Control Structures

Selection structures && Repetition structures

Jessica Sizemore SIP 2015 , 16 JUN 2015

Selection Structures

Using `If` and `else`

Overview

- A *selection structure* is needed when a decision must be made (based on some condition) before selecting an instruction to execute
- Must be phrased as a Boolean expression (evaluates to true or false)

Example: Entering the correct or incorrect password into your computer.

- **Single-alternative selection structure**
  - Set of instructions is executed only if condition evaluates to true
- **Dual-alternative selection structure**
  - Executes different sets of instructions based on whether condition evaluates to true or false

Overview

- **True path**
  - Instructions followed when condition evaluates to true
  - Begins with *if*
- **False path**
  - Instructions followed when condition evaluates to false
  - Begins with *else*

Coding a Selection Structure in C++

- Syntax

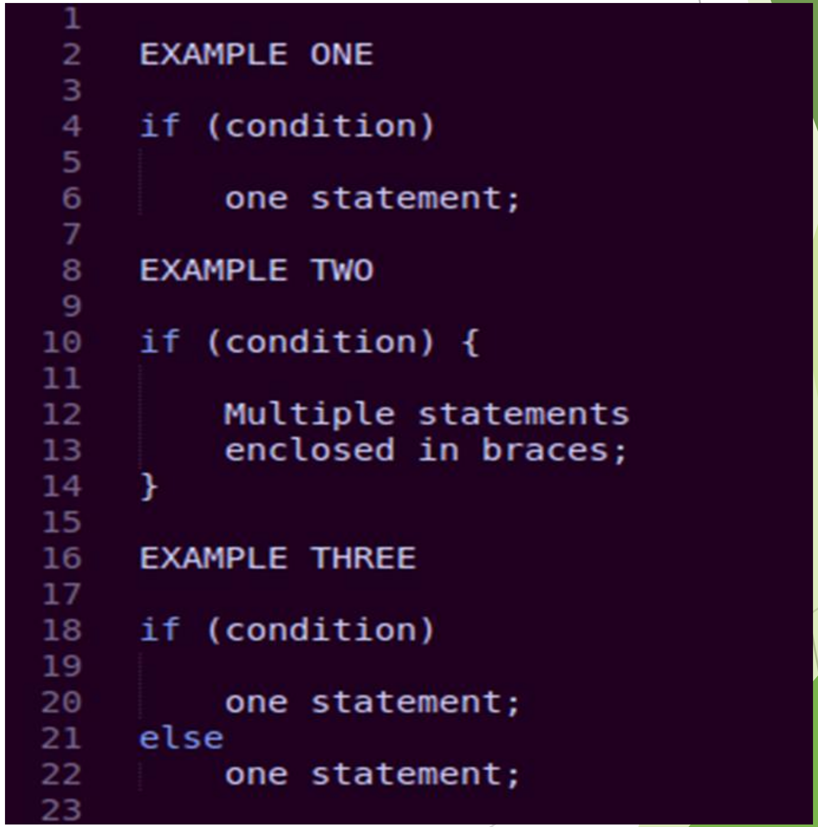
`if` ***(condition)*** ** Required

one or more statements; (true path)

`else` ** Required

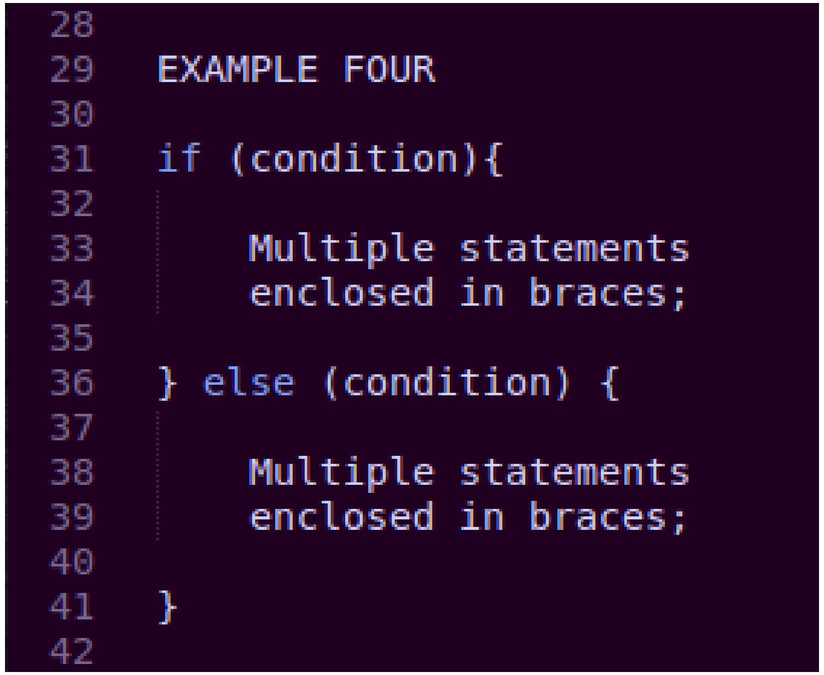
one or more statements; (false path)

- If a path contains more than one statement, the statement much be entered as a Statement Block, meaning they must be enclosed in a set of braces… {example}



```

EXAMPLE ONE

if (condition)

    one statement;

EXAMPLE TWO

if (condition) {

    Multiple statements
    enclosed in braces;
}

EXAMPLE THREE

if (condition)

    one statement;
else
    one statement;

