Programming

- A digital computer is a useful tool for solving a great variety of **problems.**
- **A** solution to a problem is called an **algorithm; it describes the sequence of steps to** be performed for the problem to be solved.
- To make an algorithm intelligible to a computer, it needs to be expressed in a language understood by a computer (**machine language).**
- **Programs expressed in** the machine language are said to be **executable.**
- A further abstraction of machine language is the **assembly language.**
- **High-level languages** such as C++ provide a much more convenient notation for implementing algorithms.

Compiler

- A program written in a high-level language is translated to assembly language by a translator called a **compiler.**
- **The assembly** code produced by the compiler is then assembled to produce an executable program.

C++ Compiler

- Compiling a C++ program involves a number of steps :
- First, the C++ **preprocessor goes over the program text and carries out the** instructions specified by the preprocessor directives (e.g., #include). The result is a modified program text which no longer contains any directives.
- Then, the C++ **compiler translates the program code. The compiler may be a** true C++ compiler which generates native (assembly or

machine) code, or just a translator which translates the code into C. In the latter case, the resulting C code is then passed through a C compiler to produce native object code. In either case, the outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program.

- Finally, the **linker completes the object code by linking it with the object code** of any library modules that the program may have referred to. The final result is an executable file.

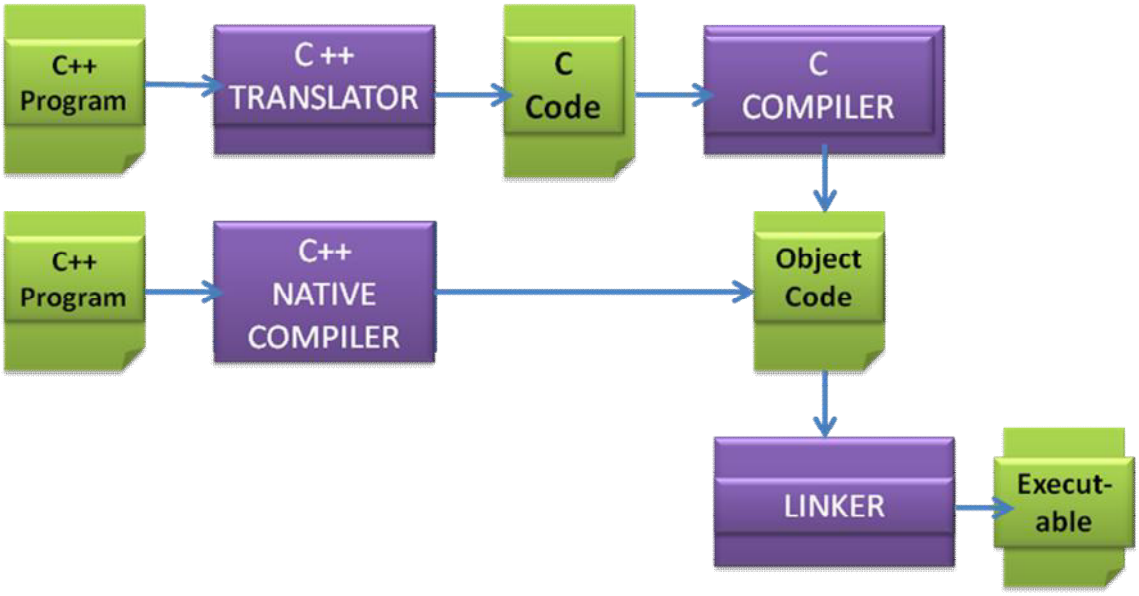


Figure1:C++ compilation

Variables

- ⦿ Variable is a symbolic name for a memory location in which data can be stored and subsequently recalled.
- ⦿ All variables have two important attributes:
- ❖ A **type which is established when the variable is defined (e.g., integer, real,** character). Once defined, the type of a C++ variable cannot be changed.
- ❖ A **value which can be changed by assigning a new value to the variable. The** kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).

Simple variables

Example 1:

```
#include <iostream.h>
int main (void)
{
        int workDays;
        float workHours, payRate, weeklyPay;
        workDays = 5;
        workHours = 7.5;
        payRate = 38.55;
        weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = ";
        cout << weeklyPay;
        cout << '\n';
}
```

Initialization

Example 2:

```
#include <iostream.h>
int main (void)
{
        int workDays = 5;
        float workHours = 7.5;
        float payRate = 38.55;
        float weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = ";
        cout << weeklyPay;
        cout << '\n';
}
```

Comments

- A comment is a piece of descriptive text which explains some aspect of a program.
- Program comments are totally ignored by the compiler and are only intended for human readers.
- C++ provides two types of comment delimiters:
- Anything after // (until the end of the line on which it appears) is considered a comment.
- Anything enclosed by the pair /* and */ is considered a comment.

