Chapter 5: Loops

Sections 5.1–5.6, 5.9

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)
Updated by Dr. Ashraf Suyyagh (Spring 2021)

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Introduction

Suppose that you need to print a string (e.g., "Welcome to C++!") a hundred times. It would be tedious to have to write the following statement a hundred times:

```
cout  << "Welcome to C++!" << endl;
```

Introduction

```
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;


…


cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
```

`100 times`

So, how do you solve this problem?

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

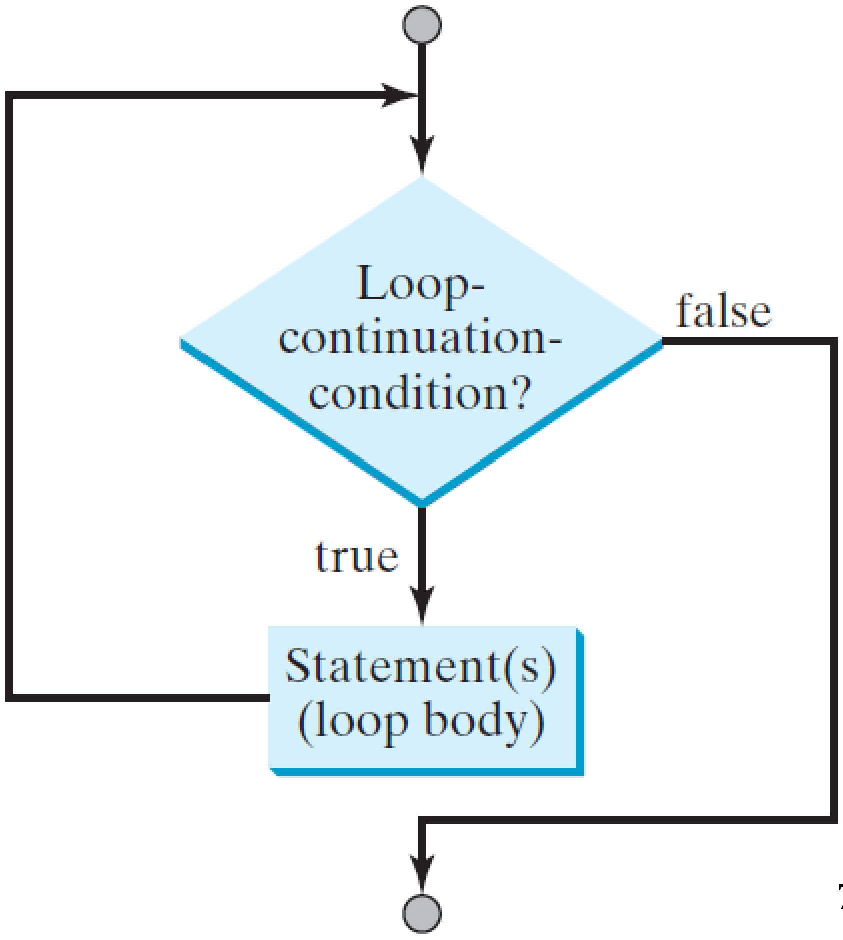
Introducing `while` Loops

*A **`while`** loop executes statements repeatedly while the condition is true.*

```
int count = 0;
while (count < 100)
{
   cout << "Welcome to C++!\n";
   count++;
}
```

`while` Loop Flow Chart

```
while (loop-continuation-condition)
{
  // Loop body
  Statement(s);
}
```



animation

Trace while Loop

Initialize count

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

animation

Trace while Loop, cont.

(count < 2) is true

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

animation

Trace `while` Loop, cont.

```
int count = 0;
while (count < 2)
{
  cout << "Welcome to C++!";
  count++;
}
```

Print Welcome to C++

animation

Trace `while` Loop, cont.

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

Increase count by 1
count is 1 now

Trace `while` Loop, cont.

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

(count < 2) is still true since count is 1

animation

Trace while Loop, cont.

```
int count = 0;
while (count < 2)
{
  cout << "Welcome to C++!";
  count++;
}
```

Print Welcome to C++

Trace while Loop, cont.

Increase count by 1
count is 2 now

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

animation

Trace while Loop, cont.

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

(count < 2) is false since count is 2 now

animation

Trace while Loop

The loop exits. Execute the next statement after the loop.

