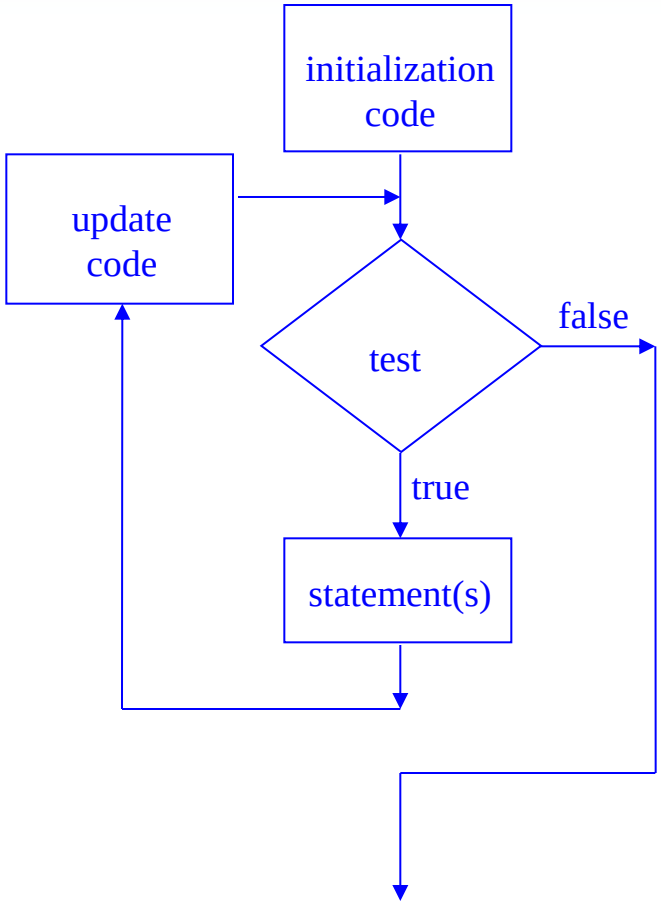
Step 2: Evaluate the test expression.
If it is true, go to step 3.
Otherwise, terminate the loop.

```
for (count = 1; count <= 5; count++)
{  cout << "Hello" << endl;
}
```

Step 3: Execute the body of the loop.

Step 4: Perform the update expression.
Then go back to step 2.

`for` Loop Flow of Control



`for` Loop Example

```
int sum = 0, num;
for (num = 1; num <= 10; num++)
   sum += num;
cout << "Sum of numbers 1 - 10 is "
                << sum << endl;
```

`for` Loop Notes

- If ***`test`*** is false the first time it is evaluated, the body of the loop will not be executed
- The update expression can increment or decrement by any amount
- Variables used in the initialization section should not be modified in the body of the loop

`for` Loop Modifications

- Can define variables in initialization code
  - Their scope is the `for` loop
- Initialization and update code can contain more than one statement
  - Separate the statements with commas
- Example:

```
for (int sum = 0, num = 1; num <= 10; num++)
    sum += num;
```

More **for** Loop Modifications

(These are NOT Recommended)

- Can omit ***initialization*** if already done

```
int sum = 0, num = 1;
for (; num <= 10; num++)
    sum += num;
```

- Can omit ***update*** if done in loop

```
for (sum = 0, num = 1; num <= 10;)
    sum += num++;
```

- Can omit ***test*** – may cause an infinite loop

```
for (sum = 0, num = 1; ; num++)
    sum += num;
```

- Can omit loop body if all work is done in header

5.7 Keeping a Running Total

- running total: accumulated sum of numbers from each repetition of loop
- accumulator: variable that holds running total

```
int sum = 0, num = 1; // sum is the
while (num <= 10)     // accumulator
{   sum += num;
    num++;
}
cout << "Sum of numbers 1 - 10 is "
     << sum << endl;
```

5.8 Sentinels

- sentinel: value in a list of values that indicates end of the list
- Special value that cannot be confused with a valid value, *e.g.*, **`-999`** for a test score
- Used to terminate input when user may not know how many values will be entered

Sentinel Example

```
int total = 0;
cout << "Enter points earned "
     << "(or -1 to quit): ";
cin  >> points;
while (points != -1) // -1 is the sentinel
{
   total += points;
   cout << "Enter points earned: ";
   cin  >> points;
}
```

