Chapter 5: Control Structures II
(Repetition)
Objectives

In this chapter, you will:

• Learn about repetition (looping) control structures

• Explore how to construct and use count-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.)

• Learn how to avoid bugs by avoiding patches
• Learn how to debug loops
Why Is Repetition Needed?

• Repetition allows you to efficiently use 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

• The general form of the `while` statement is:

```
while (expression)
statement
```

• `while` is a reserved word
• 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.)

- **Infinite loop**: continues to execute endlessly
  - Avoided by including statements in loop body that assure exit condition is eventually **false**
while Looping (Repetition) Structure (cont'd.)

**EXAMPLE 5-1**

Consider the following C++ program segment:

```cpp
i = 0; //Line 1

while (i <= 20) //Line 2
{
    cout << i << " "; //Line 3
    i = i + 5; //Line 4
}

cout << endl;

Sample Run:
0 5 10 15 20
```
Designing **while** Loops

**EXAMPLE 5-2**

Consider the following C++ program segment:

```cpp
i = 20;               //Line 1
while (i < 20)        //Line 2
{
    cout << i << " ";  //Line 3
    i = i + 5;         //Line 4
}
cout << endl;         //Line 5
```

It is easy to overlook the difference between this example and Example 5-1. In this example, in Line 1, `i` is set to 20. Because `i` is 20, the expression `i < 20` in the **while** statement (Line 2) evaluates to **false**. Because initially the loop entry condition, `i < 20`, is **false**, the body of the **while** loop never executes. Hence, no values are output and the value of `i` remains 20.
Case 1: Counter-Controlled while Loops

- If you know exactly how many pieces of data need to be read,
  - while loop becomes a counter-controlled loop

```cpp
counter = 0;       //initialize the loop control variable

while (counter < N) //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

```cpp
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
• The program converts uppercase letters to their corresponding telephone digit
Case 3: Flag-Controlled while Loops

- A flag-controlled `while` loop uses a `bool` variable to control the loop.
- The flag-controlled `while` loop takes the form:

```cpp
found = false; //initialize 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

• The program uses the function `rand` of the header file `cstdlib` to generate a random number
  – `rand()` returns an `int` value between 0 and 32767
  – To convert it to an integer greater than or equal to 0 and less than 100:
    • `rand() % 100`
Case 4: EOF-Controlled while Loops

• Use an EOF (End Of File)-controlled while loop

• The logical value returned by `cin` can determine if the program has ended input

```cpp
  cin >> variable;  //initialize the loop control variable

  while (cin)        //test the loop control variable
  {
    .
    .
    .
    cin >> variable; //update the loop control variable
    .
    .
  }
```
**eof Function**

- The function `eof` can determine the end of file status
- `eof` is a member of data type `istream`
  - Like other I/O functions
- The syntax for the function `eof` is:
  
  ```cpp
  istreamVar.eof()
  ```

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

• The expression in a `while` statement can be complex
  – For example:

