1. C++ Overview

- C++ Program Structure.
- Data Types.
- Assignment Statements.
- Input/Output Operations.
- Arithmetic Expressions.
- Interactive Mode, Batch Mode and Data Files.
- Common Programming Errors.
- Selection Statements.
- Repetition Statements.
- Functions.

C++ Program Structure

```
// comments

#include directives

using namespace region ;

int main( )

{

    Variable and Constant Declarations

    Executable C++ statements

    return 0;

}
```

Comments

- Comments make a program easier to understand
- Ignored by the C++ compiler
- // used to signify a comment on a single line
- /* */ used if comments on multi lines
- Do not embed comments within /* */ comments
- Example :
  - / / miles.cpp
  - /* Converts distances from miles to kilometers */

Compiler Directives

- #include
  - Compiler directive
  - Processed before translation of the program
  - Instructs compiler on what you want in the program
- #include <filename>
  - Used with <> - part of the C++ standard library
  - Adds library files to the program

Compiler Directives

- #include <iostream>
  - Defined in iostream class
  - Entered on the keyboard (cin)
  - Displayed on monitor (cout)
- Using namespace region ;
  - Program uses objects defined in the namespace called region
  - Should follow the #include lines
  - Example : using namespace std ;
  - C++ standard library defined in std namespace

The #define Compiler Directive

- An additional way to define constants
- Form : **#define identifier value**
- Causes all references to identifier to be replaced by value during compilation
- No storage in memory
- Constant declaration : storage in memory
- Example :
  - #define PI 3.14159
  - const float PI = 3.14159 ;

Function main

- Function basic unit (collection of related statements)
- A C++ program must contain a main function
- Beginning of the program execution
- main - lower case with ( )
- { } - braces define the function body
- int - function returns integer value
- return 0 ; - the program returns 0 value to the operating system

Declaration Statements

- Problem analysis : data requirements
- Each data needed must be declared
- Comma used to separate data identifiers
- Example : float miles, kms ;

Executable Statements

- Executable statements
  - Read and write statements
  - Assignment statements
  - Function calls
  - Selection statements
  - Repetition statements
  - Example : cout << "Enter the distance in miles : " ;

Reserved Words and Identifiers

- Reserved words (keywords)

  - a word with a specific meaning in C++

  - can NOT be used for other purposes (const, float and int are some examples)

- Identifiers (variables and constants)

  - name data and objects manipulated by the program (user defined)

  - only letters, digits and underscores

  - always begin with a letter or an underscore

  - valid identifiers - letter, letter1, _letter

Reserved Words and Identifiers

- Special symbols
- C++ has rules for special symbols

  - = * ; { } ( ) // << >>

- Appendix B
  - Examples of reserved words
  - Special characters
- C++ case sensitive
  - Compiler differentiates upper & lower case
  - Identifiers can be either
  - Be careful though (cost != Cost)

Variables and Constants

- Variable : a symbolic name for a memory cell that can hold a single data value at a time. This data value can be changed during the execution of the program
- Constant : symbolic name for a single data value that can't be changed during the execution of the program
- Variables and constants have specific data types, that is, a variable/constant can't be assigned any data value

Data Types

- Data types : a set of values and a set of operations

1- Predefined in the language

- integers - positive or negative whole numbers
- real numbers - positive or negative decimal numbers
- booleans - with two values true and false
- characters - all symbols of the keyboard

2 - Defined in the standard library

- string - a sequence of characters

Integers

- The basic integer type is int
- The number of bytes allocated for storing an integer limits the range of integers that can be represented
- On PC it is normally 16 or 32 bits
- Other integer types
  - short: typically uses less bits
  - long: typically uses more bits
- INT_MIN : smallest value of type int

INT_MAX : largest value of type int

Real Numbers

- Floating-point types represent real numbers
- Integer part and Fractional part
- The number 108.1517 breaks down into the following part :
  - 108 - integer part and 1517 - fractional part
- C++ provides three floating-point types
  - float
  - double
  - long double
- To represent very large and very small numbers :
  - 1.23e5, 1.23e+5, 0.34e-4

Internal Representations of Numeric Data

- Integers and real numbers are stored differently in memory
- Fixed-Point Format
  - integers stored as a sequence of 0 and 1
  - the leftmost bit contains the sign (+ or -)
- Floating-Point Format
  - real number = mantissa $\times$ $10^{exponent}$
  - in memory : sign-mantissa-exponent in binary form