Comments forms

Example 3:

```
#include <iostream.h>

/* This program calculates the weekly gross pay for a worker,
based on the total number of hours worked and the hourly pay rate. */

int main (void)
}
    int workDays = 5;          // Number of work days per week
    float workHours = 7.5;     // Number of work hours per day
    float payRate = 33.50;   // Hourly pay rate
    float weeklyPay;         // Gross weekly pay
    weeklyPay = workDays * workHours * payRate;
    cout << "Weekly Pay = " << weeklyPay << '\n';
{
```

Simple Type Conversion

- A value in any of the built-in types we have see so far can be converted (*type-cast)* to any of the other types.
- For example:

  (int) 3.14 // converts 3.14 to an int to give 3

  (long) 3.14 // converts 3.14 to a long to give 3L

  (double) 2 // converts 2 to a double to give 2.0

  (char) 122 // converts 122 to a char whose code is 122

  (unsigned short) 3.14 // gives 3 as an unsigned short
- In some cases, C++ also performs **implicit type conversion. This happens** when values of different types are mixed in an expression. For example:

  double d = 1; // d receives 1.0

```
int i = 10.5; // i receives 10
i = i + d; // means: i = int(double(i) + d)
```

Statements

- Simple statement
- Compound Statements
- The if Statement
- The switch Statement
- The while Statement
- The do Statement
- The for Statement
- The continue Statement
- The break Statement
- The goto Statement
- The return Statement

Simple Statements

- A simple statement is a computation terminated by a semicolon. Variable definitions and semicolon terminated expressions are examples:

```
int i; // declaration statement
++i; // this has a side-effect
double d = 10.5; // declaration statement
d + 5; // useless statement!
```

The last example represents a useless statement, because it has no side-effect (d is added to 5 and the result is just discarded).

- The simplest statement is the null statement which consists of just a semicolon:

```
;    // null statement
```

The if Statement

- t is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The if statement provides a way of expressing this, the general form of which is:

if (*expression)*

Statement1

else

Statement2

- First *expression is evaluated. If the outcome is nonzero then statement is* executed. Otherwise, nothing happens.
- Example

```
if (count != 0)
average = sum / count;
```

The switch statement

- The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression. The general form of the switch statement is:

switch (*expression) {*

case constant$_1$:

statements;

...

case constant$_n$:

statements;

default:

statements;

}

```
switch (operator) {
      case '+': result = operand1 + operand2;
      break;
      case '-': result = operand1 - operand2;
      break;
      case '*': result = operand1 * operand2;
      break;
      case '/': result = operand1 / operand2;
      break;
      default: cout << "unknown operator: " << ch <<
'\n';
      Break;
      }
```

The while Statement

The general form of the while statement is:

while (*expression)*

statement;

- First *expression (called the* ***loop condition) is evaluated. If the outcome is nonzero***
- then *statement (called the* ***loop body) is executed and the whole process is*** repeated.
- Otherwise, the loop is terminated.

The do Statement

The general form of the do statement is:

do

statement;

while (expression);

- First statement is executed and then expression is evaluated.
- If the outcome of the latter is nonzero
- Then the whole process is repeated. Otherwise, the loop is terminated.

The for Statement

The general form of the for statement is:

for (*expression1; expression2; expression3)*

statement;

- First *expression1 is evaluated. Each time round the loop, expression2 is* evaluated.
- If the outcome is nonzero then statement is executed and *expression3 is* evaluated.
- Otherwise, the loop is terminated.

The continue Statement

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It applies to the loop immediately enclosing the continue statement.
- It is an error to use the continue statement outside a loop.

```
do {
cin  >> num;
if  (num < 0) continue;
// process num here...
} while (num != 0);
```

The break Statement

- A break statement may appear inside a loop (while, do, or for) or a switch statement.
- It causes a jump out of these constructs, and hence terminates them.
- Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it.
- It is an error to use the break statement outside a loop or a switch.

- For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts:

```
for (i = 0; i < attempts; ++i)
{
cout << "Please enter your password: ";
cin >> password;
if (Verify(password)) // check password for correctness
break; // drop out of the loop
cout << "Incorrect!\n";
}
```

The goto Statement

The goto statement has the general form:

goto *label;*

- where *label is an identifier which marks the jump destination of goto. The label* should be followed by a colon and appear before a statement within the same function as the goto statement itself.
- For example

```
for (i = 0; i < attempts; ++i)
{
cout << "Please enter your password: ";
cin >> password;
if  (Verify(password)) // check password for correctness
goto out; // drop out of the loop
cout << "Incorrect!\n";
{
out:
//etc...
```

The return Statement

The return statement enables a function to return a value to its caller. It has the general form:

return *expression;*

- where *expression denotes the value returned by the function. The type of this value* should match the return type of the function.
- For a function whose return type isvoid, *expression should be empty:* return;

Overview for functions

A function is a set of statements designed to accomplish a particular task. The advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program. C++ has added many new features to functions to make them more reliable and flexible.