```

Coding a Selection Structure in C++

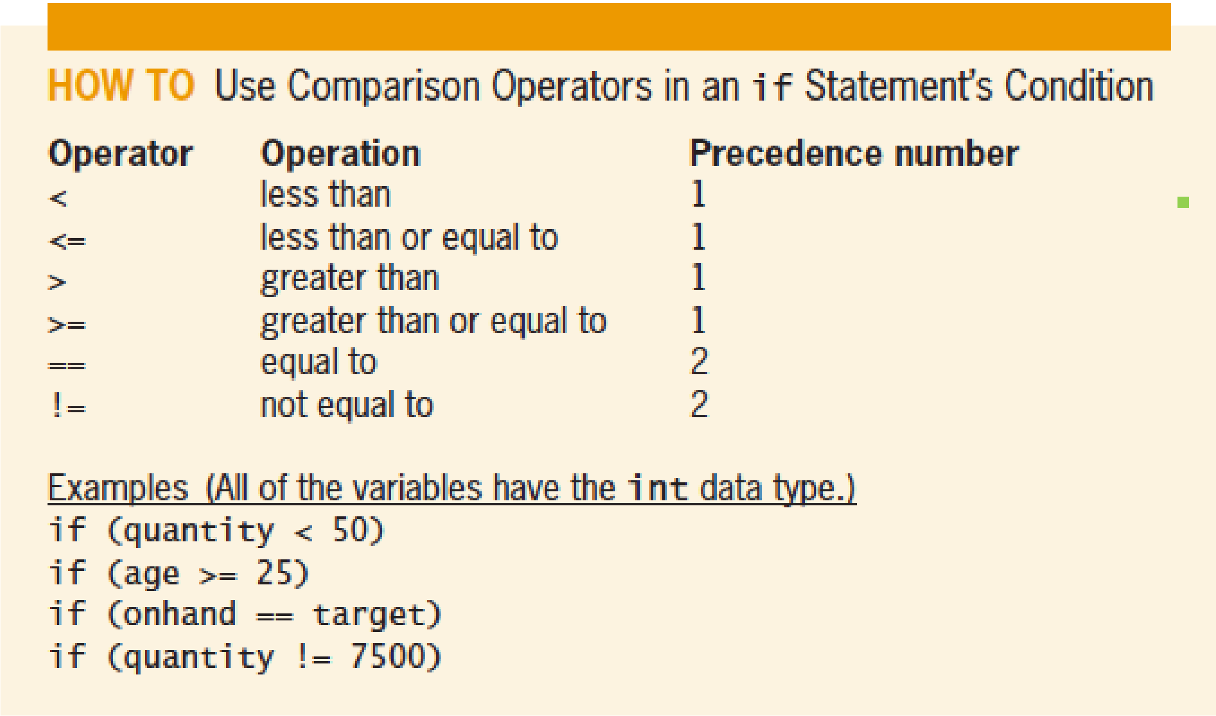


```

EXAMPLE FOUR

if (condition){

    Multiple statements
    enclosed in braces;

} else (condition) {

    Multiple statements
    enclosed in braces;

}

```

- The condition must be a Boolean expression. They may contain variables, constants, arithmetic operators, comparison operators, and logical operators

- You are able to use Single and Multiple Statements where need. It is always better to use braces if you are unsure whether to use them or not.

Using Comparison Operators



HOW TO Use Comparison Operators in an `if` Statement's Condition

Operator	Operation	Precedence number
<	less than	1
<=	less than or equal to	1
>	greater than	1
>=	greater than or equal to	1
==	equal to	2
!=	not equal to	2

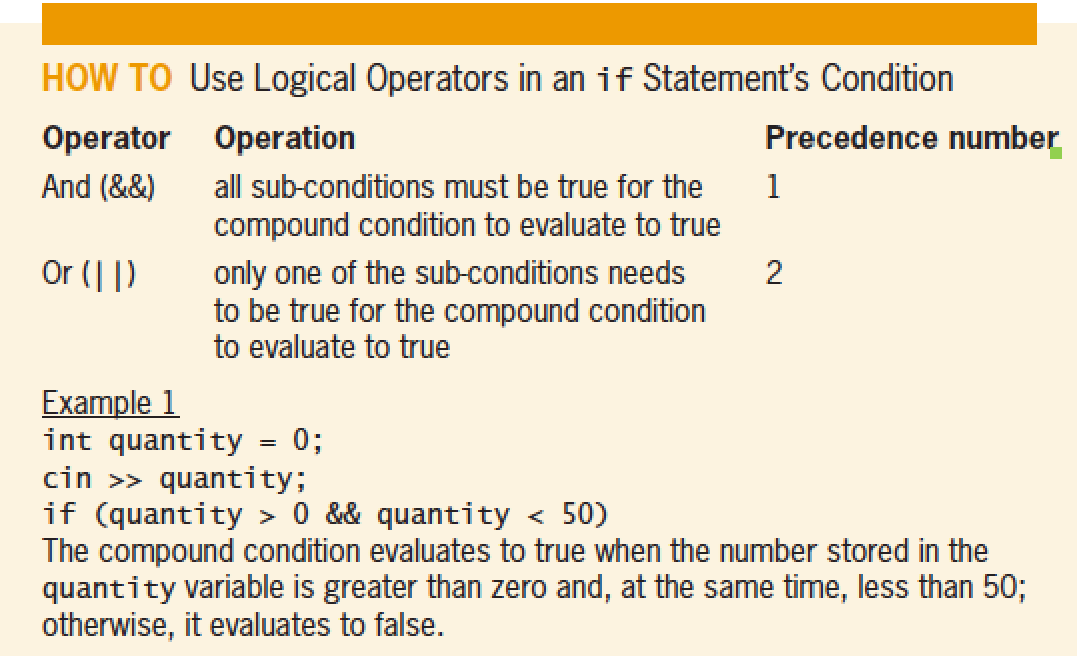
Examples (All of the variables have the `int` data type.)

```
if (quantity < 50)
if (age >= 25)
if (onhand == target)
if (quantity != 7500)
```

- A variable that can be used only within the statement block in which it is defined is referred to as a ***local variable***.

Figure 5-8 How to use comparison operators in an if statement's condition

Using Logic Operators in If Statements



HOW TO Use Logical Operators in an `if` Statement's Condition

Operator	Operation	Precedence number
And (&&)	all sub-conditions must be true for the compound condition to evaluate to true	1
Or (||)	only one of the sub-conditions needs to be true for the compound condition to evaluate to true	2

Example 1

```
int quantity = 0;
cin >> quantity;
if (quantity > 0 && quantity < 50)
```

The compound condition evaluates to true when the number stored in the `quantity` variable is greater than zero and, at the same time, less than 50; otherwise, it evaluates to false.

