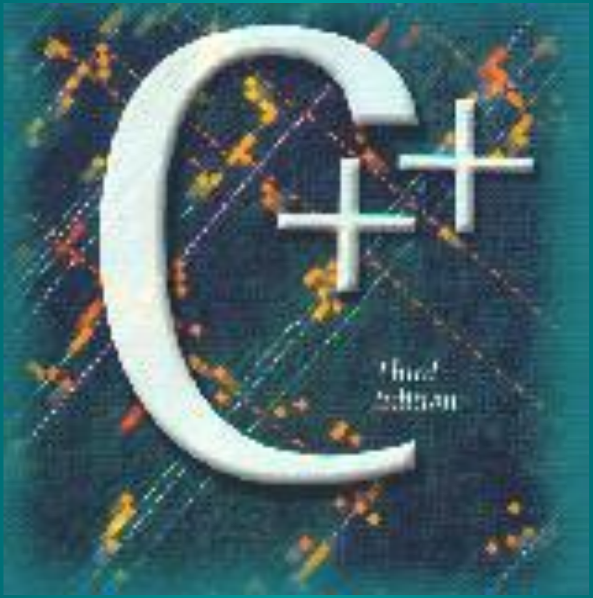


COS120 Software Development Using C++
AUBG, COS dept

Lecture 12: Control Flow. Repetition and Loop structures

Lecture Contents:

- General concept of loop statements
- The `for` loop statement
- The `while` loop statement
- The `do` ... `while` loop statement
- Demo programs
- Exercises

Control Structures

- Three methods of processing a program
  - In sequence
  - Branching
  - **Looping**
- Branch: altering the flow of program execution by making a selection or choice
- **Loop: altering the flow of program execution by repetition of statement(s)**