General Format of a Function Definition:

Functions can be define before the definition of the main() function, or they can be declared before it and define after it.

Declaring a function means listing its return type, name, and arguments. This line is called the function prototype. A function prototype tells the compiler the type of data returned by the function. It is usually defined after the preprocessing statements at the beginning of the program.

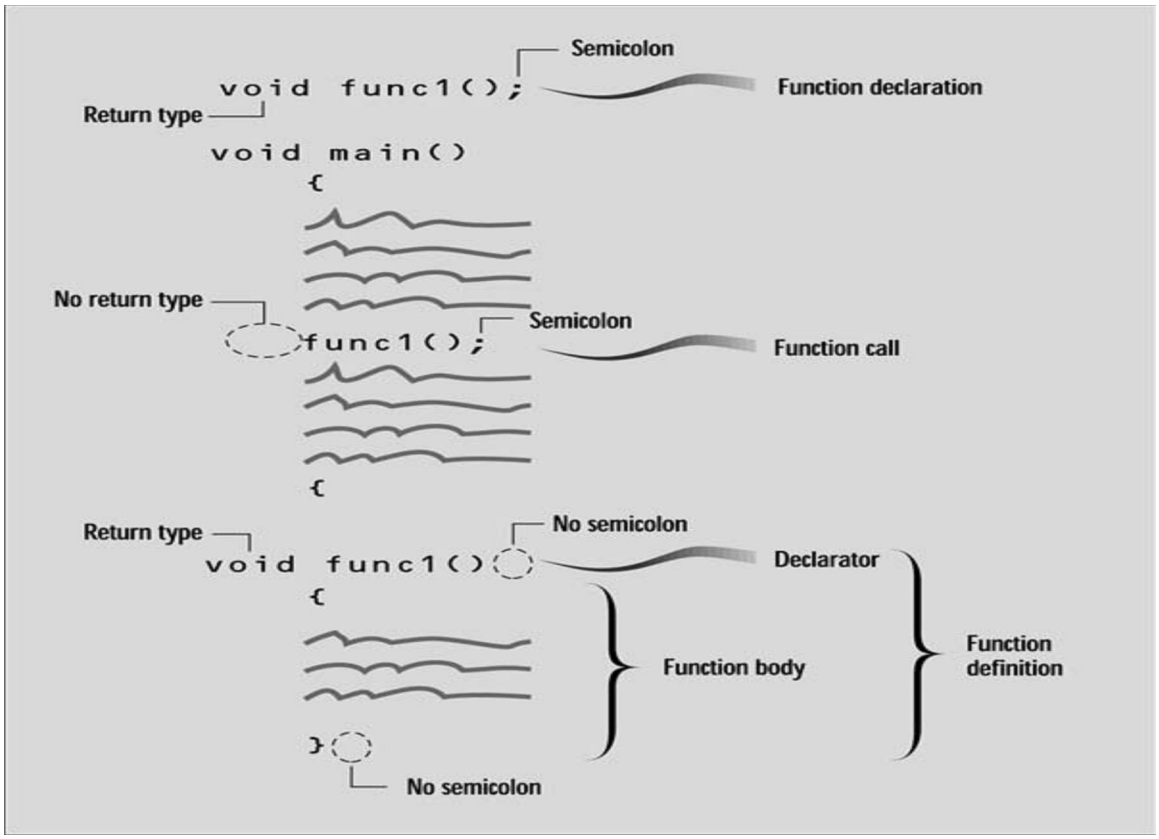


Figure 1: *Function syntax.*:

Example 1:

Design a function to calculate the squared value of an integer passed from the main function. Use this function in a program to calculate the squares of integers from 1..10.

```
#include<iostream.h>
int square(int);
void main( )
{
        int x;
        for (x=1;x<=10;x++)
                cout << square(x)<<endl;
}
int square(int y)
{
        return (y*y);
}
```

Output

```
1
4
9
16
25
36
49
64
81
100
```

Local and Global Variables:

All variables define inside the block of a function are called local variables. These variables belong to the function exclusively. That is no other function has access to it.

Example 2:

Write a function to find the largest integer among three integers entered by the user in the main function.

```
#include <iostream.h>
int max(int,int,int);
void main( )
{
        int largest,x1,x2,x3;
        cout<<"Enter 3 integer numbers:";
        cin>>x1>>x2>>x3;
        largest=max(x1,x2,x3);
        cout<<largest;
}
```

```
int max(int y1, int y2, int y3)
{
         int big;
         big=y1;
         if (y2>big) big=y2;
         if (y3>big) big=y3;
         return (big);
}
```

When variables declared outside any function, it is called global variables. This type of variables can be accessed by all the functions in the program as well as the main function.