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

Case Study: Guessing Numbers

Write a program that randomly generates an integer between **0** and **100**, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can choose the next input intelligently. Here is a sample run:

GuessNumberOneTime Run

GuessNumber Run

GuessNumber.cpp 1/2

```
#include <iostream>
#include <cstdlib>
#include <ctime> // Needed for the time function
using namespace std;

int main()
{
    // Generate a random number to be guessed
    srand(time(0));
    int number = rand() % 101;

    cout << "Guess a magic number between 0 and 100";
```

GuessNumber.cpp 1/2

```
    int guess = -1;
    while (guess != number)
    {
        // Prompt the user to guess the number
        cout << "\nEnter your guess: ";
        cin >> guess;

        if (guess == number)
            cout << "Yes, the number is " << number << endl;
        else if (guess > number)
            cout << "Your guess is too high" << endl;
        else
            cout << "Your guess is too low" << endl;
    } // End of loop

    return 0;
}
```

Loop Design Strategy

Step 1: Identify the statements that need to be repeated.

Step 2: Wrap these statements in a loop as follows:

```
while (true)
{
  Statements;
}
```

Step 3: Code the loop-continuation-condition and add appropriate statements for controlling the loop.

```
while (loop-continuation-condition)
{
  Statements;
  Additional statements for controlling the loop;
}
```

Case Study: Multiple Subtraction Quiz

Take the subtraction quiz 5 times.

Report number of correct answers and the quiz time.

SubtractionQuiz

Run

SubtractionQuizLoop.cpp 1/3

```
#include <iostream>
#include <ctime> // Needed for time function
#include <cstdlib> // Needed for the srand and rand functions
using namespace std;
int main()
{
    int correctCount = 0; // Count the number of correct answers
    int count = 0; // Count the number of questions
    long startTime = time(0);
    const int NUMBER_OF_QUESTIONS = 5;
    srand(time(0)); // Set a random seed
    while (count < NUMBER_OF_QUESTIONS)
    {      See next slides      }
    long endTime = time(0);
    long testTime = endTime - startTime;
    cout << "Correct count is " << correctCount << "\nTest time is "
        << testTime << " seconds\n";
    return 0;
}
```

SubtractionQuizLoop.cpp 2/3

```
while (count < NUMBER_OF_QUESTIONS)
    {
        // 1. Generate two random single-digit integers
        int number1 = rand() % 10;
        int number2 = rand() % 10;

        // 2. If number1 < number2, swap number1 with number2
        if (number1 < number2)
        {
            int temp = number1;
            number1 = number2;
            number2 = temp;
        }
```

SubtractionQuizLoop.cpp 3/3

```
    // 3. Prompt the student to answer “what is num1 – num2?”
    cout << "What is " << number1 << " - " << number2 << "? ";
    int answer;
    cin >> answer;
    // 4. Grade the answer and display the result
    if (number1 - number2 == answer)
    {
        cout << "You are correct!\n";
        correctCount++;
    }
    else
        cout << "Your answer is wrong.\n" << number1 << " - " <<
        number2 << " should be " << (number1 - number2) << endl;
    // Increase the count
    count++;
}
```

Controlling a Loop with User Confirmation

```
char continueLoop = 'Y';
while (continueLoop == 'Y')
{
  // Execute the loop body once
  ...

  // Prompt the user for confirmation
  cout << "Enter Y to continue and N to quit: ";
  cin >> continueLoop;
}
```

Controlling a Loop with a Sentinel Value

You may use an input value to signify the end of the loop. Such a value is known as a *sentinel value*.

A program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

SentinelValue Run

SentinelValue.cpp

```cpp
int data;
cin >> data;

// Keep reading data until the input is 0
int sum = 0;
while (data != 0)
{
    sum += data;

    // Read the next data
    cout << "Enter an integer (the input ends " <<
        "if it is 0): ";
    cin >> data;
}

cout << "The sum is " << sum << endl;
```

Input and Output Redirections

- If you have a large number of data to enter, it would be cumbersome to type from the keyboard.
- You may store the data separated by whitespaces in a text file, say `input.txt`, and run the program and redirecting input to the file.
- You can also redirect program output to a text file, say `output.txt`.

```
SentinelValue.exe < input.txt > output.txt
```

Reading Data from a File

- If you have many numbers to read from a file, you need to write a loop to read all these numbers.
- You can invoke the **`eof()`** function on the input object to detect the end of file.
- A program that reads all numbers from the file **`numbers.txt`**.

ReadAllData Run

ReadAllData.cpp

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    // Open a file
    ifstream input("numbers.txt");

    double sum = 0;
    double number;
    while (!input.eof()) // Read data to the end of file
    {
        input >> number; // Read data
        cout << number << " "; // Display data
        sum += number;
    }
    input.close();
    cout << "\nTotal is " << sum << endl;
    return 0;
}
```

Caution

- Don’t use floating-point values for equality checking in a loop control expression; they are approximations, using them can result in inaccurate results.
- The following loop does not stop.