Ranges for int and float Types

- See table 7.1
- These constants defined in the **climits** and **cfloat** libraries
- The actual values of these constants will vary from computer to computer
- Example
  - CHAR_BIT equals 8 defined in climits library
  - FLT_MAX equals 1.0e+37 defined in cfloat library

Numerical Inaccuracies

- Representational error
  - a floating-point number cannot be represented exactly
  - computers don't perform floating point calculations precisely
  - example : 1.0/3.0 equals 0.333333... depends on the number of bits used in mantissa
- Cancellation error
  - applying an operation on 2 operands of very different magnitudes
  - the large number cancels out the small number
  - example : 1000.0 + 0.00001234 equals 1000.0

Numerical Inaccuracies

- Arithmetic overflow
  - computation is too large to be represented
  - example : adding 2 very large numbers
- Arithmetic underflow
  - computation is too small to be represented
  - example : multiplying 2 very small fractions
- Run-Time errors

Characters

- char
- Individual character value (letter, digit or special symbol)
- Character literal enclosed in single quotes
- 'A', '2', '*', ':' , ' '
- Each character in represented by one byte

ASCII Code

- ASCII is the dominant encoding scheme
- Examples

’ ’ encoded as 32

’+’ encoded as 43

’A’ encoded as 65

’Z’ encoded as 90

’a’ encoded as 97

’z’ encoded as 122

- Appendix A : ASCII Character Set

Character Data Type

- Each character has an numeric code
- Storage of a character value : binary form of its code
- ASCII code uses 8 bits to represent characters
- Relational operators used to compare character values

(comparaison based on ASCII code)

== (equal to), **!=** (not equal to), $<=$, $<$, $>=$ and $>$

- Example
  - ’0’ $<=$ ’9’ is true
  - ’a’ $>=$ ’A’ is true
  - ’*’ $<$ ’ ’ is false

Character Functions

- *cctype* library provides many functions to the programmer
  - convert the case of a letter
  - test the kind of a character
- int(c) returns the code of the character c
  - int('*') equals 42
- char(x) returns the corresponding character of the integer x
  - char(35) equals '#'
- see table 7.2 : tolower(c) returns the lowercase of a uppercase letter
  - tolower('Z') equals 'z'

string Class

- string object data type
- A literal string constant is a sequence of zero or more characters enclosed in double quotes
  - ”Are you aware? \ n”
- Individual characters of string are stored in consecutive memory locations
- The null character (’\0’) is appended to strings so that the compiler knows where in memory strings ends

string Class

- String literal
  - ”A”
  - ”1234”
  - “Enter the distance”
- Additional data types included in library
- #include <string>
- Various operations on strings

Boolean Data Type

- Used in assignment statements and logical expressions (true and false)
- Logical operators
  - ! operator not
  - && operator and
  - || operator or

- ! used to form the complement of a boolean expression

Relational Operator	**Complement**
$<$	$>=$
$<=$	$>$
$>$	$<=$
$>=$	$<$
==	!=
!=	==

Logical Expression	**Complement**
(expr && expr)	!expr || !expr
(expr || expr)	!expr && !expr

- Type bool functions : see table 7.2

- isdigit(c) returns true if c is a digit character otherwise returns false

Declarations

- Identifiers should be

  - Short enough to be reasonable to type (single word is norm)

  - Standard abbreviations are fine (but only standard abbreviations)

  - Long enough to be understandable

  - When using multiple word identifiers capitalize the first letter of each word

Variable Declarations

- Examples
  - char response;
  - int minelement;
  - float score;
  - float temperature;
  - int i;
  - int n;
  - char c;
  - float x;

Constant Declarations

- Types of constants
  - integer
  - float
  - char
  - Bool
  - string objects
- Associate meaningful terms
  - const float PAYRATE = 10.25;

Assignment Statements

- Form : **variable = expression ;**
- Stores a value or a computed result in a variable
- Example :
  - x = 34 ;
  - x = -y ;
  - x = y * z + 1 ;
  - sum = sum + item ;
- Memory state : before and after the execution of each executable statement

Input/Output Operations

- Form : **cin >> variable ;**
- Reads data from the standard input device *cin* into memory
- cin associated with keyboard
- >> (the input operator) directs input to variable
- used with int, float, char, bool, and string
- #include<iostream>
- Any blank characters preceding the value is ignored during the input operation

- Example
  - int x ;
  - float y ;
  - char z ;
  - cin >> x ; // gets value from cin into variable x
  - cin >> y >> z ;