Logical Operators allow you to combine two or more conditions; Also called ***Boolean Operators*** because they always evaluate true or false.

Figure 5-15 How to use logical operators in an if statement's condition

Using Logic Operators in If Statements

- Example One

```
if ( quantity > 0 && quantity < 50 );
```

  - Is true when the quantity is greater than 0 AND also less then 50

- Example Two

```
if ( age == 21 || age > 55);
```

  - Is true when age is equal to 21 OR greater than 55

- Example Three

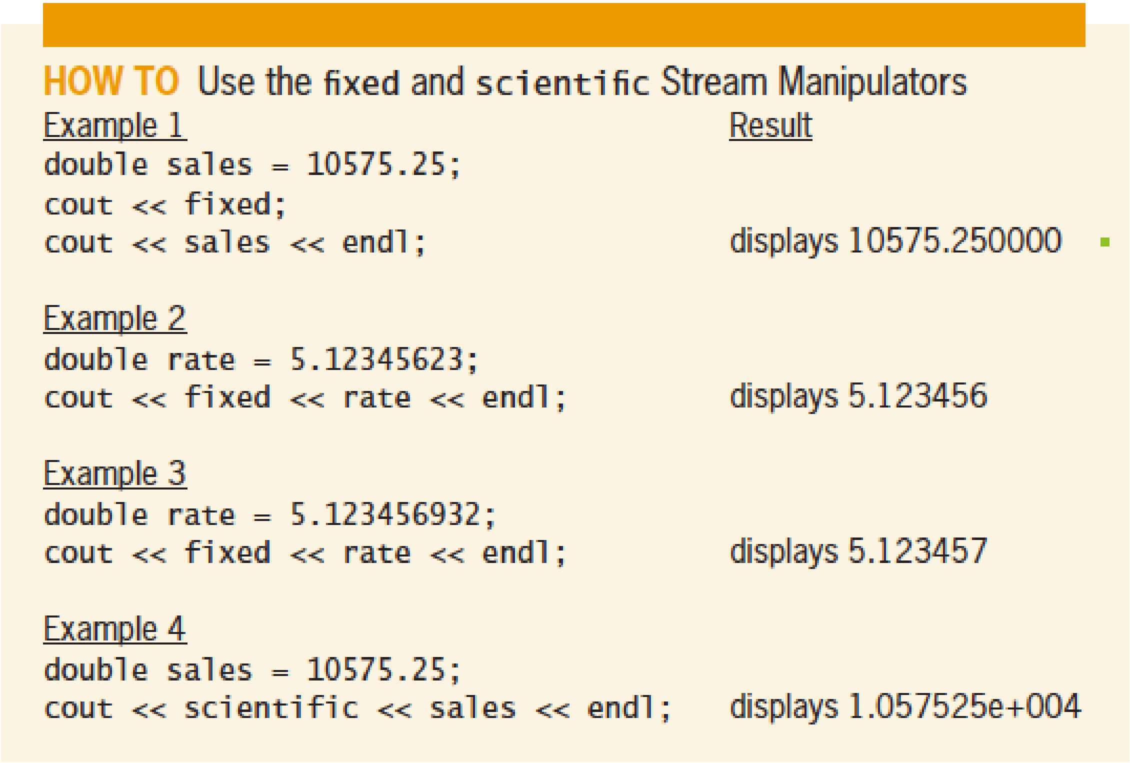
```
if ( quantity > 0 && quantity < 100 || price > 34.55);
```

  - When is this true?

Formatting Numeric Output

- Real numbers are displayed in either `fixed` or `scientific` (e) notation
- Small numbers can be displayed in `fixed` notation
- Large numbers can be displayed in `scientific` (e) notation
- **`fixed`** and **`scientific`** stream manipulators can appear alone in a **`cout`** statement but must be declared before the numbers that you want format
- Manipulators are in effect until the end of the program

Formatting Numeric Output



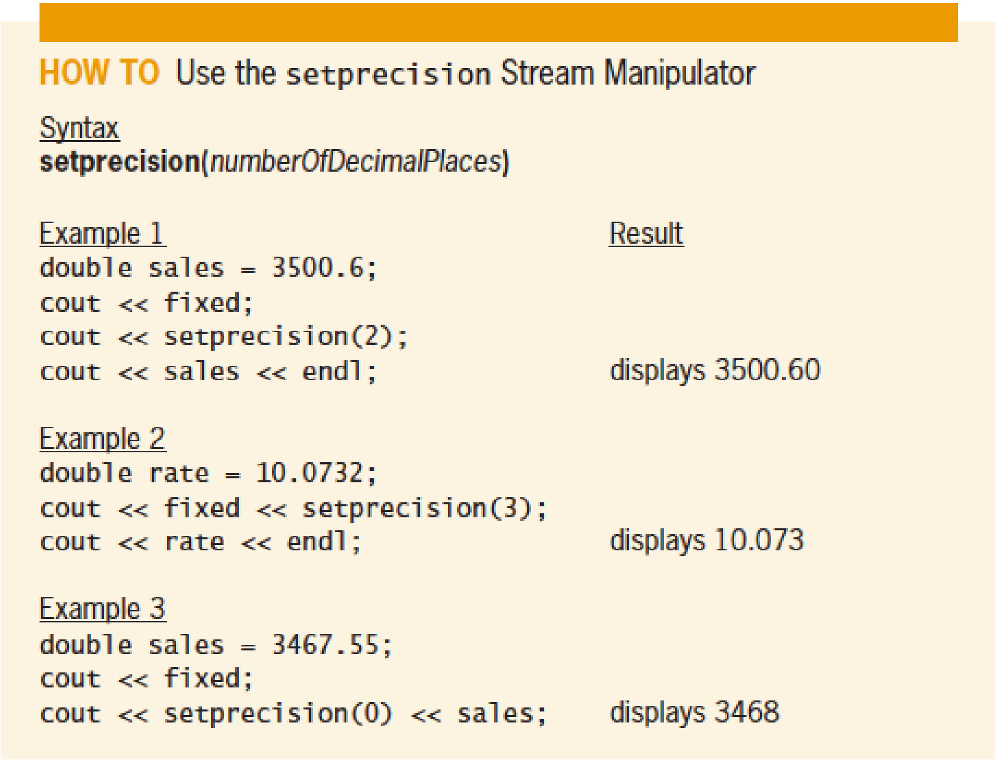
HOW TO Use the fixed and scientific Stream Manipulators

```
Example 1                                  Result
double sales = 10575.25;
cout << fixed;
cout << sales << endl;                     displays 10575.250000

Example 2
double rate = 5.12345623;
cout << fixed << rate << endl;             displays 5.123456

Example 3
double rate = 5.123456932;
cout << fixed << rate << endl;             displays 5.123457

Example 4
double sales = 10575.25;
cout << scientific << sales << endl;       displays 1.057525e+004
```