Inline Function:

C++ suppliers' programmers with the inline keyword, which can speed up programs by making very short functions execute more efficiently. Normally a function resides in a separate part of memory, and is referred to by a

running program in which it is called. Inline functions save the step of retrieving the function during execution time, at the cost of a larger compiled program.

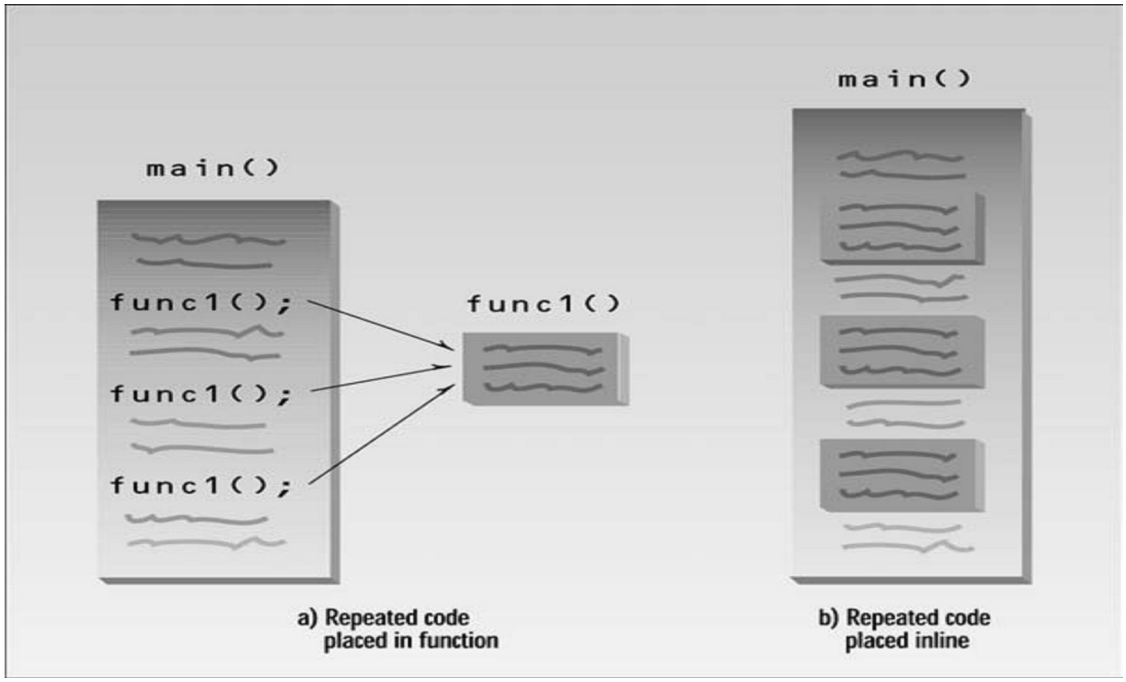


Figure: ***Functions versus inline code***

Example 3:

The following MAX is an inline function that returns the maximum of two values.

```
#include <iostream.h>
inline int MAX(int a, int b)
{
        return (a>b)?a:b;
}
void main( )
{
        int x1,x2;
        cout<<"Enter 2 integer numbers:";
        cin>>x1>>x2;
        cout<<MAX(x1,x2);
}
```

Function Overloading:

Overloading refers to the use the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks.

We can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call.

Example 4:

The following program illustrates function overloading.

```
#include <iostream.h>

int       volume(int);
double volume(double,int);
long     volume(long,int,int);

void main( )
{
        cout<<volume(10)<<"\n";
        cout<<volume(2.5,8)<<"\n";
        cout<<volume(100L,75,15);
}

int volume(int s)
{
        return(s*s*s);                  // cube
}

double volume(double r,int h)       // cylinder
{
        return(3.14519*r*r*h);
}

long volume(long l,int b,int h)       // rectangular box
{
        return(l*b*h);
}
```

Passing Parameters:

There are two main methods for passing parameters to a program:

1- Passing by Value:

When parameters are passed by value, a copy of the parameters value is taken from the calling function and passed to the called function. The original variables inside the calling function, regardless of changes made by the function to it are parameters will not change. All the pervious examples used this method.

2- Passing by Reference:

When parameters are passed by reference their addresses are copied to the corresponding arguments in the called function, instead of copying their values. Thus pointers are usually used in function arguments list to receive passed references.

This method is more efficient and provides higher execution speed than the call by value method, but call by value is more direct and easy to use.