```c++
while ((noOfGuesses < 5) && (!isGuessed))
{
    ...
}
```
Programming Example: Fibonacci Number

• Consider the following sequence of numbers:
  – 1, 1, 2, 3, 5, 8, 13, 21, 34, ....

• Given the first two numbers of the sequence (say, $a_1$ and $a_2$)
  – $n$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**
  - \( a_2 = 1 \)
  - \( a_1 = 1 \)
  - Determine the \( n \)th number, \( a_n \), \( n \geq 3 \)
Programming Example: Fibonacci Number (cont'd.)

• Suppose $a_2 = 6$ and $a_1 = 3$
  - $a_3 = a_2 + a_1 = 6 + 3 = 9$;
  - $a_4 = a_3 + a_2 = 9 + 6 = 15$

• 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 Fibonacci number in the sequence
  – Calculate the next Fibonacci number
    • By adding the previous two elements of the Fibonacci sequence
Programming Example: Problem Analysis and Algorithm Design (cont'd.)

– 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

1. Prompt the user for the first two numbers—that is, \texttt{previous1} and \texttt{previous2}
2. Read (input) the first two numbers into \texttt{previous1} and \texttt{previous2}
3. Output the first two Fibonacci numbers
4. Prompt the user for the position of the desired Fibonacci number
5. Read the position of the desired Fibonacci number into \texttt{nthFibonacci}
Programming Example: Main Algorithm (cont'd.)

6.

a. if (nthFibonacci == 1)
   The desired Fibonacci number is the first Fibonacci number. Copy the value of previous1 into current

b. else if (nthFibonacci == 2)
   The desired Fibonacci number is the second Fibonacci number. Copy the value of previous2 into current.
6. (cont’d.)

c. 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:
  current = previous2 + previous1;
Programming Example: Main Algorithm (cont'd.)

6. c. (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.)

7. Output the \text{n}th Fibonacci number, which is current
for Looping (Repetition) Structure

• The general form of the for statement is:

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

• The initial statement, loop condition, and update statement are called for loop control statements
  – initial statement usually initializes a variable (called the for loop control, or for indexed, variable)

• In C++, for is a reserved word
for Looping (Repetition) Structure (cont'd.)

**FIGURE 5-2** for loop
for Loopping (Repetition) Structure (cont'd.)

**EXAMPLE 5-9**

The following for loop prints the first 10 nonnegative integers:

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

The initial statement, `i = 0;`, initializes the `int` variable `i` to 0. Next, 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.
for Looping (Repetition) Structure (cont'd.)

• C++ allows you to use fractional values for loop control variables of the double type – Results may differ

• The following is a semantic error:

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

• The following is a legal for loop:

```cpp
for (;;)
  cout << "Hello" << endl;
```
**for Looping (Repetition) Structure (cont'd.)**

---

**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:

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

---

**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, *i* 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**

- **General form of a do...while:**
  ```cpp
do statement
while (expression);
```

- **The statement executes first, and then the expression is evaluated**
- **To avoid an infinite loop, body must contain a statement that makes the expression false**
- **The statement can be simple or compound**
- **Loop always iterates at least once**
**do...while Loopping (Repetition) Structure (cont'd.)**

![Diagram of do-while loop]

**FIGURE 5-3 do...while loop**
do...while Looping (Repetition) Structure (cont'd.)

```cpp
EXAMPLE 5-18

i = 0;
do
{   cout << i << " ";
    i = i + 5;
} while (i <= 20);

The output of this code is:
0 5 10 15 20

After 20 is output, the statement:
i = i + 5;
changes the value of i to 25 and so i <= 20 becomes false, which halts the loop.
```
do...while Looping (Repetition) Structure (cont'd.)

**EXAMPLE 5-19**

Consider the following two loops:

a.  
   ```
   i = 11;
   while (i <= 10)
   {
     cout << i << " ";
     i = i + 5;
   }
   cout << endl;
   ```

b.  
   ```
   i = 11;
   do
   {
     cout << i << " ";
     i = i + 5;
   }
   while (i <= 10);
   ```
   
   cout << endl;

In (a), the **while** loop produces nothing. In (b), the **do...while** loop outputs the number 11 and also changes the value of i to 16.
Example 5-20: Divisibility Test by 3 and 9

```cpp
sum = 0;

do
{
    sum = sum + num % 10; //extract the last digit
    num = num / 10;      //and add it to sum
}
while (num > 0);

cout << "The sum of the digits = " << sum << endl;

if (sum % 3 == 0)
    cout << temp << " is divisible by 3" << endl;
else
    cout << temp << " is not divisible by 3" << endl;

if (sum % 9 == 0)
    cout << temp << " is divisible by 9" << endl;
else
    cout << temp << " is not divisible by 9" << endl;
```
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 the **switch** structure

• After the **break** statement 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 the following?

```c++
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 the 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