Output Operations

- Form : **cout $<<$ data-element ;**
- data-element : variable, constant, object or literal
- Displays information on the standard output device *cout*
- cout associated with screen
- $<<$ output operator
- #include$<$iostream$>$
- Prompts : a displayed message that informs the user to enter data
- endl (end-line) : advances the cursor to the next line

- Example
  - int x = 6 ;
  - char y ;
  - cout << "Enter the value of y " << endl;
  - cin >> y ;
  - cout << "The value of x is " << x << endl ;
  - cout << "The value of y is " << y << endl ;

Formatting for numbers with decimal point

Depending on the compiler, the following statements

double x=12.5; cout $<<$ "x = " $<<$ x;

can produce any of the following outputs:

- x = 12.5 (fixed-point notation)
- x = 12.500000 (fixed-point notation)
- x = 1.25e01 (scientific notation)
- x = 1.2500000e01 (scientific notation)

Stream functions can be used to produce the output you want.

For example:

- **cout.setf(ios::fixed); cout.precision(2);** will produce
  - x = 12.50
- while **cout.setf(ios::scientific); cout.precision(4);** will output
  - x = 1.2500e+01

Arithmetic Expressions

Arithmetic Operators

- **+ - *** / used with type int and type float
- **%** modulus or remainder used only with type int
- Example of division :
  - 15 / 2 equals 7 an integer division
  - 15.0 / 2.0 equals 7.5
  - 0 / 15 equals 0
  - 15 / 0 is undefined

- Example of integer modulus :
  - 7 % 2 equals 1
  - 49 % 5 equals 4
  - 15 % 0 is undefined
- m equals (m / n) * n + (m % n)

Mixed-Type Expressions

- An expression involving operands of type int and of type float
- If all operands are of type int then the result is of type int
- If at least one operand is of type float then the result is of type float
- Example
  - 2 + 4 is of type int
  - 5 / 2.0 is of type float
  - 8.0 + 2 is of type float

Mixed-Type Assignment Statements

- Expression evaluated first
- Result stored in the variable on the left side
- A type int and type float expression may be assigned to a type float variable
- Example
  - float a ; a = 10 ;
  - float b ; b = 10 / 7 ;
  - int x ; x = 6.0 + 2.7 ; warning error in compilation

Expressions with Multiple Operators

- Operator precedence rule tells how to evaluate expressions
- Standard precedence order:
  - ( ) : evaluated first, if nested the innermost is evaluated first
  - Unary **+ -** : evaluated second, if there are several operators then evaluate from left to right
  - ***** / **%** : evaluated third, if there are several operators then evaluate from left to right
  - Binary **+** **-** : evaluated fourth, if there are several operators then evaluate from left to right

Precedence of Operators

- Most operators are evaluated (grouped) in a left-to-right order

a / b / c / d is equivalent to (((a/b)/c)/d)

- Assignment operators group in a right-to-left order so the expression

x = y = z = 0.0 is equivalent to (x=(y=(z=0.0)))

Precedence of Operators

Operator	Description
( )	parentheses
! + -	Not, unary plus/minus
*** / %**	Multiply/divide/remainder
+ -	Binary plus/minus
< > <= >=	Less/Greater, Less/Greater or equal
== !=	Equal, Not equal
&& ||	Logical and, Logical or
=	Assign expression

Mixing Types

- Type Promotions
  - promotes (converts) a lower type to a higher type
  - the mixed-type operands will have the same type
  - type int to type float, type float to type double
- Type conversions
  - int to float (number.0)
  - float to int (truncation occurs)

Type Casting

- Avoid mixed-type expressions and assignments by using the casting operators
- Type casting allows you to change a type of a variable or an expression
- Form : **new_type(variable)**

  or **new_type(expression)**
- Example :

```
int n ; float sum, average ;
average = sum /float(n) ;
```

- Values and types are preserved

Interactive Mode, Batch Mode and Data Files

- 2 basic modes of program execution :

Interactive Mode

- The user interacts with a running program, entering data as requested

Batch Mode

- A non interactive mode of execution that requires all program data to be supplied before execution begins

- Prepare a batch data file before executing the program

Input Redirection

- Using operating system commands to associate the standard input device with an input file instead of the keyboard
- Example : miles.cpp rewritten as a batch program
  - The input data are associated with a data file called *mydata*
  - The executable file is called *milesbatch*
  - UNIX command : milesbatch $<$ mydata
  - cin $>>$ miles ; reads the value of miles from the *mydata* file
- Replace each prompt with an **echo print** that follows each input operation