Example 5:

The following program illustrates passing parameter by reference.

```
#include <iostream.h>
void swap(int *a,int *b);
void main( )
{
        int x=10;
        int y=15;
        cout<<"x before swapping is:"<<x<<"\n";
        cout<<"y before swapping is:"<<y<<"\n";
```

```
        swap(&x,&y);
        cout<<"x after swapping is:"<<x<<"\n";
        cout<<"y after swapping is:"<<y<<"\n";
}

void swap(int *a,int *b)
{
        int c;
        c=*a;
        *a=*b;
        *b=c;
}
```

Example 6:

 Write a C++ program using functions to print the contents of an integer array of 10 elements.

```
#include <iostream.h>

int const size=10;

void pary(int y[]);

void main( )
{
        int s[size]={2,4,6,8,10,12,14,16,18,20};
        pary(s);
}

void pary(int x[])
{
        int i;
        for (i=0;i<size;i++)
            cout<<x[i]<<endl;
}
```

Example 7:

Write a function that counts uppercase letter in a string entered by the user in the main program. Assume the maximum string length is 100.

```
#include<iostream.h>
#include<string.h>

int const size=100;

int upperCount(char[]);

void main( )
{
        char str[size];
        cout<<"Enter Your String: ";
        cin.getline(str,size,'\n');
        cout<<upperCount(str);
}

int upperCount(char x[])
{
        int count=0;
        for (int i=0;i<strlen(x);i++)
           if (x[i]>='A'&&x[i]<='Z')     count++;
        return (count);
}
```

Example 8:

Write a function that counts the number of words in a string entered by the user in the main program. Assume the maximum string length is 100.
Hint: *the number of words in a sentence can be found by counting the number of spaces.*

```
#include<iostream.h>
#include<string.h>

int const size=100;

int wordCount(char[]);

void main( )
{
        char str[size];
        cout<<"Enter a Sentenc: ";
        cin.getline(str,size,'\n');
        cout<<wordCount(str);
}

int wordCount(char x[])
{
        int count=1;
```

```
        for (int i=0;i<strlen(x);i++)
                if (x[i]==' ')    count++;
        return (count);
}
```

Example 9:

 Design a function that takes the third element of an integer array defined in the main program. The function must multiply the element by 2 and return the result to the main program.

```
#include<iostream.h>
int elementedit(int);

void main( )
{
        int a[4]={2,5,3,7};
        int k;
        k=elementedit(a[2]);
        cout<<"k after change is: "<<k;
}

int elementedit(int m)
{
        m=m*2;
        return(m);
}
```

Arrays

An array consists of a set of objects (called its elements), all of which are of the same type and are arranged contiguously in memory. In general, only the array itself has a symbolic name, not its elements. Each element is identified by an index which denotes the position of the element in the array. The number of elements in an array is called its dimension. The dimension of an array is fixed and predetermined; it cannot be changed during program execution.

Arrays are suitable for representing composite data which consist of many similar, individual items. Examples include: a list of names, a table of world cities and their current temperatures, or the monthly transactions for a bank account.

An array variable is defined by specifying its dimension and the type of its elements. For example, an array representing 10 height measurements (each being an integer quantity) may be defined as:

```
int heights[10];
```

The individual elements of the array are accessed by indexing the array. The first array element always has the index 0. Therefore, heights[0] and heights[9] denote, respectively, the first and last element of heights. Each of heights elements can be treated as an integer variable. So, for example, to set the third element to 177, we may write:

```
heights[2] = 177;
```

Attempting to access a nonexistent array element (e.g., heights[-1] or heights[10]) leads to a serious runtime error (called ‘index out of bounds’ error).

Processing of an array usually involves a loop which goes through the array element by element. Listing 5.13 illustrates this using a function which takes an array of integers and returns the average of its elements.

The following example illustrates using a function which takes an array of integers and returns the average of its elements.

const int size = 3;
double Average (int nums[size])
{
double average = 0;
for (i = 0; i < size; ++i)
average += nums[i];
return average/size;
}

Array Initialization

An array may have an initializer. Braces are used to specify a list of comma-separated initial values for array elements. For example,

```
int nums[3] = {5, 10, 15};
```

initializes the three elements of nums to 5, 10, and 15, respectively. When the number of values in the initializer is less than the number of elements, the remaining elements are initialized to zero:

```
int nums[3] = {5, 10};
```

When a complete initializer is used, the array dimension becomes redundant, because the number of elements is implicit in the initializer. The first definition of nums can therefore be equivalently written as:

```
int nums[] = {5, 10, 15}; // no dimension needed
```

Strings

A C++ string is simply an array of characters. For example,

```
char str[ ] = "HELLO";
```

defines str to be an array of six characters: five letters and a null character. The terminating null character is inserted by the compiler. By contrast,

```
char str[ ] = {'H', 'E', 'L', 'L', 'O'};
```