- Numbers that are formatted with `fixed` notation will have SIX numbers to the right of the decimal place
  - EX. 123.456 is displayed as 123.456000
  - EX. 123.3456789 is displayed as 123.345679

Figure 5-25 How to use the fixed and scientific stream manipulators

Formatting Numeric Output



HOW TO Use the `setprecision` Stream Manipulator

Syntax

setprecision(*numberOfDecimalPlaces*)

```
Example 1                               Result
double sales = 3500.6;
cout << fixed;
cout << setprecision(2);
cout << sales << endl;                  displays 3500.60

Example 2
double rate = 10.0732;
cout << fixed << setprecision(3);
cout << rate << endl;                   displays 10.073

Example 3
double sales = 3467.55;
cout << fixed;
cout << setprecision(0) << sales;       displays 3468
```

Figure 5-26 How to use the setprecision stream manipulator

- **`setprecision` stream manipulator** controls the number of decimal places that appears when a real number is displayed
- Definition of **`setprecision`** manipulator contained in **`iomanip`** file
- Program must contain:

```
#include <iomanip>
```

Repetition Structures

Using **`While`** and **`for`**

What is a Repetition Structure?

- **Repetition structure**, or **loop**, processes one or more instructions repeatedly
- Every loop contains a Boolean condition that controls whether the instructions are repeated

- A **looping condition** says whether to continue looping through instructions
- A **loop exit condition** says whether to stop looping through the instructions

What is a Repetition Structure?

- The instructions that are inside a loop, and are told to repeat are called the ***loop body***

- A loop body can contain pretest or posttest
  - **Pretest loop**, the condition is evaluated *before* the instructions in the loop are processed
  - **Posttest loop**, the condition is evaluated *after* the instructions in the loop are processed
- In both cases, the condition is evaluated with each repetition

Using Pretest Loops

- Some loops require the user to enter a special **sentinel value** to end the loop to quit the loop

- When a loop's condition evaluates to **`TRUE`**, the instructions in the loop body are processed
- When a loop's condition evaluates to **`FALSE`**, the instructions are skipped and processing continues with the first instruction after the loop

Using Pretest Loops

- After each time the loop body runs through its instructions, the loop's condition is then reevaluates to determine if the loop should be processed again

- A **priming read** is an instruction that appear before the loop; It is set up with an initial value that is entered by a user
- An **update read** is an instruction that is written within a loop that allows the user to enter a new value each time the loop is processed

The `while` Statement

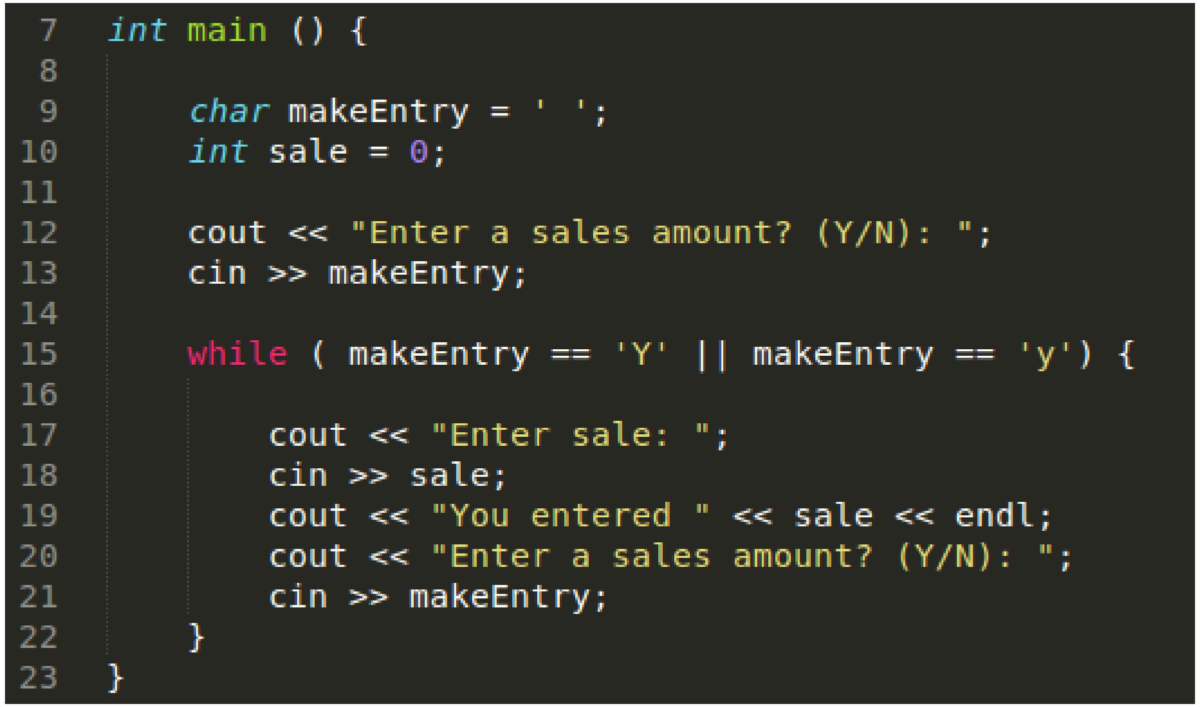
- Syntax is:

