4.1 Introduction to Repetition Structures

- Programs often need to repeat a specific task
  - Disadvantages of duplicated code
    - Makes the program large
    - Time consuming
    - Any changes to one part must be repeated in all parts
  - Repetition structures (*loops*) provide a better way
    - Write code once and computer repeats it
    - Two types of loops:
      - *Condition-Controlled loop* (uses a true/false condition to repeat)
      - *Count-Controlled loop* (repeats a specific number of times)
4.2 Condition-Controlled Loops: The \textbf{while} and \textbf{do-while} Loops

- The \textbf{while} loop gets its name from the way it works:
  - \textit{While an expression is true, do some task}

- Has two parts:
  - (1) Boolean expression that is tested for true/false
  - (2) Statement(s) that are repeated as long as the Boolean expression is true

- Each time the loop executes its statement(s) is called an \textit{iteration} of the loop

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{logic_of_while_loop}
\caption{The logic of a \textbf{while} loop.}
\end{figure}

Copyright © 2013 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
4.2 Condition-Controlled Loops: The while and do-while Loops

Here is the general format of the while loop:

```c
while (BooleanExpression)
{
    statement;
    statement;
    etc;
}
```

Copyright © 2013 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
4.2 Condition-Controlled Loops: The while and do-while Loops

• The while loop is a pretest loop
  – It tests its condition before performing an iteration
    • Condition is usually set prior to loop to make sure it executes at least once
  – It will never execute if its condition is false to start with
    • This is an important characteristic
    • Exactly what you want to happen in some programs
      – Program 4-2, for example
4.2 Condition-Controlled Loops: The while and do-while Loops

- Loops must contain a way to terminate within themselves
  - Something inside the loop must eventually make the test condition false
- If a loop does not have a way of stopping, it is called an infinite loop
  - Continues to repeat until the program is interrupted
  - Usually occurs by accident and should be avoided
- Program 4-3, for example
4.2 Condition-Controlled Loops: The while and do-while Loops

- The do-while loop is a posttest loop
  - It tests its condition after performing an iteration
- It always performs at least one iteration, even if its expression is false to begin with
4.2 Condition-Controlled Loops: The while and do-while Loops

Here is the general format of the do-while loop:

```
int number = 1;
while (number < 0)
{
    cout << number << endl;
}
```

In the `while` loop, the `cout` statement will not execute.

```
int number = 1;
do
{
    cout << number << endl;
} while (number < 0);
```

In the `do-while` loop, the `cout` statement will execute once.
4.3 The Increment and Decrement Operators

- To *increment* a variable is to *increase* its value
- To *decrement* a variable is to *decrease* its value
- Special unary operators are provided to do this
  - `++`
    - Pronounced “plus-plus”
    - Increments a variable by 1 (`num++;`)
  - `--`
    - Pronounced “minus-minus”
    - Decrements a variable by 1 (`num--;`)

These operators can be used in two modes:
- *Postfix mode* - operator on the right (after operand)
  - `num++;`
  - `num--;`
- *Prefix mode* - operator on the left (before operand)
  - `++num;`
  - `--num;`

With simple statements, either mode is fine
- Be careful mixing modes with complex statements!
4.4 Count-Controlled Loops: The **for** Loop

- A count-controlled loop iterates a specific number of times
  - A *counter variable* stores the number of iterations
  - Three actions are performed using this variable
    - (1) *Initialization* – counter is initialized to starting value
    - (2) *Test* – counter is compared to maximum value
    - (3) *Increment* – counter is incremented
- The **for** loop is specifically designed to handle this task
4.4 Count-Controlled Loops: The for Loop

Here is the general format of the for loop:

```c++
for (InitializationExpression; TestExpression; IncrementExpression)
{
    statement;
    statement;
    etc.
}
```

- Program 4-4, for example

Program 4-4 (ForLoop.cpp)

```c++
1 // This program demonstrates the for loop.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    int counter;
8    const int MAX_VALUE = 5;
9    for (counter = 0; counter < MAX_VALUE; counter++)
10    {
11        cout << "Hello world" << endl;
12    }
13    return 0;
14 }
```

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```
4.4 Count-Controlled Loops: The \texttt{for} Loop

- The counter variable may also be declared in the initialization expression of a \texttt{for} loop

```cpp
const int MAX_VALUE = 5;
for (int counter = 0; count < MAX_VALUE; ++counter)
{
    cout << "Hello" << endl;
}
```

The variable's \texttt{scope} is limited to the loop

```cpp
const int MAX_VALUE = 5;
for (int count = 0; count < MAX_VALUE; ++count)
{
    cout << "Hello" << endl;
    cout << "The value of the counter is " << count << endl;
}
```

Error!
4.4 Count-Controlled Loops: The \texttt{for} Loop

- Primary purpose of the counter variable:
  - Stores the number of times the loop has iterated
  - Can be helpful in calculations or other tasks
    - Program 4-5, for example

\begin{tabular}{|c|c|}
\hline
Number & Square \\
\hline
1 & 1 \\
2 & 4 \\
3 & 9 \\
4 & 16 \\
5 & 25 \\
6 & 36 \\
7 & 49 \\
8 & 64 \\
9 & 81 \\
10 & 100 \\
\hline
\end{tabular}
4.4 Count-Controlled Loops: The \textbf{for} Loop