Output Redirection

- Using operating system commands to associate the standard output device with an output file instead of the screen
- Example
  - Redirect the program output from the screen to a file called myoutput
  - UNIX command : milesbatch $>$ myoutput
- Using input and output redirection :
  - milesbatch $<$ mydata $>$ myoutput

Common Programming Errors

- Three kind of programming errors:
  - Syntax Errors
  - Run-Time Errors
  - Logic Errors
- **Debugging** : removing errors from a program

Syntax Errors

- A violation of C++ grammar rules (syntax of the language)
- Errors are displayed during the compilation
- The compilation process fails
- Warning errors : the compilation is done and you can perform the program
- Example of errors :
  - A variable or constant is not declared
  - A statement doesn't end with **;**

Run-Time Errors

- Errors are detected during the program execution
- The program execution is aborted
- Performing an illegal operation
- Example :
  - dividing a number by zero (runtimeError.cpp)

Logic Errors

- Detect these errors by testing the program
- The program executes without problems but the results are incorrect
- These errors are difficult to detect because they usually do not cause runtime errors and do not display error messages

Selection Statements

- Executes one of several alternative statements
  - Conditions (evaluated to true or false)
  - Boolean operators, relational and equality operators
  - if statement (do this only if a condition is true)
  - if-else statement (do either this or that)
  - nested if statement (more than two alternatives)
  - switch statement (more than two alternatives)

Control Structures

- Programs must often anticipate a variety of situations
- Consider an Automated Teller Machine:
  - ATMs must serve valid bank customers
  - They must also reject invalid PINs
  - The code that controls an ATM must permit these different requests
  - Software developers must implement code that anticipates all possible transactions

if Control Statement

- The if is the first statement that alters strict sequential control
- General form : **if (condition) statement ;**
- condition : any logical expression evaluated to nonzero (true) or zero (false)
- if (x $>=$ 0) cout $<<$ x $<<$ " is positive number " ;

  if (c == 'h') cout $<<$ "HELLO" ;

if Control Statements with Two Alternatives

- The condition is evaluated. When true, the statement_i is executed and the statement_j is disregarded. When false, only the statement_j is executed
- General Form

if (condition) statement_i;

else statement_j ;

- if (x >= 0) cout << x << " is positive number " ;

  else cout << x << " is negative number " ;

If Statements with Compound Alternatives

- General form (also known as a block) :

```
{
statement-1 ;
statement-2 ;
...
statement-N ;
}
```

- The compound statement groups together many statements that are treated as one

Writing Compound Statements

```
if (transactionType == 'c')
{   // process check
   cout << "Check for " << transactionAmount << endl;
   balance = balance - transactionAmount; }
else
{   // process deposit
   cout << "Deposit of " << transactionAmount << endl;
  balance = balance + transactionAmount; }
```

Nested if Statements

- Nested if is one selection structure containing another selection structure
- An if...else inside another if...else
- Example

```
if (x > 0) numPos = numPos + 1 ;
else    if (x < 0)
            numNeg = NumNeg + 1;
        else
            numZero = numZero + 1;
```

- Nested if statements can become quite complex. If there are more than three alternatives and indentation is not consistent, it may be difficult to determine the logical structure of the if statement.

```
// Function displayGrade
void displayGrade(int score) {
if (score >= 90)   cout << "Grade is A " << endl;
else if (score >= 80)   cout << "Grade is B " << endl;
      else    if (score >= 70)
                    cout << "Grade is C " << endl;
              else    if (score >= 60)
                        cout << "Grade is D " << endl;
                      else cout << "Grade is F " << endl; }
```

The switch Control Statement

```
switch (selector) {
   case value-1 :
         statement(s)-1
         break ;
   ... // many cases are allowed
    case value-n :
          statement(s)-n
          break ;
    default : default-statement(s) // optional }
```

- When a switch statement is encountered :
  - the selector is evaluated. A selector is a variable or an expression of type int, char or bool
  - the value of selector is compared to each case value until selector == case value
  - all statements after the column **:** are executed until the break statement
  - if no case value matches the selector, the statements following the default are performed
- It is important to include the break statement

```
switch(watts) {
  case 25 : cout << " Life expectancy is 2500 hours. " << endl;
            break;
  case 40: case 60:
      cout << " Life expectancy is 1000 hours. " << endl;
      break;
   case 75: case 100:
       cout << " Life expectancy is 750 hours. " << endl;
       break;
  default : cout << "Invalid Bulb !! " << endl;
} // end switch
```