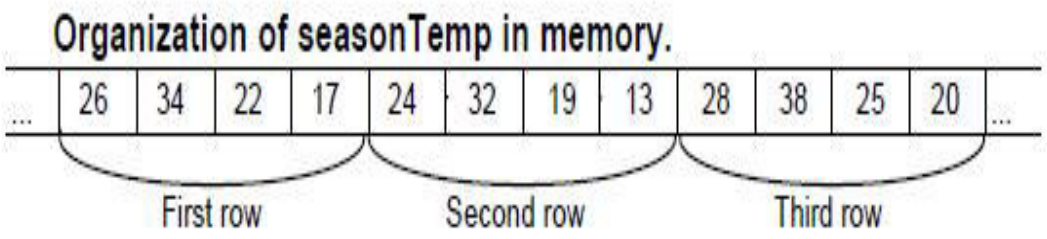
defines str to be an array of five characters.

Multidimensional Arrays

An array may have more than one dimension (i.e., two, three, or higher). The organization of the array in memory is still the same (a contiguous sequence of elements), but the programmer's perceived organization of the elements is different, for Example

```
int seasonTemp[3][4];
```

The organization of this array in memory is as 12 consecutive integer elements. The programmer, however, can imagine it as three rows of four integer entries each.



Multidimensional Arrays Initialization

The array may be initialized using a nested initializer:

```
int   seasonTemp[3][4] = {
{26, 34, 22, 17},
{24, 32, 19, 13},
{28, 38, 25, 10}
};
```

Because this is mapped to a one-dimensional array of 12 elements in memory, it is equivalent to:

int seasonTemp[3][4] = {26, 34, 22, 17, 24, 32, 19, 13, 28, 38, 25, 10};

Multidimensional Arrays Processing

Processing a multidimensional array is similar to a one-dimensional array, but uses nested loops instead of a single loop.:

Example 1

```
const int rows = 3;
const int columns = 4;
int seasonTemp[rows][columns] = {
{26, 34, 22, 17},
{24, 32, 19, 13},
{28, 38, 25, 10}
};
 int HighestTemp (int temp[rows][columns])
{  int highest = 0;
for (i = 0; i < rows; ++i)
for (j = 0; j < columns; ++j)
```

```
if (temp[i][j] > highest)
highest = temp[i][j];
return highest;
}
```

Pointers and References

- A pointer is simply the address of a variable in memory. Generally, variables can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to a variable via a pointer to it, is called dereferencing the pointer. Pointer variables are defined to point to variables of a specific type so that when the pointer is dereferencing, a typed variable is obtained.
- Pointers are useful for creating dynamic variables during program execution.
- Unlike normal (global and local) variables which are allocated storage on the runtime stack, a dynamic variable is allocated memory from a different storage area called the heap.
- Dynamic variables do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.
- A pointer variable is defined to 'point to' data of a specific type. For example:

  int *ptr1; // pointer to an int

char *ptr2; // pointer to a char

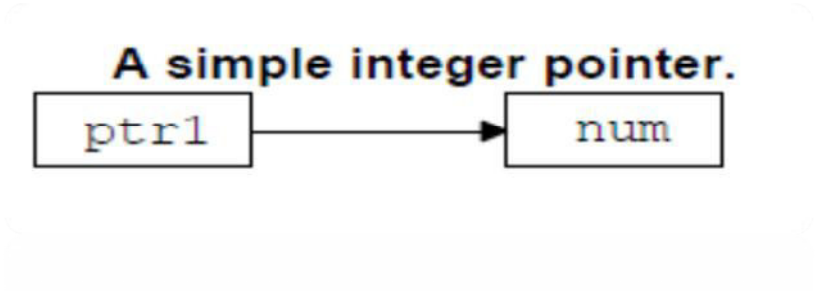
- The value of a pointer variable is the address to which it points. For example, given the definitions

int num;

- we can write:

ptr1 = #

- The symbol & is the address operator; it takes a variable as argument and returns the memory address of that variable. The effect of the above assignment is that the address of num is assigned to ptr1. Therefore, we say that ptr1 points to num.



Pointer Values

- Given that ptr1 points to num, the expression

***ptr1**

- Dereferences ptr1 to get to what it points to, and is therefore equivalent to num. The symbol * is the dereference operator; it takes a pointer as argument and returns the contents of the location to which it points.

- In general, the type of a pointer must match the type of the data it is set to point to.
- Regardless of its type, a pointer may be assigned the value 0 (called the null pointer). The null pointer is used for initializing pointers, and for marking the end of pointer-based data structures (e.g., linked lists).