5.9 Deciding Which Loop to Use

- **`while`**: pretest loop (loop body may not be executed at all)
- **`do-while`**: post test loop (loop body will always be executed at least once)
- **`for`**: pretest loop (loop body may not be executed at all); has initialization and update code; is useful with counters or if precise number of repetitions is known

5.10 Nested Loops

- A nested loop is a loop inside the body of another loop
- Example:

```
for (row = 1; row <= 3; row++)
{
   for (col = 1; col <= 3; col++)
   {
      cout << row * col << endl;
   }
}
```

outer loop

inner loop

Notes on Nested Loops

- Inner loop goes through all its repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops. In previous example, inner loop repeats 9 times

5.11 Breaking Out of a Loop

- Can use **`break`** to terminate execution of a loop
- Use sparingly if at all – makes code harder to understand
- When used in an inner loop, terminates that loop only and returns to the outer loop

The `continue` Statement

- Can use `continue` to go to end of loop and prepare for next repetition
  - `while` and `do-while` loops go to test and repeat the loop if test condition is true
  - `for` loop goes to update step, then tests, and repeats loop if test condition is true
- Use sparingly – like `break`, can make program logic hard to follow

5.12 Using Files for Data Storage

- We can use a file instead of monitor screen for program output
- Files are stored on secondary storage media, such as disk
- Files allow data to be retained between program executions
- We can later use the file instead of a keyboard for program input

File Types

- Text file – contains information encoded as text, such as letters, digits, and punctuation. Can be viewed with a text editor such as Notepad.
- Binary file – contains binary (0s and 1s) information that has not been encoded as text. It cannot be viewed with a text editor.

File Access – Ways to Use the Data in a File

- Sequential access – read the 1st piece of data, read the 2nd piece of data, …, read the last piece of data. To access the n-th piece of data, you have to retrieve the preceding n pieces first.
- Random (direct) access – retrieve any piece of data directly, without the need to retrieve preceding data items.

What is Needed to Use Files

1. Include the **`fstream`** header file
2. Define a file stream object
   - **`ifstream`** for input from a file

     ```
     ifstream inFile;
     ```

   - **`ofstream`** for output to a file

     ```
     ofstream outFile;
     ```

Open the File

3. Open the file

- Use the **`open`** member function

```
inFile.open("inventory.dat");
outFile.open("report.txt");
```

- Filename may include drive, path info.
- Output file will be created if necessary; existing output file will be erased first
- Input file must exist for **`open`** to work

Use the File

4. Use the file

- Can use output file object and **<<** to send data to a file

```
outFile << "Inventory report";
```

- Can use input file object and **>>** to copy data from file to variables

```
inFile >> partNum;
inFile >> qtyInStock >> qtyOnOrder;
```

Close the File

5. Close the file

- Use the **`close`** member function

```
inFile.close();
outFile.close();
```

- Don't wait for operating system to close files at program end
  - There may be limit on number of open files
  - There may be buffered output data waiting to be sent to a file that could be lost

Input File – the Read Position

- Read Position – location of the next piece of data in an input file
- Initially set to the first byte in the file
- Advances for each data item that is read. Successive reads will retrieve successive data items.

Using Loops to Process Files

- A loop can be used to read data from or write data to a file
- It is not necessary to know how much data is in the file or will be written to the file
- Several methods exist to test for the end of the file

Using the >> Operator to Test for End of File (EOF) on an Input File

- The stream extraction operator (**>>**) returns a true or false value indicating if a read is successful
- This can be tested to find the end of file since the read "fails" when there is no more data
- Example:

```
while (inFile >> score)
   sum += score;
```

File Open Errors

- An error will occur if an attempt to open a file for input fails:
  - File does not exist
  - Filename is misspelled
  - File exists, but is in a different place
- The file stream object is set to true if the open operation succeeded. It can be tested to see if the file can be used:

```
if (inFile)
{
  // process data from file
}  else
   cout << "Error on file open\n";
```

User-Specified Filenames

- Program can prompt user to enter the names of input and/or output files. This makes the program more versatile.
- Filenames can be read into string objects. The C-string representation of the string object can then be passed to the open function:

```
cout << "Which input file? ";
cin >> inputFileName;
inFile.open(inputFileName.c_str());
```

5.13 Creating Good Test Data

- When testing a program, the quality of the test data is more important than the quantity.
- Test data should show how different parts of the program execute
- Test data should evaluate how program handles:
  - normal data
  - data that is at the limits the valid range
  - invalid data

Chapter 5: Looping

The End!!