PayrollFunctions.cpp

```
// File: payrollFunctions.cpp

// Computes and displays gross pay and net pay

// given an hourly rate and number of hours

// worked. Deducts union dues of $15 if gross

// salary exceeds $100; otherwise, deducts no dues.

#include <iostream>

#include "money.h"

using namespace std;

// Functions used ...

void instructUser();
```

```
money computeGross(float, money);

money computeNet(money);

const money MAX_NO_DUES = 100.00;

const money dues = 15.00;

const float MAX_NO_OVERTIME = 40.0;

const float OVERTIME_RATE = 1.5;

int main () { float hours; float rate;

money gross; money net;

instructUser();

// Enter hours and rate.

cout << "Hours worked: ";
```

```
cin >> hours;
cout << "Hourly rate: ";
cin >> rate;
// Compute gross salary.
gross = computeGross(hours, rate);
// Compute net salary.
net = computeNet(gross);
// Print gross and net.
cout << "Gross salary is " << gross << endl;
cout << "Net salary is " << net << endl;
return 0; }
```

```
// Displays user instructions

void instructUser() {

cout << "This program computes gross and net salary." << endl;

cout << "A dues amount of " << dues << " is deducted for" <<
endl;

cout << "an employee who earns more than " <<
MAX_NO_DUES << endl << endl;

cout << "Overtime is paid at the rate of " << OVERTIME_RATE
<< endl;

cout << "times the regular rate for hours worked over " <<
MAX_NO_OVERTIME << endl << endl;

cout << "Enter hours worked and hourly rate" << endl;
```

```
cout << "on separate lines after the prompts. " << endl;

cout << "Press <return> after typing each number." << endl
<< endl;

} // end instructUser
```

```
// FIND THE GROSS PAY

money computeGross (float hours, money rate) {

money gross; money regularPay;

money overtimePay;

if (hours > MAX_NO_OVERTIME) {

  regularPay = MAX_NO_OVERTIME * rate;

  overtimePay = (hours - MAX_NO_OVERTIME) *
OVERTIME_RATE * rate;

  gross = regularPay + overtimePay; }

else gross = hours * rate;

return gross; } // end computeGross
```

```
// Find the net pay

money computeNet(money gross) {

money net;

// Compute net pay.

if (gross > MAX_NO_DUES)

    net = gross - dues;

else     net = gross;

return net; } // end computeNet
```

Program output

This program computes gross and net salary.

A dues amount of $15.00 is deducted for an

employee who earns more than $100.00

Overtime is paid at the rate of 1.5 times the

regular rate on hours worked over 40

Enter hours worked and hourly rate on separate

lines after the prompts.

Press <return> after typing each number.

Hours worked: 50

Hourly rate: 6

Gross salary is $330.00

Net salary is $315.00

Repetition Statements

- Avoid writing the same statements over and over again
- Because many algorithms require many iterations over the same statements :
  - To average 100 numbers, we would need 99 plus statements
  - Or we could use a statement that has the ability to repeat a collection of statements
- Repetition statements : **while**, **for** and **do-while**

Counting Loops and the while Statement

- General form of the while statement :

while (loop-test)

{

iterative part //**statements**

}

- 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

- Sum 100 values the hard way

```
int sum = 0;

cout << "Enter number: ";

cin >> number;

sum = sum + number;

.

cout << "Enter number: ";

cin >> number;

sum = sum + number;

average = sum / 100;
```

- Collections of statements are delimited with $\{$ and $\}$
- counter = 1 ; sum = 0 ;

while (counter $<= 100$)

$\{$ cout $<<$ "Enter number: ";

cin $>>$ number;

sum = sum + number;

counter = counter + 1;

$\}$

average = sum / 100;

Infinite loops

- Infinite loop: a loop that never terminates
- Infinite loops are usually not desirable
- Here is an example of an infinite loop:

```
int counter = 1; int j = 2;

while(counter <= j) {

    j = j + 1; counter = counter + 1 ;

    cout << counter << endl; }

cout << "Do we ever get here?" << endl;
```

- There is no step that brings us to termination

Compound Assignment Operators

- Short hand notation

  totalPay += pay ;

same as

totalPay = totalPay + pay ;

- Operators : **+=, -=, *=, /=, %=**

The for Statement

- 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

- Form :

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

```
for(int counter = 1; counter <=100; counter = counter+1)
{     cout << "Enter number: ";
   cin >> number;
   sum += number; }
```

