Objectives

• In this chapter, you will:
  – Learn about repetition (looping) control structures
  – Explore how to construct and use counter-controlled, sentinel-controlled, flag-controlled, and EOF-controlled repetition structures
  – Examine break and continue statements
  – Discover how to form and use nested control structures

Objectives (cont’d.)

• In this chapter, you will (cont’d.):
  – Learn how to avoid bugs by avoiding patches
  – Learn how to debug loops

Why Is Repetition Needed?

• Repetition allows efficient use of variables
• Can input, add, and average multiple numbers using a limited number of variables
• For example, to add five numbers:
  – Declare a variable for each number, input the numbers and add the variables together
  – Create a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers

while Looping (Repetition) Structure

• Syntax of the while statement:
  
  ```
  while (expression)
  statement
  ```

  • statement can be simple or compound
  • expression acts as a decision maker and is usually a logical expression
  • statement is called the body of the loop
  • The parentheses are part of the syntax
while Looping (Repetition) Structure (cont’d.)

• \( i \) in previous Example 5-1 is called the loop control variable (LCV)
• Infinite loop: continues to execute endlessly
  – Avoided by including statements in loop body that assure the exit condition is eventually false

Case 1: Counter-Controlled while Loops

• When you know exactly how many times the statements need to be executed
  – Use a counter-controlled while loop
    
    counter = 0; //initialize the loop control variable
    while (counter < \( M \)) //test the loop control variable
    {
      ...
      counter++; //update the loop control variable
    ...}

Case 2: Sentinel-Controlled while Loops

• Sentinel variable is tested in the condition
• Loop ends when sentinel is encountered
  
  cin >> variable; //initialize the loop control variable
  while (variable != sentinel) //test the loop control variable
  {
    ...
    cin >> variable; //update the loop control variable
    ...
  }
Example 5-5: Telephone Digits

Example 5-5 provides an example of a sentinel-controlled loop (Pg: 278)

- The program converts uppercase letters to their corresponding telephone digit
  - case 'A': case 'B': case 'C': cout << 2 << endl;
  - case 'D': case 'E': case 'F': cout << 3 << endl;
  - case 'G': case 'H': case 'I': cout << 4 << endl;
  - case 'J': case 'K': case 'L': cout << 5 << endl;

Case 3: Flag-Controlled while Loops

- Flag-controlled while loop: uses a bool variable to control the loop

```cpp
found = false; //initialise the loop control variable
while (!found) //test the loop control variable
{
    //
    if (expression)
    found = true; //update the loop control variable
    //
}
```

Number Guessing Game

Example 5-6 implements a number guessing game using a flag-controlled while loop (Pg: 282)

Uses the function rand of the header file cstdlib to generate a random number

```cpp
srand(time(0));
Num = rand() % 100;
```

- rand() returns an int value between 0 and 32767
- To convert to an integer >= 0 and < 100:
  ```cpp
  rand() % 100
  ```

Case 4: EOF-Controlled while Loops

- End-of-file (EOF) - controlled while loop: when it is difficult to select a sentinel value
- The logical value returned by cin can determine if there is no more input

```cpp
end_of_file
```

Case 4: EOF-Controlled while Loops (cont’d.)

```cpp
EXAMPLE 5-7
The following code uses an EOF-controlled while loop to find the sum of a set of numbers:
int sum = 0;
int num;
cin >> num;
while (cin)
{
    sum = sum + num; //Add the number to sum
    cin >> num; //Get the next number
}
cout << "sum = " << sum << endl;
```

eof Function

- The function eof can determine the end of file status
- eof is a member of data type istream
- Syntax for the function eof:
  ```cpp
  istreamVar.eof()
  ```

- istreamVar is an input stream variable, such as cin
More on Expressions in while Statements

- The expression in a while statement can be complex
  - Example:
    ```
    while ((noOfGuesses < 5) && (!isGuessed))
    {
        ...
    }
    ```

Programming Example: Fibonacci Number

- Consider the following sequence of numbers:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Called the Fibonacci sequence
- Given the first two numbers of the sequence (say, a1 and a2)
  - \( n^{th} \) number \( a_n \), \( n \geq 3 \), of this sequence is given by:
    \[
    a_n = a_{n-1} + a_{n-2}
    \]

Programming Example: Fibonacci Number (cont’d.)