Flow of Execution

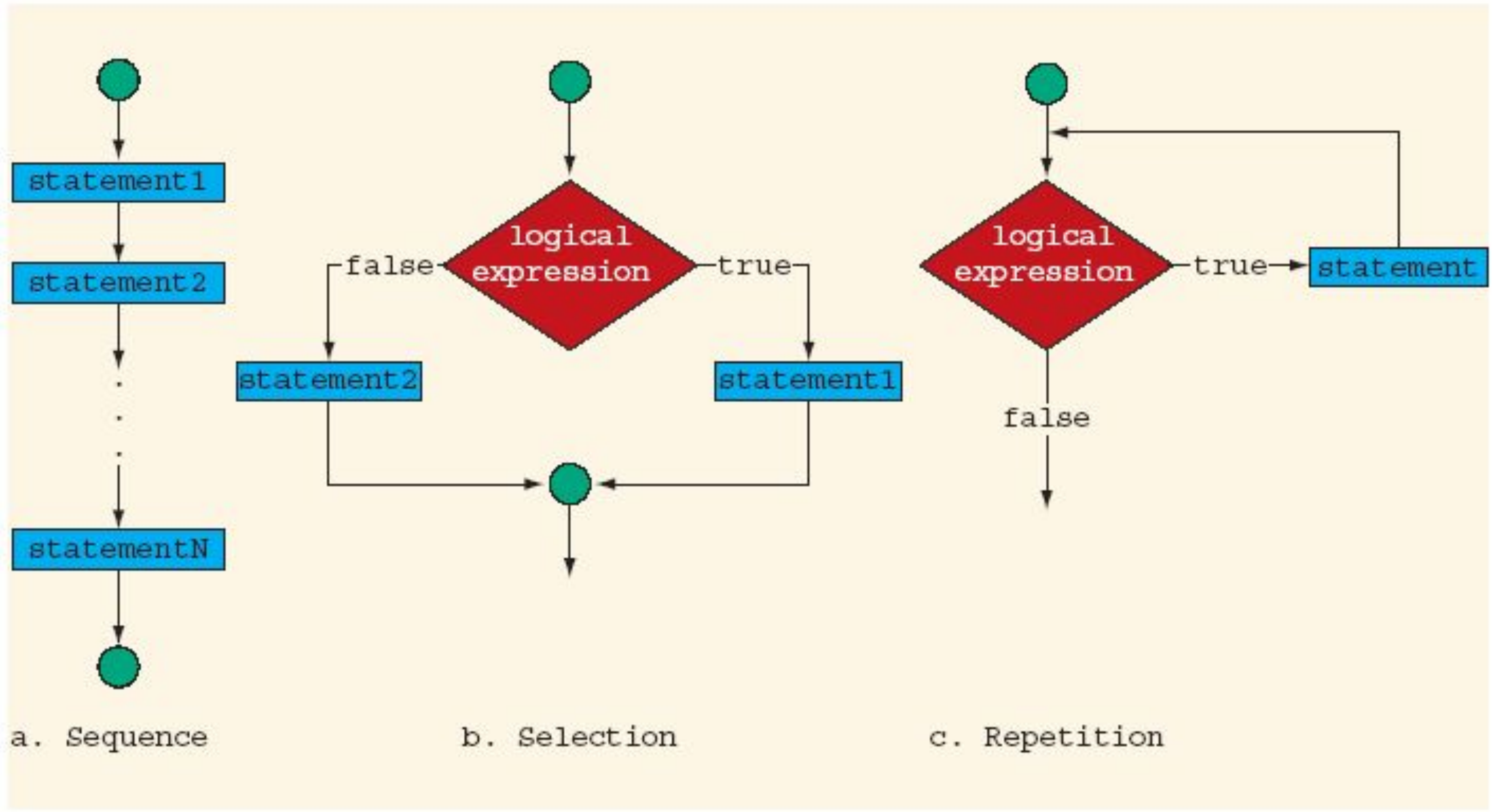


FIGURE 4-1 Flow of execution

.

General concept of loop statements

- Statement or a group of statements to be executed many times;
- Fixed number of iterations (counter controlled loop);
- Indefinite number of iterations (logically controlled loop);
- Pre test loop control structures (0, 1 or more iterations);
- Post test loop control structures (1 or more iterations).

Digression on increment/decrement operators

- Problem: to increment (add 1 to) a variable
- C/C++ offers 4 ways to solve this task

```
              |
var = var+1;| var = var+value;
var += 1;   |   var += value;
var++;      |
++var;      |
              |
```

Digression on increment/decrement operators

- Problem: to decrement (subtract 1) a variable
- C/C++ offers 4 ways to solve this task:

```
            |
var = var-1;| var = var-value;
var -= 1;   |   var -= value;
var--;      |
--var;      |
            |
```

The for loop statement

The **for** loop statement

The for loop statement

Syntax and flowchart fragment:

for (initialization expression ; loop repetition condition ; update expression) statement;

for (<express1>;<express2>;<express3>) <stmt>;

```
int I;
for (I=0; I<=9; I=I+1)  cout << "\nAUBG";
```

The for loop statement

Syntax and flowchart fragment:

for (initialization expression ; loop repetition condition ; update expression) statement;

for (<express1>;<express2>;<express3>) <stmt>;

```
for (int I=0; I<=9; I+=1)  cout << “\nAUBG”;
```

The for loop statement

Syntax and flowchart fragment:

for (initialization expression ; loop repetition condition ; update expression) statement;

for (<express1>;<express2>;<express3>) <stmt>;

```
for (int I=0; I<=9; I++)  cout << "\nAUBG";
```

The for loop statement

Syntax and flowchart fragment:

for (initialization expression ; loop repetition condition ; update expression) statement;

for (<express1>;<express2>;<express3>) <stmt>;

```
for (int I=0; I<=9; ++I)  cout << “\nAUBG”;
```

The for loop statement

Write a C++ program to run your first loop

What is the output expected to be displayed?

```
int main()
{
 for (int I=0; I<=9; I=I+1)
  cout << “\nAUBG  ”;
  cout << “Blagoevgrad”;
 return 0;
}
```

The for loop statement

Write a C++ program to run your first loop

Reminder on compound statement

```
int main()
{
 for (int I=0; I<=9; ++I)
  {
     cout << "\nAUBG  ";
     cout << "Blagoevgrad";
  }
 return 0;
}
```

The while loop statement

The **while** loop statement

The while loop statement

Syntax and flowchart fragment:

while (loop repetition condition) statement;

while (<expression>) <statement>;

```
int I=0;
while (I<=9)
   {
      cout<<"\nAUBG";
      I = I + 1;
   }
```

The while loop statement

Syntax and flowchart fragment:

while (loop repetition condition) statement;

while (<expression>) <statement>;

```
int I=0;
while (I<=9) {cout<<”\nAUBG”; I++;}
```