Other Incrementing Operators

- The unary ++ and −− operators add 1 and subtract 1 from the operand, respectively
- int n = 0;

n++ is equivalent to n = n + 1;

n- -; is equivalent to n = n -1 ;

- It is common to see counter-controlled loops of this form where n is the number of loops

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

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
     } while (option != 'N' || option != 'O' || option != 'S');
   return option; }
```

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

Nested Loops

- Possible to nest loops
- Loops inside other loops
- Start with outer jump to inner loop
- Inner loop continues until complete
- Back to outer loop and start again
- Example : int n ; cin >> n ;

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

Functions

- How to write inputs to a function and how to return a single output ?
- How to write functions that use output parameters and return more than one result ?
- **Call by Value** versus **Call by Reference**
- Modifying Objects

Value and Reference Parameters

- computeSumAve(x, y, sum, mean) ;
- Four parameters : two for input and two for output
- In the function header, & indicates output parameters
- Insert & after the type of the formal parameter
- The void function computeSumAve can return values to the calling function through its output parameters

ACTUAL ARGUMENTS	FORMAL PARAMETERS	
x	num1	Passing by Value
y	num2	Passing by Value
sum	sum	Passing by Reference
mean	average	Passing by Reference

computeSumAve.cpp

```
// File: computeSumAve.cpp

#include <iostream>

using namespace std;

void computeSumAve(float, float, float&, float&);

int main () {

float x, y, sum, mean ;

cout << "Enter 2 numbers: ";

cin >> x >> y;

computeSumAve(x, y, sum, mean);

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

```
cout << " Average is " << mean << endl ;

return 0; }

// COMPUTES THE SUM AND AVERAGE OF NUM1 AND NUM2

// Pre: num1 and num2 are assigned values.

// Post: The sum and average of num1 and num2

// are computed and returned as function outputs.

void computesumave(float num1, float num2,float& sum, float&
average) { sum = num1 + num2 ; average = sum / 2.0; }
```

Call-by-Value and Call-by-Reference

- Call by Value : values of actual arguments are copied to the corresponding formal parameters
- Call by Reference (syntax &) : addresses of actual arguments are copied to the corresponding formal parameters
- If one return value is enough, use value arguments with a return statement. If more than one return is needed, use reference arguments

- Three types of parameters :
  - Input parameter or value parameter : parameter used to receive value from the calling function
  - Output parameter or reference parameter : parameter used to return a result to the calling function
  - Input/Output parameter or **inout** parameter : a parameter that receives value from the calling function and returns a value to it

Call by Value

- Value of actual argument is stored in the called function
- The function execution cannot change the actual argument value
- Actual argument can be an expression, a variable, or a constant
- Formal parameters are used to store data passed to a function

Call by Reference

- Memory address of the actual argument is what is passed to the function. Because it is the address in memory of the actual argument you can modify its value. Data can flow into a function and out of the function
- The function execution can change the actual argument value
- Actual argument must be a variable. So, it should be declared in the calling function
- Formal parameters are used to return output from a function (output parameters) or to change the value of a function argument (inout parameters)
- Insert & in the formal parameter list and in the function prototype

Functions with In/out Parameters

- Examine functions that have only output or inout parameters
- See testGetFrac.cpp :

void getFrac(int&, int&) ; // function to get a fraction

- Data items are entered at the keyboard

testGetFrac.cpp

```
// File: testGetFrac.cpp

// Tests the fractions.

#include <iostream>

using namespace std;

void getFrac(int&, int&);

int main() {

int num, denom;

cout << "Enter a common fraction " << "as 2 integers separated
by a slash: ";

getFrac(num, denom);
```

```
cout << "Fraction read is " << num << " / " << denom <<
endl;
return 0; }
// Reads a fraction.
// Pre: none
// Post: numerator returns fraction numerator,
// denominator returns fraction denominator
void getFrac(int& numerator, int& denominator)
{ char slash;
// Read the fraction
cin >> numerator >> slash >> denominator; }
```

Program Output

```
Enter a fraction as 2 integers separated by a slash :

3 / 4

The Fraction is : 3 / 4
```

Function Syntax & Arguments

- Correspondence between actual and formal arguments is determined by position in their respective argument lists. These lists must be the same size. The names of corresponding actual and formal arguments may be different. Formal arguments and corresponding actual arguments should agree with respect to type

- For reference arguments, an actual argument must be a variable. For value arguments, an actual argument may be a variable, a constant or an expression