  `while` ***(condition)***

  - one statement or a statement block to be processed as long as the condition is true

- The **`while`** statement can be used to code a pretest loop
- The condition must be supplied as a Boolean expression
- A loop whose instructions are processed indefinitely is called an **infinite loop** or **endless loop**
- You can usually stop a program that has entered an infinite loop by pressing **Ctrl+c**

The while Statement



```cpp
int main () {

    char makeEntry = ' ';
    int sale = 0;

    cout << "Enter a sales amount? (Y/N): ";
    cin >> makeEntry;

    while ( makeEntry == 'Y' || makeEntry == 'y') {

        cout << "Enter sale: ";
        cin >> sale;
        cout << "You entered " << sale << endl;
        cout << "Enter a sales amount? (Y/N): ";
        cin >> makeEntry;
    }
}
```

Counters and Accumulators

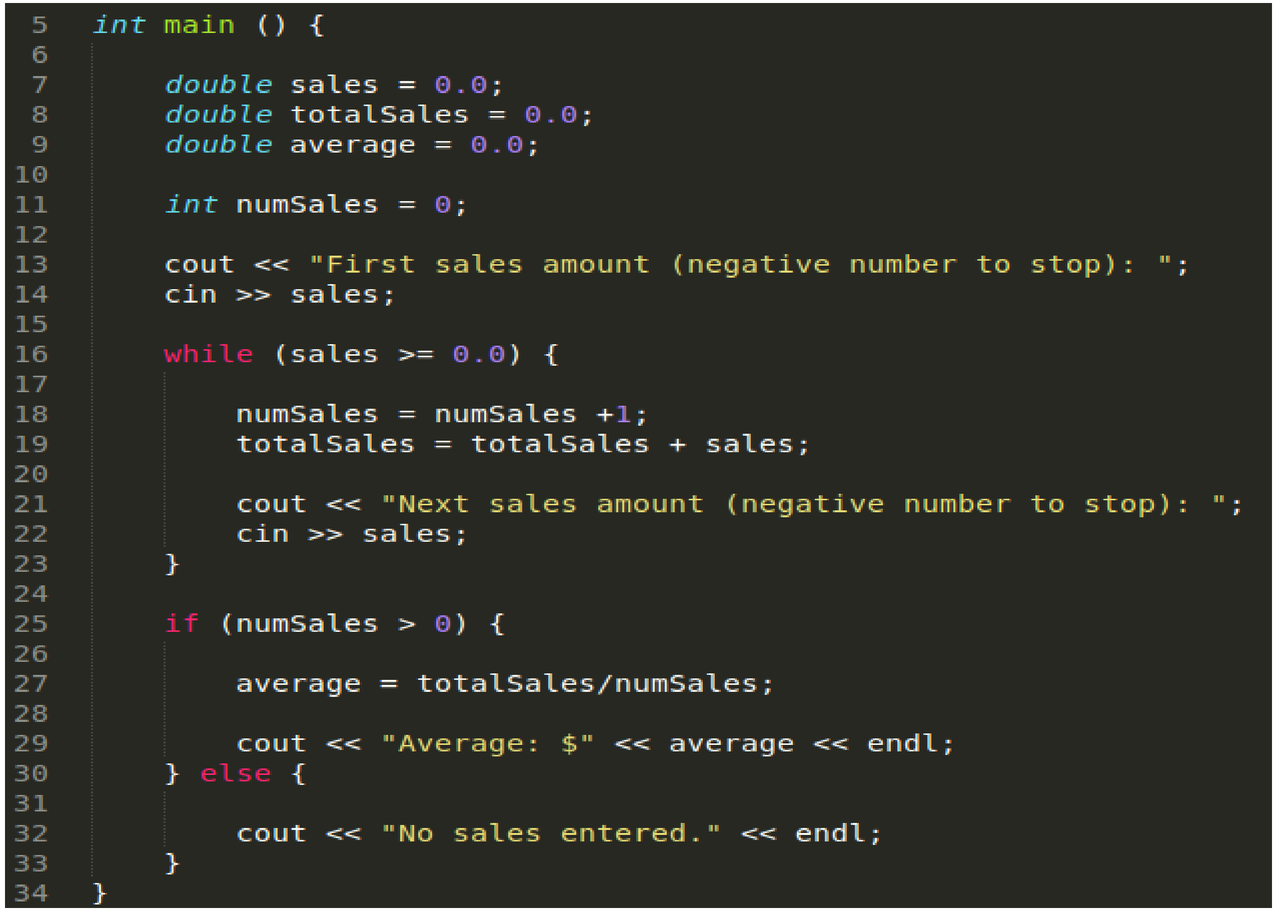
- You may be asked to calculate a total average, to do this you use a counter and/or an accumulator.

- A **counter** is a numeric variable used for counting something
- An **accumulator** is a numeric variable used for accumulating (adding together) multiple values
- Two tasks are associated with counters and accumulators:
  - **initializing and updating**

Counters and Accumulators

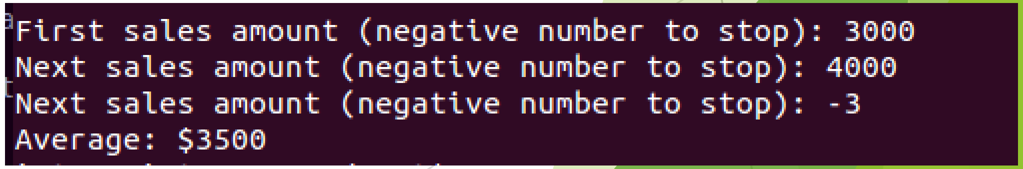
- **Initializing** means assigning a beginning value to a counter or accumulator (usually 0) – happens once, before the loop is processed
- **Updating** (or **incrementing**) means adding a number to the value of a counter or accumulator
  - A **counter** is updated by a constant value (usually 1)
  - An **accumulator** is updated by a value that varies
- Update statements are placed in the body of a loop since they must be performed at each iteration