The do … while loop statement

The

do … while

loop statement

The do … while loop statement

Syntax and flowchart fragment:

do statement *while* (loop repetition condition);

do <statement> *while* (<expression>);

```
int I=0;
do {
   cout<<"\nAUBG";
   I++;
     }
while (I<=9);
```

The do … while loop statement

Syntax and flowchart fragment:

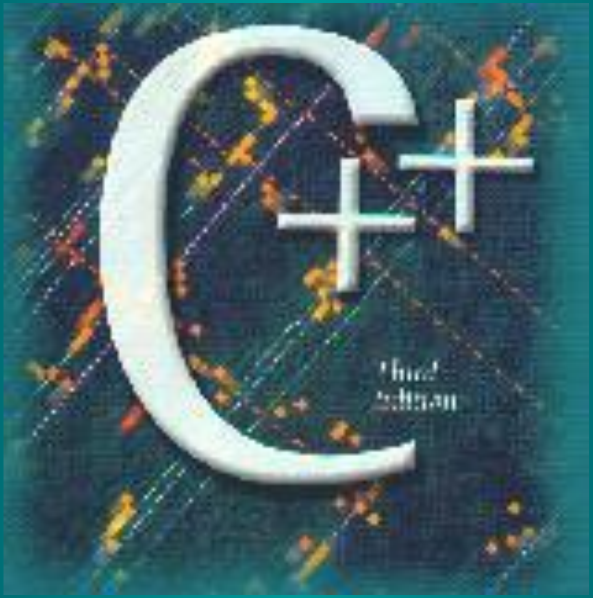
do statement *while* (loop repetition condition);

do <statement> *while* (<expression>);

```
int I=0;
do { cout<<”\nAUBG”; I++; } while (I<=9);
```

More on loop statement(s)

Extract from Friedman/Koffman, chapter 5



Repetition and Loop Statements

Chapter 5

Why iterate?

- Use the computer's speed to do the same task faster than if done by hand.
- Avoid writing the same statements over and over again.

Repetitive control structures

- Because many algorithms require many iterations over the same statements.
  - To average 100 numbers, we would need 300 plus statements.
  - Or we could use a statement that has the ability to repeat a collection of statements:
  - Pre test loops
  - Post test loops.

5.1 Counting Loops and the **while** Statement

– General form of the while statement:

```
while (  loop-test  )
  {
       iterative-part
  }
```

– When a while loop executes, the loop-test is evaluated. If true (non-zero), the iterative part is executed and the loop-test is reevaluated. This process continues until the loop test is false.

– Pre test loop

Collections of statements are delimited with { and }

// while there is another number, do the following

```
{
      cout << "Enter number: ";
      cin >> number;
      sum = sum + number;
}
average = sum / 100;
```

Sum 100 values the hard way

```
int sum = 0;
cout << "\n Enter number: "; // <-Repeat these three
cin >> number;                   // <- statements for each
sum = sum + number;                // <- number in the set
cout << "\n Enter number: ";
cin >> number;
sum = sum + number;
/*
  . . .  97*3 = 291 statements deleted ...
*/
cout << "\n Enter number: ";
cin >> number;
sum = sum + number;
average = sum / 100;
```

Sum 100 values the soft way

```
int sum = 0;
 int I=1;
 while (I<= 100)
   {
    cout << "\n Enter number: ";
    cin >> number;
    sum = sum + number;

    I = I + 1;
   }
average = sum / 100;
```

Sum 100 values the soft way

```
int sum = 0;
 int I;
 for( I=1; I<= 100; I=I+1)
 {
  cout << “\n Enter number: ";
  cin >> number;
  sum = sum + number;
 }
average = sum / 100;
```

Compound Assignment Operators

- Lets look at the idea of adding together a group of numbers
- Short hand notation

```
totalPay += pay;
```

  – same as

```
totalPay = totalPay + pay;
```

5.3 The **for** Statement

- The for loop is similar to the other C++ looping construct the while loop.
- The for loop forces us to write, as part of the for loop, an initializing statement, the loop-test, and a statement that is automatically repeated for each iteration.
- Pre test loop.

Example **for** loop

- This is a for-loop version of a counter-controlled loop :
- Scope of the loop control variable:

```
for(  int counter = 1;  counter<=5;  counter = counter+1)
{
    cout << counter << "  ";
}
```

- Output: _____?