Pointer Arithmetic

- In C++ one can add an integer quantity to or subtract an integer quantity from a pointer. This is frequently used by programmers and is called pointer arithmetic. Pointer arithmetic is not the same as integer arithmetic, because the outcome depends on the size of the object pointed to. For example, suppose that an int is represented by 4 bytes. Now, given

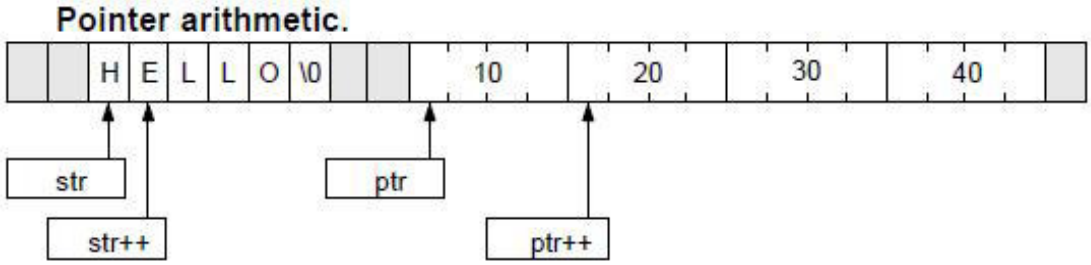
```
char   *str = "HELLO";

int   nums[] = {10, 20, 30, 40};

int   *ptr = &nums[0]; // pointer to first element
```

str++ advances str by one char (i.e., one byte) so that it points to the second character of "HELLO", whereas ptr++ advances ptr by one int (i.e., four bytes) of that it points to the second element of nums.

Pointer Arithmetic – cont.



It follows, therefore, that the elements of "HELLO" can be referred to as *str, *(str + 1), *(str + 2), etc. Similarly, the elements of nums can be referred to as *ptr, *(ptr + 1), *(ptr + 2), and *(ptr + 3).

- Pointer arithmetic is very handy when processing the elements of an array.

Example 2 The following example shows a string copying function similar to strcpy.

```
void CopyString (char *dest, char *src)

{

while (*dest++ = *src++);

}
```

Example 3:- **The following example shows how to pass a string argument and return a string to the main program:**

```
#include <string.h>
#include <iostream>
using namespace std;
char* stf(char *str3);
int main()
{
char *str1 = "ABC";
const int MAX = 10; //size of str2 buffer
char str2[MAX]; //empty string
char *st;
strcpy(str2, str1); //copy str1 to str2
cout << str2 << endl; //display str2
st=stf(str1);
cout << str1 << endl; //display str1
cout << st << endl; //display str1
return 0;
}
char* stf(char *str3)
{
    char* pointer;
    char *str1 ="XYZ";
    str3=str1;
    pointer=&str3[0];
    cout << str3 << endl;
    return pointer;
}
```

Output:

```
ABC
XYZ
ABC
XYZ
```

References

- A reference provides an alternative symbolic name (alias) for an object. Accessing an object through a reference is exactly the same as accessing it through its original name. References offer the power of pointers and the convenience of direct access to objects. They are used to support the call-by-reference style of function parameters, especially when large objects are being passed to functions.

Example 4:-

```
#include "stdafx.h"
#include <iostream.h>


void main()
{
int z=1;
int &x = z;
++x;
int y = x + z;
cout <<y;
    return ;
}
```

Example 5:- The following example shows the difference of passing argument by value, pointer and reference

```
#include<iostream.h>
void Swap1 (int x, int y);
void Swap2 (int *x, int *y);
void Swap3 (int &x, int &y);
int main (void)
{ int i = 10, j = 20;
Swap1(i, j);
cout << "return from swap1=";
 cout << i << ", " << j << '\n';
Swap2(&i, &j);
cout<< "return from swap2=  ";
cout << i << ", " << j << '\n';
Swap3(i, j);
cout<<"return from swap3=  ";
cout << i << ", " << j << '\n';
     return 0; }
```

```
// pass-by-value (objects)
void Swap1 (int x, int y)
{ int temp = x;
x = y;
y = temp; }

// pass-by-value (pointers)
void Swap2 (int *x, int *y)
{ int temp = *x;
*x = *y;
*y = temp; }

// pass-by-reference
void Swap3 (int &x, int &y)
{ int temp = x;
x = y;
y = temp; }
```

Typedefs

Typedef is a syntactic facility for introducing symbolic names for data types. Just as a reference defines an alias for an object, a typedef defines an alias for a type. Its main use is to simplify otherwise complicated type declarations as an aid to improved readability. Here are a few examples:

```
typedef   char *String;

Typedef   char Name[12];

typedef   unsigned int uint;
```

The effect of these definitions is that String becomes an alias for char*, Name becomes an alias for an array of 12 chars, and uint becomes an alias for unsigned int. Therefore:

```
String   str; // is the same as: char *str;
```

```
Name   name; // is the same as: char name[12];
uint   n; // is the same as: unsigned int n;
```

Structures

A structure is a collection of simple variables. The variables in a structure can be of different types: Some can be int, some can be float, and so on. (This is unlike the array, which we'll meet later, in which all the variables must be the same type.) The data items in a structure are called the *members* of the structure.