Average Sales Amount Calculator…



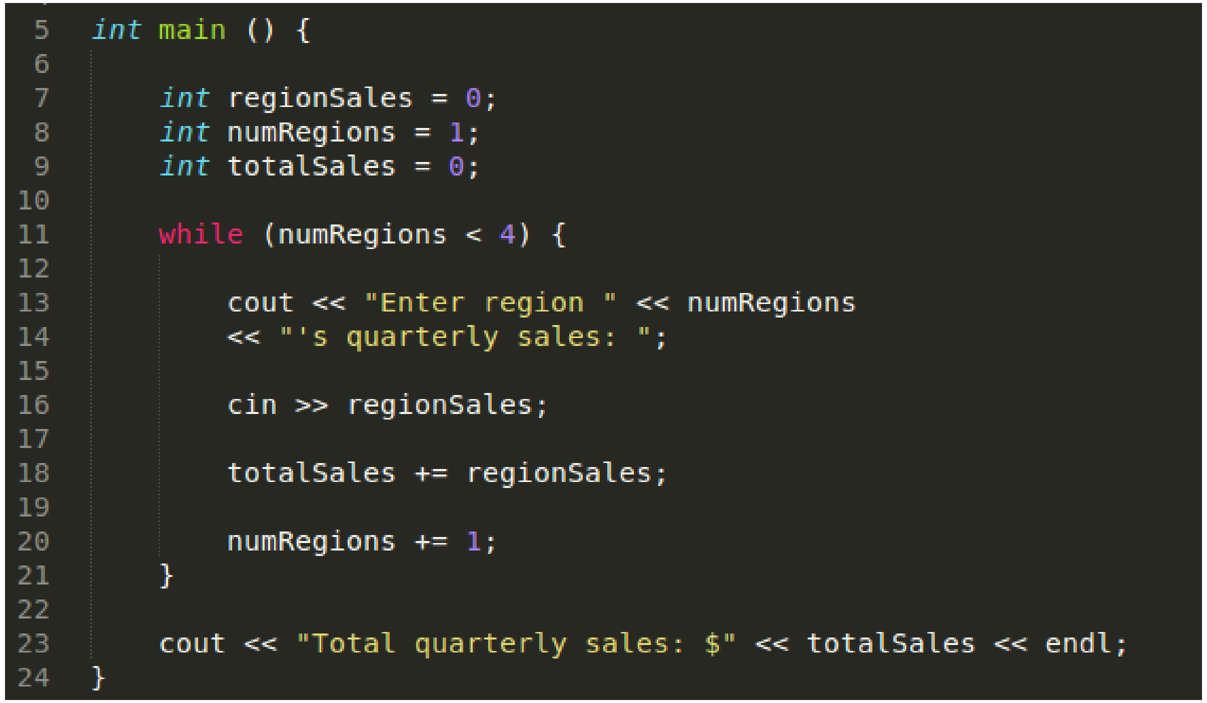
```cpp
int main () {

    double sales = 0.0;
    double totalSales = 0.0;
    double average = 0.0;

    int numSales = 0;

    cout << "First sales amount (negative number to stop): ";
    cin >> sales;

    while (sales >= 0.0) {

        numSales = numSales +1;
        totalSales = totalSales + sales;

        cout << "Next sales amount (negative number to stop): ";
        cin >> sales;
    }

    if (numSales > 0) {

        average = totalSales/numSales;

        cout << "Average: $" << average << endl;
    } else {

        cout << "No sales entered." << endl;
    }
}
```

- Uses a counter to keep track of the number of sales entered and an accumulator to keep track of the total sales
- Both are initialized to 0
- The loop ends when the user enters a sentinel value (-1)



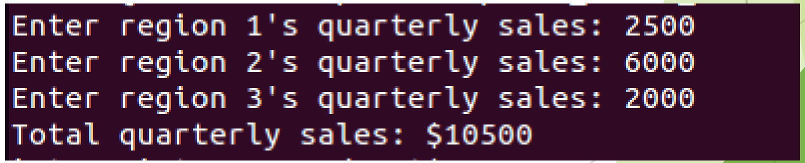
```
First sales amount (negative number to stop): 3000
Next sales amount (negative number to stop): 4000
Next sales amount (negative number to stop): -3
Average: $3500
```

Average Sale Calculator with Counter-Controlled Loops



```
int main () {

    int regionSales = 0;
    int numRegions = 1;
    int totalSales = 0;

    while (numRegions < 4) {

        cout << "Enter region " << numRegions
        << "'s quarterly sales: ";

        cin >> regionSales;

        totalSales += regionSales;

        numRegions += 1;
    }

    cout << "Total quarterly sales: $" << totalSales << endl;
}
```

- Counter-controlled loop is used that totals the quarterly sales from three regions
- Loop repeats three times, once for each region, using a counter to keep track