General form of a **for** loop

for(*initial statement* ; *loop-test* ; *repeated statement***)**
 {
   iterative-part
 }

– When a for loop is encountered, the initial-statement is executed. The loop-test is executed. If the loop-test is false, the for loop is terminated. If loop-test is true, the iterative-part is executed and the repeated-statement is executed.

Other Incrementing Operators

- The unary ++ and -- operators add 1 and subtract 1 from the operand, respectively.
  - **int n = 0;**
  - **n++; // n is now 1 Equivalent to n=n+1;**
  - **n++; // n is now 2**
  - **n--;// n is now 1 again**
- The expression n++; is equivalent to the longer
  n = n + 1;
- It is common to see counter-controlled loops of this form where n is the number of reps

5.4 Conditional Loops

- In many programming situations, you will not be able to determine the exact number of loop repetitions
- Conditional Loop
  - Initialize the loop control variable
  - While a condition involving the loop control variable is true
  - Continue processing
  - Update the loop control variable

5.6 The **do-while** Statement

- The do while statement is similar to the while loop, but the do while loop has the test at the end. General form:

```
do {
        iterative-part
    } while ( loop-test ) ;
```

- Notice the iterative part executes BEFORE the loop-test)

When to use the **do-while** loop

- The do while loop is a good choice for obtaining interactive input from menu selections.
- Consider a function that won't stop executing until the user enters an N, O, or S:
- Post test loop

Example **do-while** loop

```
char menuOption()
{
    // POST: Return an upper case 'N', 'O' or 'S'
  char option;
  do {
      cout << "Enter N)ew, O)pen, S)ave: ";
      cin >> option;
      option = toupper(option);   // from <cctype> or <ctype.h>
       } while (option != 'N' || option != 'O' || option != 'S');
  return option;
}
```

5.7 Review of **while, for,** and **do-while** Loops

- ***while***
  - Most commonly used when repetition is not counter controlled;
  - condition test precedes each loop repetition;
  - loop body may not be executed at all

5.7 Review of **while, for,** and **do-while** Loops

- ***for***
  - Counting loop
  - When number of repetitions is known ahead of time and can be controlled by a counter;
  - also convenient for loops involving non counting loop control with simple initialization and updates;
  - condition test precedes the execution.

Review of **while, for,** and **do-while** Loops

- ***do-while***
  - Convenient when at least one repetition of loop body must be ensured.
  - Post test condition after execution of body.

5.10 Common Programming Errors

- – Coding style and use of braces.
- – Infinite loops will “hang you up !!”
- – Use lots of comments before and after a loop.
- – Test various conditions of loops.
- – Add white space between code segments using loops.
- – Initialize looping variables or use internal loop control variables (lcv) in the for loop.

Exercise 12.1

Build programs based on loop algorithms using the repetition statements:

- To display the even numbers in the range 2 … 36;

Exercise 12.2

Build programs based on loop algorithms using the repetition statements:

- To compute the sum of consecutive numbers 1, 2, 3… n (n is an input value);

Exercise 12.3

Build programs based on loop algorithms using the repetition statements:

- To compute the product of series of odd numbers 1, 3, 5 … n (n is an input value);

Exercise 12.4

Build programs based on loop algorithms using the repetition statements:

- To display a table of Fahrenheit Celsius temperature degrees in range 0 … 100 (+20)

  F = 9/5 * C + 32 or

  C = 5/9 * (F – 32);

Exercise 12.5

Build programs based on loop algorithms using the repetition statements:

- To display the distance driven by an automobile traveled at an average speed of 55 miles/hour after .5, 1.0, 1.5, … 4.0 hours;

Before lecture end

Lecture:

Control Flow. Repetition and loop structures

More to read:

Friedman/Koffman, Chapter 05

Chapter 5: Repetition and Loop Statements

Problem Solving,
Abstraction, and Design using C++ 5e

by Frank L. Friedman and Elliot B. Koffman

Control Structures

- Sequence
- Selection
- **Repetition**

5.1 Counting Loops and **while**

- **Loop** – a control structure that repeats a group of statements in a program
- **Loop body** – the statements that are repeated in a loop

Counter-Controlled Loop

- Repetition managed by a loop control variable whose value represents a count
- Counting Loop
  - Set **loop control variable** to an initial value of 0
  - While loop control variable < final value
    - …
    - Increase loop control variable by 1

Counter-Controlled Loop

- Used when we can determine prior to loop execution how many loop repetitions will be needed to solve problem
- Number of repetitions should appear as the final count in the **while** condition