```
double item = 1;
double sum = 0;
while (item != 0) // No guarantee it will be 0
{
    sum += item;
    item -= 0.1;
}
```

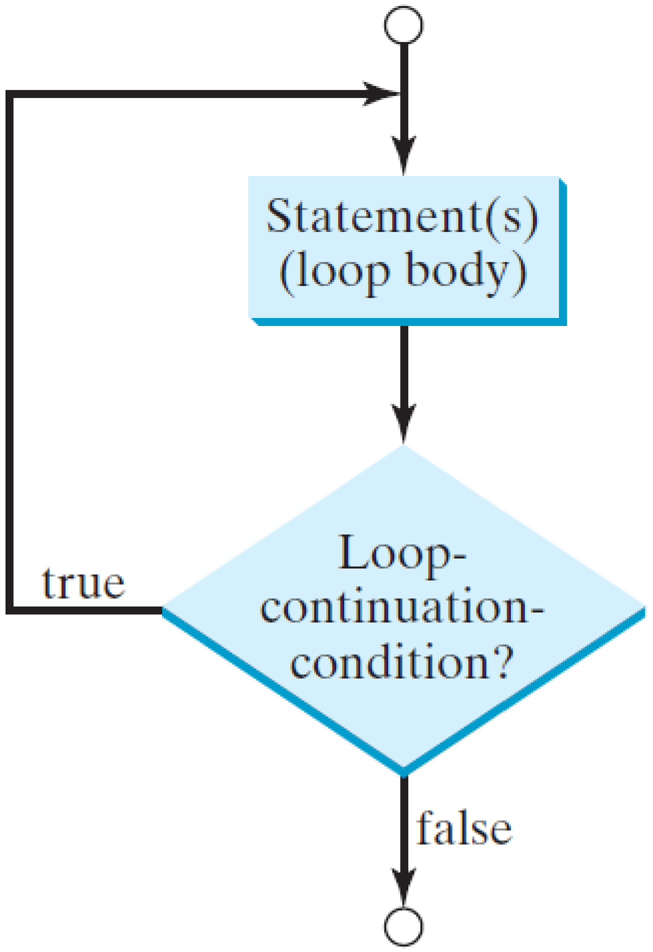
Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

`do-while` Loop

A **`do-while`** *loop is the same as a while loop except that it executes the loop body first and then checks the loop continuation condition.*

```
do
{
   // Loop body;
   Statement(s);
} while (loop-continuation-cond
```



TestDoWhile

Run

TestDoWhile.cpp

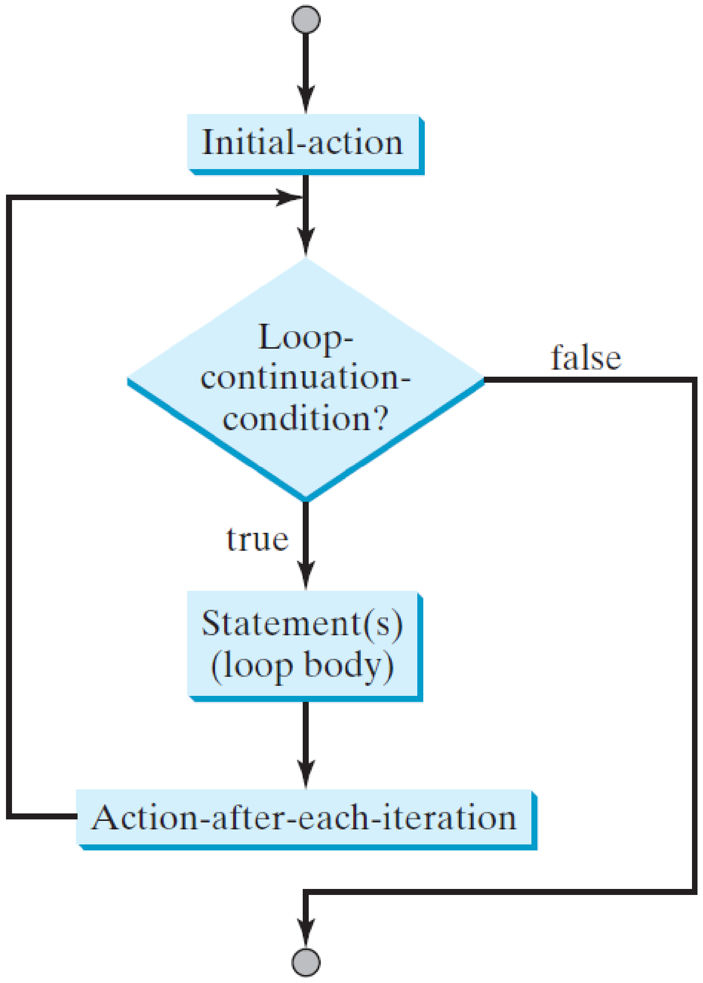
```
// Initialize data and sum
int data = 0;
int sum = 0;

do
{
    sum += data;