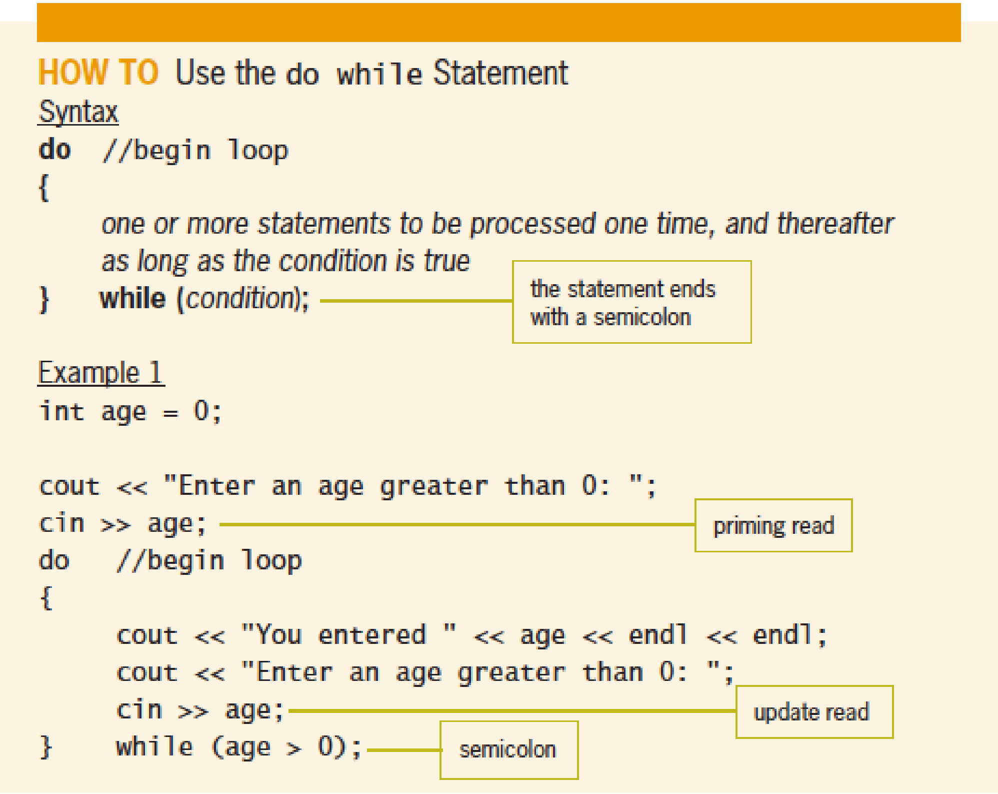
```
Enter region 1's quarterly sales: 2500
Enter region 2's quarterly sales: 6000
Enter region 3's quarterly sales: 2000
Total quarterly sales: $10500
```

The `do while` Statement

- **`do while`** statement is used to code posttest loops in C++
- Syntax:

```
 do {
     one or more statements to be processed one time,
     and thereafter as long as the condition is true
} while (condition);
```

The `do while` Statement



HOW TO Use the `do while` Statement

Syntax

```
do   //begin loop
{
     one or more statements to be processed one time, and thereafter
     as long as the condition is true
}    while (condition);
```

Example 1

```
int age = 0;

cout << "Enter an age greater than 0: ";
cin >> age;
do    //begin loop
{
      cout << "You entered " << age << endl << endl;
      cout << "Enter an age greater than 0: ";
      cin >> age;
}     while (age > 0);
```

the statement ends with a semicolon
priming read
update read
semicolon

Figure 5-15 How to use logical operators in an `if` statement's condition

- Programmer must provide loop *condition*
  - Must evaluate to a Boolean value
  - May contain variables, constants, functions, arithmetic operators, comparison operators, and logical operators
- Programmer must also provide statements to be executed when *condition* evaluates to true
- Braces are required around statements if there are more than one

The `for` Statement

- The **`for`** statement can also be used to code any pretest loop
- Syntax:

  `for` ***([initialization]; condition; [update])***

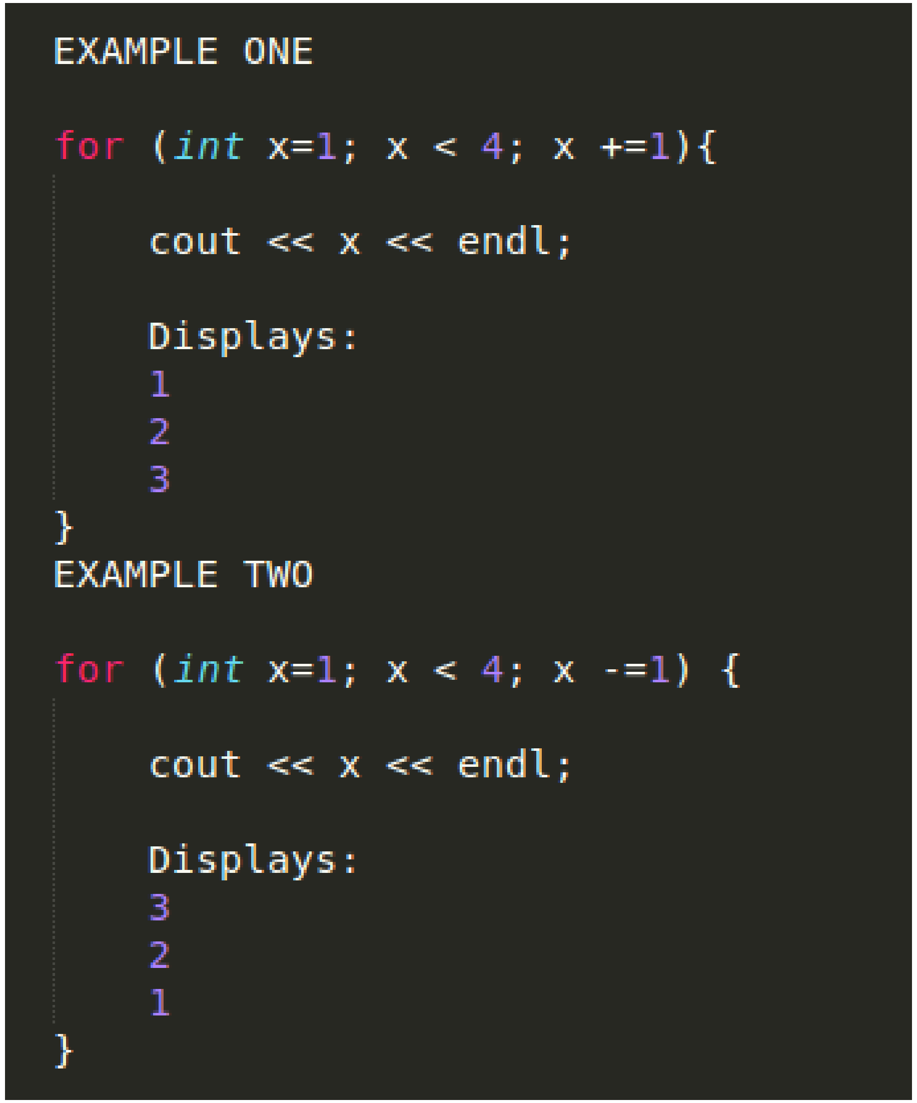
  - *one statement or a statement block to be processed as long as the condition is true*