- Other forms of the update expression
  - Displaying even numbers from 2 through 100:

\begin{verbatim}
for (int counter = 2; counter <= 100; counter += 2) 
cout << counter << endl;
\end{verbatim}

- Counting backward from 10 to 0:

\begin{verbatim}
for (int counter = 10; counter >= 0; counter--)
cout << counter << endl;
\end{verbatim}

- Program 4-6, for example

Copyright © 2013 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
4.4 Count-Controlled Loops: The *for* Loop

- In many cases, the programmer knows the exact number of iterations a loop must perform
- Sometimes the programmer needs to let the user decide
- How is it done?
  - Get input from the user
  - Use input variable as the counter’s ending value
- Program 4-7, for example

```
Program 4-7
(UserSquares.cpp)
1 // This program prints a table of numbers
2 // and their squares.
3 // Includes <iostream>
4 using namespace std;
5 %
6 int main()
7 { int upperLimit;
8     // Set the upper limit.
9     cout << "This program displays numbers, starting at 1, " << endl;
10    cout << "and their squares. How high should I get? " << endl;
11    cin >> upperLimit;
12    // Display the table headings.
13    cout << "Number " << setw(10) << "Square" << endl;
14    // Display the values.
15    for (int number = 1; number <= upperLimit; number++)
16        cout << number << " " << number * number << endl;
17    return 0;
18 %
```

Program Output with Input Shown in Bold

```
This program displays numbers, starting at 1,
and their squares. How high should I get? (upperLimit)
1 1
2 4
3 9
4 16
5 25
```
4.5 Calculating a Running Total

- Many programming tasks require you to total a series of numbers
  - Typically two elements used to do this:
    - (1) A loop that reads each number in the series
    - (2) A variable that accumulates the total of numbers
  - Loop keeps a *running total* as it reads each number
  - Variable used to accumulate total is called an *accumulator*
    - Program 4-8, for example
4.5 Calculating a Running Total

```
Program 4-6

1 // This program calculates the sum of a series of numbers.
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     // Variables
7     int number; // To hold each number
8     total = 0; // Accumulator, initialized with 0
9
10     // Constant for the number of numbers
11     const int MAX_NUMS = 5;
12
13     // Explain the program's purpose.
14     // Get the numbers and accumulate them.
15     for (int counter = 0; counter < MAX_NUMS; counter++)
16     {
17         cout << "Enter a number: ";
18         cin >> number;
19         total += number;
20     } // Display the total.
21     cout << "The total is: " << total << endl;
22     return 0;
23 }
```.

4.6 Nested Loops

- **Nested loop** - a loop that is inside another loop
  - An analog clock, for example:
    - The hour hand makes a revolution
      - Which is equal to 12 revolutions of the minute hand
    - The minute hand makes a revolution
      - Which is equal to 60 revolutions for the second hand
    - For every revolution of the hour hand
      - The second hand makes 720 revolutions (60 x 12)
  - How about a digital clock?
4.6 Nested Loops

An Example of a Simulated Digital Clock

```cpp
const int MAX SECONDS = 60;
const int MAX_MINUTES = 60;
const int MAX HOURS = 24;
for (int hours = 0; hours < MAX HOURS; hours++)
{
    for (int minutes = 0; minutes < MAX_MINUTES; minutes++)
    {
        for (int seconds = 0; seconds < MAX_SECONDS; seconds++)
        {
            cout << hours << ":" << minutes << ":" << seconds << endl;
        }
    }
}
```

A few points about nested loops:

- The inner loop goes through all its iterations for a single iteration of the outer loop
- Inner loops complete iterations faster than outer loops
- To get the total number of iterations
  - Multiply the number of iterations of all the loops
4.6 Nested Loops

• An interesting way to learn about nested loops?
  – Use nested loops to display patterns on the screen!

• Examples:
  – Program 4-9
    • Rectangular Pattern
  – Program 4-10
    • Triangle Pattern
  – Program 4-11
    • Stair Step Pattern
4.6 Nested Loops

```
Program 4-10 (TrianglePattern.cpp)
1  // This program displays a triangle pattern.
2  #include <iostream>
3  using namespace std;
4
5 int main()
6 {
7  cout << "triangle pattern:
8  
9  
10  
11  
12  
13  
14  
15  
16  return 0;
17 }
```

Program Output:
```
*
**
***
****
*****
********
```

Copyright © 2013 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

4.6 Nested Loops

```
Program 4-11 (StairStepPattern.cpp)
1  // This program displays a stair-step pattern.
2  #include <iostream>
3  using namespace std;
4
5 int main()
6 {
7  cout << "stair-step pattern:
8  
9  
10  
11  
12  
13  
14  
15  
16  return 0;
17 }
```

Program Output:
```
*
*
*
```

Copyright © 2013 Pearson Education, Inc. Publishing as Pearson Addison-Wesley