Listing 5.1 Program fragment with a loop

```
countEmp = 0;                 // no employees processed yet
while (countEmp < 7)          // test the count of employees
{
   cout << "Hours: ";
   cin >> hours;
   cout << "Rate : $";
   cin >> rate;
   pay = hours * rate;
   cout << "Weekly pay is " << pay << endl;
   countEmp = countEmp + 1;   // increment count of
                              //    employees
}
cout << "All employees processed" << endl;
```

The **while** Statement - Example

- Loop Body
  - Compound statement
  - Gets an employee’s payroll data
  - Computes and displays employee’s pay
- After 7 weekly pay amounts are displayed, the statement following loop body executes
  - Displays message “All employees processed.”

The **while** Statement - Example

- countEmp = 0;
  - Sets initial value of 0, representing the count of employees processed so far
- Condition evaluated (countEmp < 7)
  - If true, loop body statements are executed
  - If false, loop body is skipped and control passes to the display statement (cout) that follows the loop body

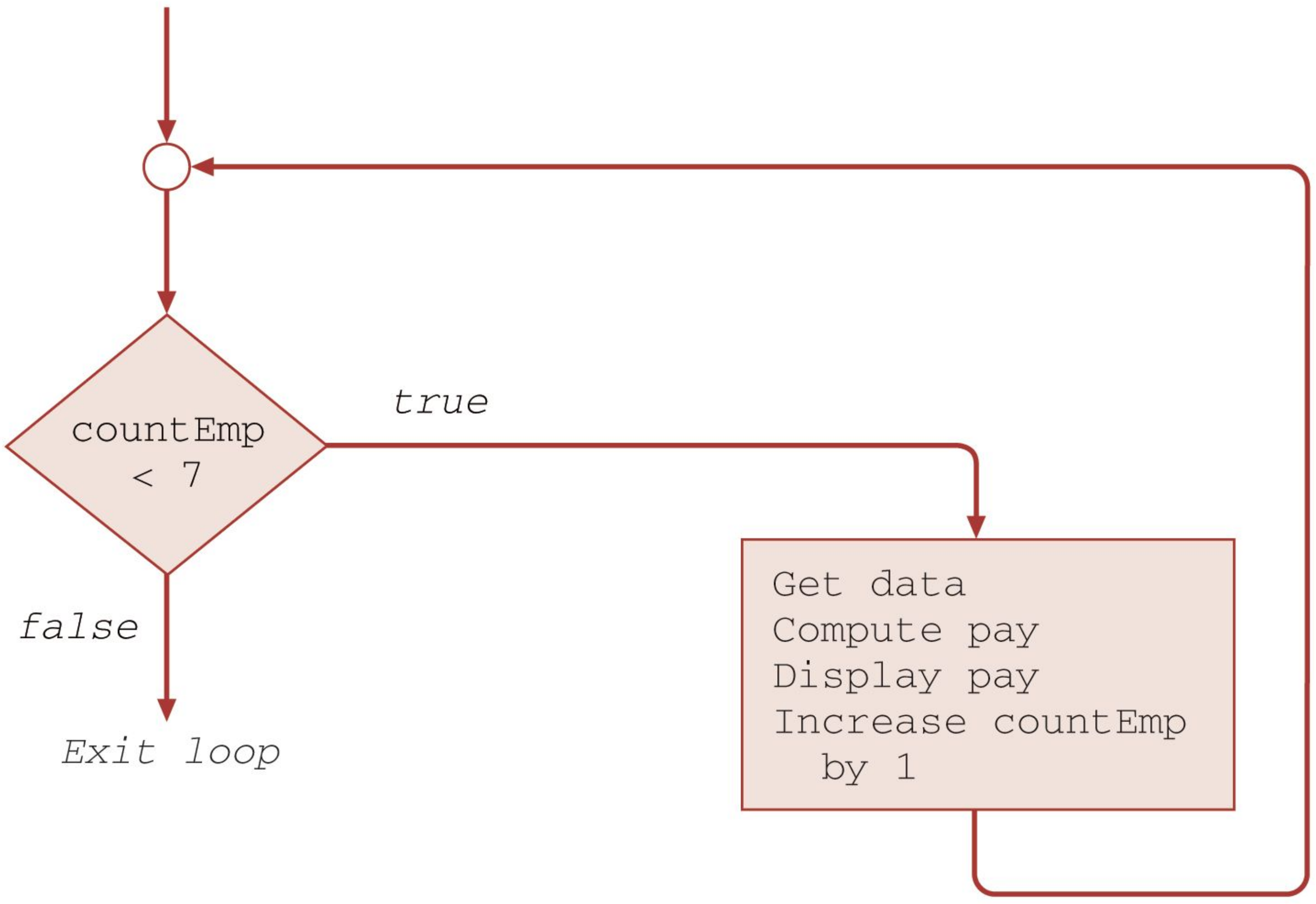
The **while** Statement - Example

- countEmp = countEmp + 1;
  - Increments the current value of the counter by 1
- After executing the last statement of the loop body
  - Control returns to the beginning of the **while**
  - The condition is reevaluated

Loop Repetition Condition

- Follows **while** reserved word
- Surrounded by parentheses
- When true, the loop body is repeated
- When false, exit the loop

Figure 5.1 Flowchart for a **`while`** loop



Loop Control Variable

- Initialize
- Test
- Update

while Statement Syntax

- Form

  while (*loop repetition condition*)

  statement;

- E.g.

```
countStar = 0;
while (countStar < n)
{
    cout << "*";
    countStar = countStar + 1;
}
```

Loop Notes

- If the loop control variable is not properly updated, an **infinite loop** can result.
- If the loop repetition condition evaluates to false the first time it’s tested, the loop body statements are *never* executed.

5.2 Accumulating a Sum or Product in a Loop

- Loops often accumulate a sum or product by repeating an addition of multiplication operation.

Listing 5.2 Program to compute company payroll

```
// File: computePay.cpp
// Computes the payroll for a company

#include <iostream>
using namespace std;

int main()
{
   int numberEmp;           // input - number of employees
   int countEmp;            // counter - current employee number
   float hours;             // input - hours worked
   float rate;              // input - hourly rate