- ***Initialization*** and ***update*** arguments are optional

The `for` Statement

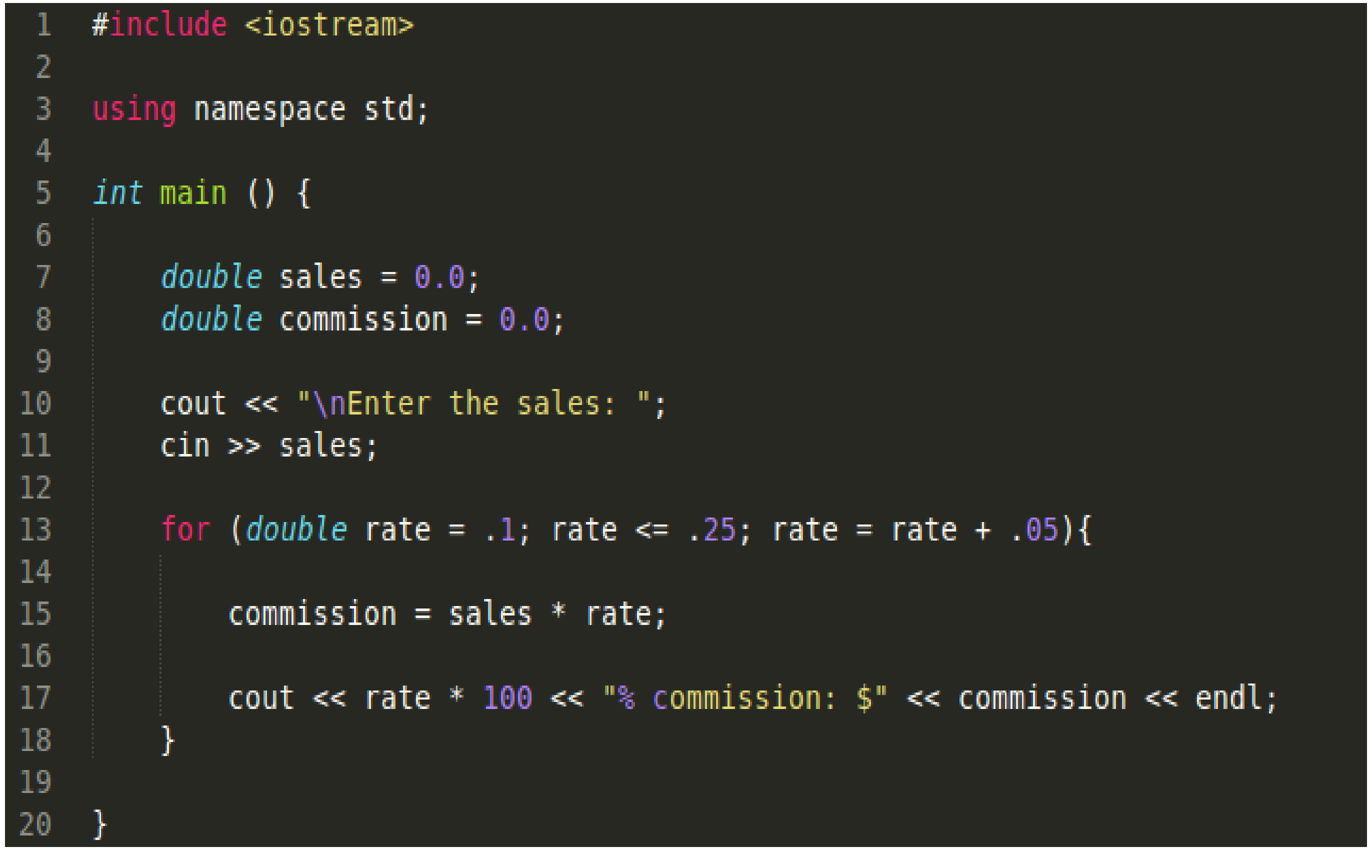
- ***Initialization*** argument usually creates and initializes a counter variable
  - Counter variable is local to **`for`** statement
- ***Condition*** argument specifies condition that must be true for the loop body to be processed; loop ends when it evaluates to false
  - Condition must be a Boolean expression
- ***Update*** argument usually contains an expression that updates the counter variable

The for Statement



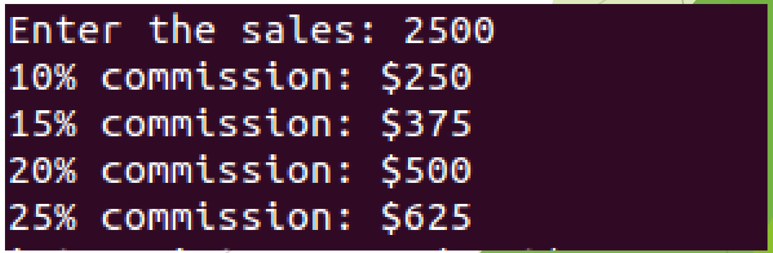
```
EXAMPLE ONE

for (int x=1; x < 4; x +=1){

    cout << x << endl;

    Displays:
    1
    2
    3
}
EXAMPLE TWO

for (int x=1; x < 4; x -=1) {

    cout << x << endl;

    Displays:
    3
    2
    1
}
```

The Colfax Sales Program



```cpp
#include <iostream>

using namespace std;

int main () {

    double sales = 0.0;
    double commission = 0.0;

    cout << "\nEnter the sales: ";
    cin >> sales;

    for (double rate = .1; rate <= .25; rate = rate + .05){

        commission = sales * rate;

        cout << rate * 100 << "% commission: $" << commission << endl;
    }

}
```

- Calculates the commission for a given sales amount using four different rates
- A `for` loop keeps track of each of the four rates



```
Enter the sales: 2500
10% commission: $250
15% commission: $375
20% commission: $500
25% commission: $625
```

Work Cited

- Diane Zax, “An Introduction to Programming with C++, Sixth Edition”,
  - Chapter 5 – The Selection Structure.
  - Chapter 7 – The Repetition Structure.
- Towson University, Professor Robert Eyer, COSC 175,
  - Chapter 5 Lecture Slides.
  - Chapter 7 Lecture Slides.