- Fibonacci sequence
  - \( n^{th} \) Fibonacci number
    - \( a_2 = 1 \)
    - \( a_1 = 1 \)
    - Determine the \( n^{th} \) number \( a_n \), \( n \geq 3 \)

Programming Example: Fibonacci Number (cont’d.)

- Suppose \( a_2 = 6 \) and \( a_3 = 3 \)
  - \( a_3 = a_2 + a_1 = 6 + 3 = 9 \)
  - \( a_4 = a_3 + a_2 = 9 + 6 = 15 \)
- Write a program that determines the \( n^{th} \) Fibonacci number, given the first two numbers

Programming Example: Input and Output

- Input: first two Fibonacci numbers and the desired Fibonacci number
- Output: \( n^{th} \) Fibonacci number

Programming Example: Problem Analysis and Algorithm Design

- Algorithm:
  - Get the first two Fibonacci numbers
  - Get the desired Fibonacci number
    - Get the position, \( n \), of the number in the sequence
    - Calculate the next Fibonacci number
      - Add the previous two elements of the sequence
    - Repeat Step 3 until the \( n^{th} \) Fibonacci number is found
  - Output the \( n^{th} \) Fibonacci number
Programming Example: Variables

```cpp
int previous1; //variable to store the first Fibonacci number
int previous2; //variable to store the second Fibonacci number
int current; //variable to store the current Fibonacci number
int counter; //loop control variable
int nthFibonacci; //variable to store the desired Fibonacci number
```

Programming Example: Main Algorithm

- Prompt the user for the first two numbers—that is, `previous1` and `previous2`
- Read (input) the first two numbers into `previous1` and `previous2`
- Output the first two Fibonacci numbers
- Prompt the user for the position of the desired Fibonacci number

Programming Example: Main Algorithm (cont'd.)

- Read the position of the desired Fibonacci number into `nthFibonacci`
  - if (`nthFibonacci` == 1)
    The desired Fibonacci number is the first Fibonacci number; copy the value of `previous1` into `current`
  - else if (`nthFibonacci` == 2)
    The desired Fibonacci number is the second Fibonacci number; copy the value of `previous2` into `current`

Programming Example: Main Algorithm (cont’d.)

- else calculate the desired Fibonacci number as follows:
  - Start by determining the third Fibonacci number
  - Initialize `counter` to 3 to keep track of the calculated Fibonacci numbers.
  - Calculate the next Fibonacci number, as follows:
    ```cpp
    current = previous2 + previous1;
    ```

Programming Example: Main Algorithm (cont’d.)

- (cont’d.)
  - Assign the value of `previous2` to `previous1`
  - Assign the value of `current` to `previous2`
  - Increment `counter`
  - Repeat until Fibonacci number is calculated:
    ```cpp
    while (counter <= nthFibonacci)
    {
        current = previous2 + previous1;
        previous1 = previous2;
        previous2 = current;
        counter++;
    }
    ```

Programming Example: Main Algorithm (cont’d.)

- Output the `nthFibonacci` number, which is current
for Looping (Repetition) Structure

- **for loop**: called a counted or indexed **for loop**.
- **Syntax of the for statement:**

```
for (initial statement; loop condition; update statement)
statement
```

- **The initial statement, loop condition, and update statement are called** for loop control statements.

---

**EXAMPLE 5-9**

The following for loop prints the first 10 nonnegative integers:

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

The **initial statement**: `i = 0`, initialises the int variable `i` to 0. The loop condition, `i < 10`, is evaluated. Because `0 < 10` is true, the print statement executes and outputs 0. The update statement, `i++`, then executes, which sets the value of `i` to 1. Once again, the loop condition is evaluated, which is still true, and so on. When `i` becomes 10, the loop condition evaluates to **false**, the for loop terminates, and the statement following the for loop executes.

---

**EXAMPLE 5-10**

1. The following for loop output Hello! and a star (on separate lines) five times:

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

2. Consider the following for loop:

```
for (i = 1; i <= 5; i++)
cout << "Hello!" << endl;
cout << "*" << endl;
```

This loop outputs `Hello!` five times and the star only once.

---

**EXAMPLE 5-11**

The following for loop executes five empty statements:

```
for (i = 0; i < 5; i++)
   //Line 1
   //Line 2
```

The semicolon at the end of the for statement (before the output statement, Line 1) terminates the for loop. The action of this for loop is empty, that is, null.