   float pay;               // output - weekly pay
   float totalPay;          // output - company payroll

   // Get number of employees from user.
   cout << "Enter number of employees: ";
   cin >> numberEmp;

   // Process payroll for all employees.
   totalPay = 0.0;
   countEmp = 0;
   while (countEmp < numberEmp)
   {
```

Listing 5.2 Program to compute company payroll (continued)

```
      cout << "Hours: ";
      cin >> hours;
      cout << "Rate : $";
      cin >> rate;
      pay = hours * rate;
      cout << "Pay is $" << pay << endl << endl;
      totalPay = totalPay + pay;           // add next pay
      countEmp = countEmp + 1;
   }
   cout << "Total payroll is $" << totalPay << endl;
   cout << "All employees processed." << endl;

   return 0;
}
```

```
Enter number of employees: 3
Hours: 50
Rate : $5.25
Pay is $262.5
Hours: 6
Rate : $5
Pay is $30
Hours: 15
Rate : $7
Pay is $105
Total payroll is $397.5

All employees processed.
```

Example – Compute Payroll

- Initialization statements

```
totalPay = 0.0;     // pay accumulator
countEmp = 0;       // loop control variable that
                    // counts number of
                    // employees processed
```

- Accumulation

```
totalPay = totalPay + pay;     // add next pay
```

- Incrementation

```
countEmp = countEmp + 1;
```

Writing General Loops

- Process exactly 7 employees

```
while (countEmp < 7)
```

- Process an indefinite number of employees; number of employees must be read into variable numberEmp *before* the **while** statement executes

```
while (countEmp < numberEmp)
```

Multiplying a List of Numbers

```
product = 1;
while (product < 10000)
{
    cout << product << endl;  // display product so far
    cout << “Enter data item: “;
    cin >> item;
    product = product * item;  // update product
}
```

Conditional Loop

1. Initialize the loop control variable
2. While a condition involving the loop control variable is true
    3. Continue processing
    4. Update the loop control variable

Compound Assignment Operators

- General form of common operations

  variable = *variable* op *expression*;

- E.g.

```
countEmp = countEmp + 1;
time = time - 1;
totalPay = totalPay + pay;
product = product * item;
```

Special Assignment Operators

- += -= *= /= %=
- general form

  variable op= *expression*;
- E.g.

```
countEmp += 1;
time -= 1;
totalPay += pay;
product *= item;
```

The **for** Statement

- Especially useful for counting loops
- Form

```
for (initializing expression;
        loop repetition condition;
        update expression)
    statement;
```