    // Read the next data
    cout << "Enter an integer (the input ends " <<
        "if it is 0): ";
    cin >> data; // Keep reading until the input is 0
} while (data != 0);

cout << "The sum is " << sum << endl;
```

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

for Loops

```
for (initial-action; loop-continuation-condition;
     action-after-each-iteration)
{
   // Loop body;
   Statement(s);
}
```



*A **for** loop has a concise syntax for writing loops.*

animation

Trace for Loop

Declare i

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace for Loop, cont.

Execute initializer
i is now 0

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

Trace `for` Loop, cont.

(i < 2) is true
since i is 0

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace `for` Loop, cont.

Print Welcome to C++!

```
int i;
for (i = 0; i < 2; i++)
{
  cout <<  "Welcome to C++!";
}
```

Trace `for` Loop, cont.

Execute adjustment statement
i now is 1

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace for Loop, cont.

(i < 2) is still true
since i is 1

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace for Loop, cont.

Print Welcome to C++

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace `for` Loop, cont.

Execute adjustment statement
i now is 2

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace for Loop, cont.

(i < 2) is false
since i is 2

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace `for` Loop, cont.

Exit the loop. Execute the next statement after the loop

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

Note

- The **initial-action** in a **for** loop can be a list of zero or more comma-separated expressions.

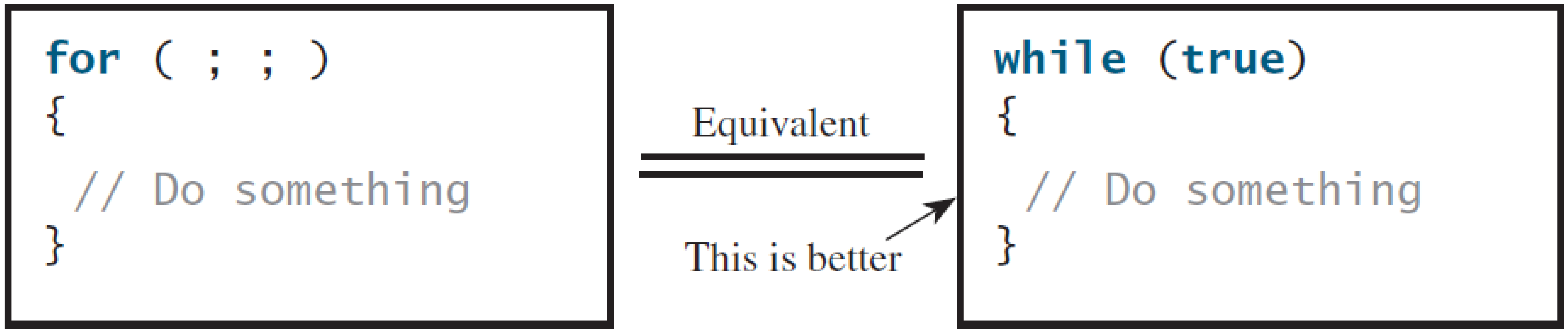
```
for (int i = 0, j = 0; i + j < 10; i++, j++)
{
  // Do something
}
```

- The **action-after-each-iteration** in a **for** loop can be a list of zero or more comma-separated statements.

```
for (int i = 1; i < 100; cout << i << endl, i++);
```

Note

- If the `loop-continuation-condition` in a `for` loop is omitted, it is implicitly true. Thus the `for` statement given below, which is an infinite loop, is correct.
- It is better to use the equivalent `while` loop to avoid confusion:



```
for ( ; ; )
{
  // Do something
}
```

```
while (true)
{
  // Do something
}
```

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Which Loop to Use?

- The loop statements, **`while`**, **`do-while`**, and **`for`**, are expressively equivalent; that is, you can write a loop in any of these three forms.

- The **`while`** loop can always be converted into the **`for`** loop.

```
while (loop-continuation-condition)
{
  // Loop body
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; )
{
  // Loop body
}
```

(b)

- The **`for`** loop can generally be converted into the **`while`** loop.