- **The following is a semantic error** for loop:

```
for (;;)  
cout << "Hello" << endl;
```

- **The following is a legal (but infinite) for loop**:

```
for (; ;)  
cout << "Hello" << endl;
```

---

**EXAMPLE 5-12**

You can count backward using a for loop if the for loop control expressions are set correctly. For example, consider the following for loop:

```
for (i = 10; i >= 1; i--)
cout << i << endl;
```

The output is:

```
10 9 8 7 6 5 4 3 2 1
```

In this for loop, the variable `i` is initialised to 10. After each iteration of the loop, `i` is decremented by 1. The loop continues to execute as long as `i >= 1`.
for Looping (Repetition) Structure (cont’d.)

**EXAMPLE 5-13**

You can increment (or decrement) the loop control variable by any fixed number. In the following for loop, the variable is initialized to 1; at the end of the for loop, 1 is incremented by 2. This for loop outputs the first 10 positive odd integers.

```cpp
for (i = 1; i <= 20; i = i + 2)
    cout << " * " << i;
    cout << endl;
```

---

do...while Looping (Repetition) Structure (cont’d.)

**do...while Looping (Repetition) Structure (cont’d.)**

- The statement can be simple or compound.
- Loop always iterates at least once.

**FIGURE 5-3**

- Syntax of a do...while loop:
  ```cpp
do statement
while (expression);
```
- The statement executes first, and then the expression is evaluated.
  - As long as expression is true, loop continues.
- To avoid an infinite loop, body must contain a statement that makes the expression false.

---

do...while Looping (Repetition) Structure (cont’d.)

**EXAMPLE 5-19**

Consider the following two loops:

a. ```cpp
i = 1;
do {
    cout << " * " << i;
    i = i + 3;
} while (i <= 10);
```
   - The output of this code is:
   - 1 2 5 8 11
   - After 20 is output, the statement:
   - ```cpp
i = i + 3;
```
   - changes the value of i to 25 and so i <= 20 becomes false, which halts the loop.

b. ```cpp
i = 12;
do {
    cout << " * " << i;
    i = i + 3;
} while (i <= 10);
```
   - In (a), the while loop produces nothing. In (b), the do...while loop outputs the number 12, and also changes the value of i to 15.
Choosing the Right Looping Structure

- All three loops have their place in C++
  - If you know or can determine in advance the number of repetitions needed, the `for` loop is the correct choice
  - If you do not know and cannot determine in advance the number of repetitions needed, and it could be zero, use a `while` loop
  - If you do not know and cannot determine in advance the number of repetitions needed, and it is at least one, use a `do...while` loop

break and continue Statements

- `break` and `continue` alter the flow of control
- `break` statement is used for two purposes:
  - To exit early from a loop
  - Can eliminate the use of certain (flag) variables
  - To skip the remainder of a `switch` structure
- After `break` executes, the program continues with the first statement after the structure

break and continue Statements (cont’d.)

- `continue` is used in `while`, `for`, and `do...while` structures
- When executed in a loop
  - It skips remaining statements and proceeds with the next iteration of the loop

Nested Control Structures

- To create the following pattern:
  
  *  
  **  
  ***  
  ****  
  *****  

- We can use the following code:

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

Nested Control Structures (cont’d.)

- What is the result if we replace the first `for` statement with this?

```cpp
for (i = 5; i >= 1; i--)
```

- Answer:

  
  *****  
  ****  
  ***  
  **  
  *  

Avoiding Bugs by Avoiding Patches

- Software patch
  - Piece of code written on top of an existing piece of code
  - Intended to fix a bug in the original code
- Some programmers address the symptom of the problem by adding a software patch
- Should instead resolve underlying issue
Debugging Loops

- Loops are harder to debug than sequence and selection structures
- Use loop invariant
  - Set of statements that remains true each time the loop body is executed
- Most common error associated with loops is off-by-one

Summary

- C++ has three looping (repetition) structures:
  - while, for, and do...while
  - while, for, and do are reserved words
  - while and for loops are called pretest loops
  - do...while loop is called a posttest loop
  - while and for may not execute at all, but do...while always executes at least once

Summary (cont’d.)

- while: expression is the decision maker, and statement is the body of the loop
- A while loop can be:
  - Counter-controlled
  - Sentinel-controlled
  - EOF-controlled
- In the Windows console environment, the end-of-file marker is entered using Ctrl+z

Summary (cont’d.)

- for loop: simplifies the writing of a counter-controlled while loop
  - Putting a semicolon at the end of the for loop is a semantic error
- Executing a break statement in the body of a loop immediately terminates the loop
- Executing a continue statement in the body of a loop skips to the next iteration