The **for** Statement

- E.g.

```
for ( countStar = 0;
      countStar < N;
      countStar += 1)
   cout << “*”;
```

The **for** Statement

- E.g.

```
for (countStar = 0; countStar < N; countStar += 1)
    cout << “*”;
```

The **for** Statement

- E.g.

```
for (countStar = 0; countStar < N; countStar += 1)
   {
       cout << “*”;
   }
```

Listing 5.3 Using a `for` statement in a counting loop

```
// Process payroll for all employees.
totalPay = 0.0;
for (countEmp = 0;            // initialization
     countEmp < numberEmp;    // test
     countEmp += 1)           // update
{

   cout << "Hours: ";
   cin >> hours;
   cout << "Rate : $";
   cin >> rate;
   pay = hours * rate;
   cout << "Pay is $" << pay << endl << endl;
   totalPay += pay;       // accumulate total pay
}
cout << "Total payroll is $" << totalPay << endl;
cout << "All employees processed." << endl;
```

Formatting the **for** Statement

- Placement of expressions can be on one line or separate lines
- Body of loop indented
- Position of { } align with **for** keyword on separate lines (style for this book)

Increment and Decrement Operators

- ++ --
- Apply to a single variable
- **Side effect** - a change in the value of a variable as a result of carrying out an operation

Increment and Decrement Operators

- Prefix operator
  - E.g.
    ```
    m = 3;
    n = ++m;
    ```
- Postfix operator
  - E.g.
    ```
    m = 3;
    n = m++;
    ```
- Often used to update loop control variable

Listing 5.4 Function to compute factorial

```
// Computes factorial (n!)
// Pre: n is greater than or equal to zero
int factorial(int n)
{
   int product;    // accumulator for product computation
   product = 1;
   // Computes the product n x (n-1) x (n-2) x  . . .   x2
   for (int i = n; i > 1; i–)
     product = product * i;
   // Returns function result
   return product;
}
```

Localized Declarations of Variables

- Commonly used for loop control variables
- Declared at point of first reference
- Value has meaning (i.e. can be referenced) only inside loop.

Example - Localized Variables

```
string firstName;
cout << “Enter your first name: “’
cin >> firstName;
for (int posChar = 0;
        posChar < firstName.length( );
        posChar++;)
    cout << firstName.at(posChar) << endl;
```

Listing 5.5 Converting Celsius to Fahrenheit

```
// File: temperatureTable.cpp
// Conversion of celsius to fahrenheit temperature

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

int main()
{
   const int CBEGIN = 10;
   const int CLIMIT = -5;
   const int CSTEP = 5;
   float fahrenheit;
```

Listing 5.5 Converting Celsius to Fahrenheit (continued)

```
    // Display the table heading.
    cout << "Celsius" << "     Fahrenheit" << endl;

    // Display the table.
    for (int celsius = CBEGIN;
         celsius >= CLIMIT;
         celsius -= CSTEP)
    {
      fahrenheit = 1.8 * celsius + 32.0;
      cout << setw(5) << celsius
           << setw(15) << fahrenheit << endl;
    }

    return 0;
}
```

```
Celsius   Fahrenheit
   10        50.00
    5        41.00
    0        32.00
   -5        23.00
```

Output - Celsius to Fahrenheit

```
Celsius      Fahrenheit
   10     50.00
    5     41.00
    0     32.00
   -5     23.00
```

Displaying a Table of Values

- setw( ) manipulator helps create neat columns
- It is a member function of the **iomanip** class.
- Requires the **iomanip** library to be included

Conditional Loops

- Used when you can’t determine before loop execution begins exactly how many loop repetitions are needed.
- The number of repetitions is generally stated by a condition that must remain true in order for the loop to continue.

Conditional Loop

```
Initialize the loop control variable.
While a condition involving the loop control
   variable is true
    Continue processing.
    Update the loop control variable
```

Case Study: Monitoring Oil Supply

- Problem We want to monitor the amount of oil remaining in a storage tank at the end of each day. The initial supply of oil in the tank and the amount taken out each day are data items. Our program should display the amount left in the tank at the end of each day and it should also display a warning when the amount left is less than or equal to 10 percent of the tank's capacity. At this point, no more oil can be removed until the tank is refilled.

Case Study: Analysis

- Clearly, the problem inputs are the initial oil supply and the amount taken out each day. The outputs are the oil remaining at the end of each day and a warning message when the oil left in the tank is less than or equal to 10 percent of its capacity.