```
for (initial-action;
     loop-continuation-condition;
     action-after-each-iteration)
{
  // Loop body;
}
```

(a)

Equivalent

```
initial-action;
while (loop-continuation-condition)
{
  // Loop body;
  action-after-each-iteration;
}
```

(b)

Which Loop to Use?

- Use the one that is most intuitive and comfortable for you.
- In general, a **`for`** loop may be used if the number of repetitions is counter-controlled, as, for example, when you need to print a message 100 times.
- A **`while`** loop may be used if the number of repetitions is sentinel-controlled, as in the case of reading the numbers until the input is 0.
- A **`do-while`** loop can be used to replace a while loop if the loop body has to be executed before testing the continuation condition.

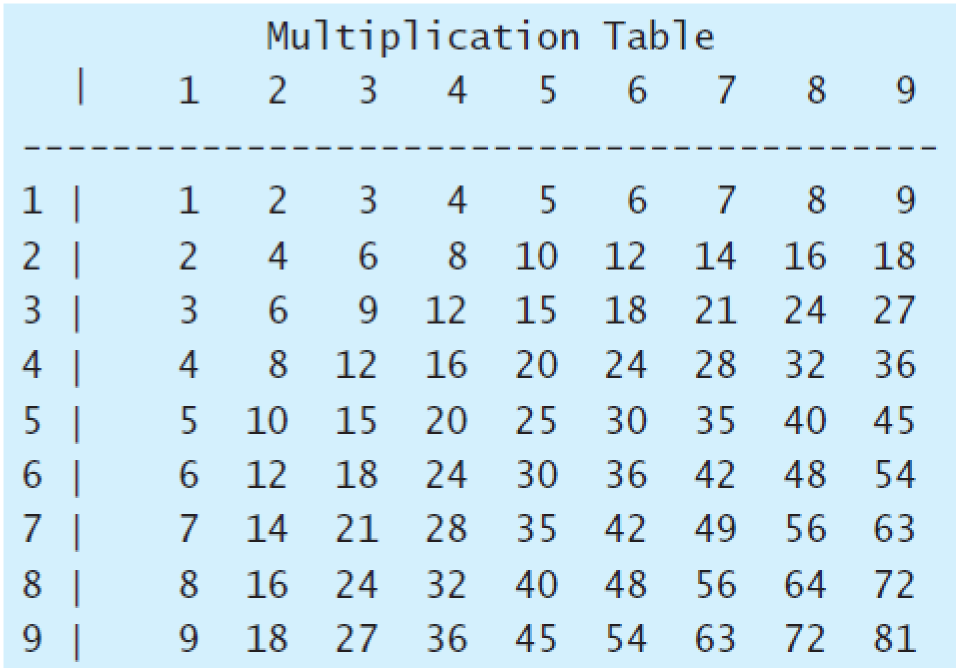
Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Nested Loops

A loop can be nested inside another loop.

Example: A program that uses nested for loops to print a multiplication table.



```
           Multiplication Table
   |    1   2   3   4   5   6   7   8   9
-----------------------------------------
1  |    1   2   3   4   5   6   7   8   9
2  |    2   4   6   8  10  12  14  16  18
3  |    3   6   9  12  15  18  21  24  27
4  |    4   8  12  16  20  24  28  32  36
5  |    5  10  15  20  25  30  35  40  45
6  |    6  12  18  24  30  36  42  48  54
7  |    7  14  21  28  35  42  49  56  63
8  |    8  16  24  32  40  48  56  64  72
9  |    9  18  27  36  45  54  63  72  81
```

MultiplicationTable

Run

MultiplicationTable.cpp 1/2

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << "        Multiplication Table\n";

    // Display the number title
    cout << "  | ";
    for (int j = 1; j <= 9; j++)
        cout << setw(3) << j;
    cout << "\n";

    cout << "-------------------------------\n";
```

MultiplicationTable.cpp 2/2

```
// Display table body
    for (int i = 1; i <= 9; i++)
    {
        cout << i << " | ";
        for (int j = 1; j <= 9; j++)
        {
            // Display the product and align properly
            cout << setw(3) << i * j;
        }
        cout << "\n";
    }

    return 0;
}
```

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Using `break` and `continue`

Use `break` in a loop to immediately terminate the loop.

Example: adding integers from 1 to 20 until sum is greater than or equal to 100.

```
while (number < 20)
{
    number++;
    sum += number;
    if (sum >= 100)
        break;
}
```

TestBreak

Run

Using `break` and `continue`

Use **`continue`** in a loop to proceed to the next iteration.

Example: adding integers from 1 to 20 except 10 and 11.

```
while (number < 20)
   {
         number++;
         if (number == 10 || number == 11)
               continue;
         sum += number;
   }
```

TestContinue Run

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**