Case Study: Data Requirements

- Problem Constants

  CAPACITY = 1000 // tank capacity

  MINPCT = 0.10 // minimum %

- Problem Input

  float supply // initial oil supply

  Each day's oil use

Case Study: Data Requirements

- Problem Output

  float oilLevel // final oil amount

  Each day's oil supply

  A warning message when the oil supply is less than minimum.

Case Study: Data Requirements

- Program Variable

  float minOil   // minimum oil supply

- Formulas

  Minimum oil supply is 10 percent of tank's capacity

Case Study: Initial Algorithm

1. Get the initial oil supply.
2. Compute the minimum oil supply.
3. Compute and display the amount of oil left each day (implement as function monitorOil).
4. Display the oil left and a warning message if necessary.

Analysis for Function monitorOil

- Function monitorOil must display a table showing the amount of oil left at the end of each day. To accomplish this, the function must read each day’s usage and deduct that amount from the oil remaining. The function needs to receive the initial oil supply and the minimum oil supply as inputs (arguments) from the main function.

Function Interface for monitorOil

- Input Parameters

  float supply // initial oil supply

  float minOil // minimum oil supply

- Output

  Returns the final oil amount

- Local Data

  float usage // input from user - each day's oil use

  float oilLeft // output from user - each day's oil supply

Design of monitorOil

- The body of **monitorOil** is a loop that displays the oil usage table. We can't use a counting loop because we don't know in advance how many days if will take to bring the supply to the critical level. We do know the initial supply of oil, and we know that we want to continue to compute and display the amount of oil remaining (**oilLeft**) as long as the amount of oil remaining does not fall below the minimum. So the loop control variable must be **oilLeft**. We need to initialize **oilLeft** to the initial supply and to repeat the loop as long as **oilLeft > minOil** is true. The update step should deduct the daily usage (a data value) from **oilLeft**.

Initial Algorithm for monitorOil

```
1. Initialize oilLeft to supply.
2. While (oilLeft > minOil)
    2.1 Read in the daily usage.
    2.2 Deduct the daily usage from oilLeft
    2.3 Display the value of oilLeft
```

Listing 5.6 Program to monitor oil supply

```
// File: oilSupply.cpp
 Displays daily usage and amount left in oil tank.
#include <iostream>
using namespace std;

float monitorOil(float, float);

int main()
{
   const float CAPACITY = 10000;  // tank capacity
   const float MINPCT = 10.0;  // minimum percent

    float supply;    // input - initial oil supply
    float oilLeft;        // output - oil left in tank
    float minOil;        // minimum oil supply
```

Listing 5.6 Program to monitor oil supply (continued)

```
   // Get the initial oil supply.
    cout << "Enter initial oil supply: ";
    cin >> supply;

   // Compute the minimum oil supply.
   minOil = CAPACITY * (MINPCT / 100.0);

    // Compute and display the amount of oil left each day
    oilLeft = monitorOil(supply, minOil);

    // Display warning message if supply is less than minimum
    cout << endl << oilLeft << " gallons left in tank."
      << endl;
   return 0;
}
```

Listing 5.6 Program to monitor oil supply (continued)

```
float monitorOil(float supply, float minOil)
{
   // Local data . . .
   float usage;     // input from user - Each day's oil use
   float oilLeft;  // Amount left each day

   oilLeft = supply;
   while (oilLeft > minOil)
   {
      cout << "Enter amount used today: ";
      cin >> usage;
      oilLeft -= usage;
      cout << "After removal of " << usage << " gallons, ";
      cout << "number of gallons left is " << oilLeft
           << endl << endl;
   }
   return oilLeft;
}
```

Case Study: Testing

- To test the program, try running it with a few samples of input data. One sample should bring the oil level remaining to exactly 10 percent of the capacity. For example, if the capacity is 10,000 gallons, enter a final daily usage amount that brings the oil supply to 1,000 gallons and see what happens.

Case Study: Testing

```
Enter initial oil supply: 7000
Enter amount used today: 1000
After removal of 1000 gallons, number of gallons left is 6000

Enter amount used today: 4000
After removal of 4000 gallons, number of gallons left is 2000

Enter amount used today: 1500
After removal of 1500 gallons, number of gallons left is 500

500 gallons left in tank
Warning - amount of oil left is below minimum!
```

Thank You

For

Your Attention!