However, for C++ programmers, structures are one of the two important building blocks in the understanding of objects and classes. In fact, the syntax of a structure is almost identical to that of a class. A structure (as typically used) is a collection of data, while a class is a collection of both data and functions. So by learning about structures we'll be paving the way for an understanding of classes and objects. Structures in C++ (and C) serve a similar purpose to *records* in some other languages such as Pascal.

A Simple Structure

Let's start off with a structure that contains three variables: two integers and a floating-point number. This structure represents an item in a widget company's parts inventory. The structure is a kind of blueprint specifying what information is necessary for a single part. The company makes several kinds of widgets, so the widget model number is the first member of the structure. The number of the part itself is the next member, and the final member is the part's cost.

The program PARTS defines the structure part, defines a structure variable of that type called part1, assigns values to its members, and then displays these values.

Example1 :

```
#include <iostream>

////////////////////////////////////////////////////////////////
struct part                     //declare a structure
   {
   int modelnumber;             //ID number of widget
   int partnumber;              //ID number of widget part
   float cost;                  //cost of part
   };
////////////////////////////////////////////////////////////////
int main()
   {
   part part1;                  //define a structure variable

   part1.modelnumber = 6244;    //give values to structure members
   part1.partnumber = 373;
   part1.cost = 217.55F;
                                //display structure members
   cout << "Model "     << part1.modelnumber;
   cout << ", part "    << part1.partnumber;
   cout << ", costs $" << part1.cost << endl;
   return 0;
   }
```

The program's output looks like this:

Model 6244, part 373, costs $217.55

The PARTS program has three main aspects: defining the structure, defining a structure variable, and accessing the members of the structure. Let's look at each of these.

Defining the Structure

The structure definition tells how the structure is organized: It specifies what members the structure will have.

Here it is:

struct part

{

int modelnumber;

int partnumber;

```
float  cost;
};
```

Syntax of the Structure Definition

The keyword struct introduces the structure definition. Next comes the *structure name* or *tag*, which is part. The declarations of the structure members—modelnumber, partnumber, and cost—are enclosed in braces. A semicolon follows the closing brace, terminating the entire structure. Note that this use of the semicolon for structures is unlike the usage for a block of code. As we've seen, blocks of code, which are used in loops, decisions, and functions, are also delimited by braces. However, they don't use a semicolon following the final brace. Figure1 shows the syntax of the structure declaration.

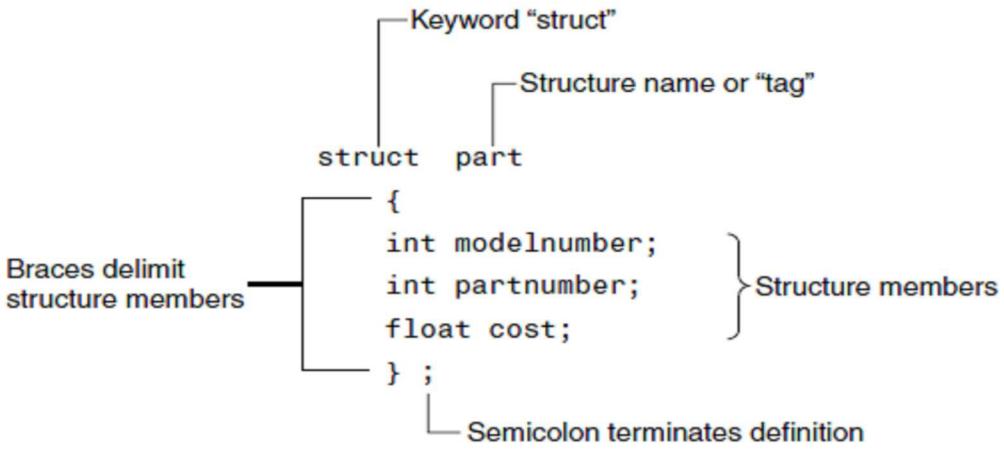


Figure 1: Syntax of the structure definition.

Defining a Structure Variable

The first statement in main() part part1; defines a variable, called part1, of type structure part. This definition reserves space in memory for part1. How much space? Enough to hold all the members of part1 namely modelnumber,

partnumber, and cost. In this case there will be 4 bytes for each of the two ints (assuming a 32-bit system), and 4 bytes for the float. Figure 2 shows how part1 looks in memory. (The figure shows 2-byte integers.) This will become more clear as we go along, but notice that the format for defining a structure variable is the same as that for defining a basic built-in data type such as int: part part1; int var1; This similarity is not accidental. One of the aims of C++ is to make the syntax and the operation of user-defined data types as similar as possible to that of built-in data types.

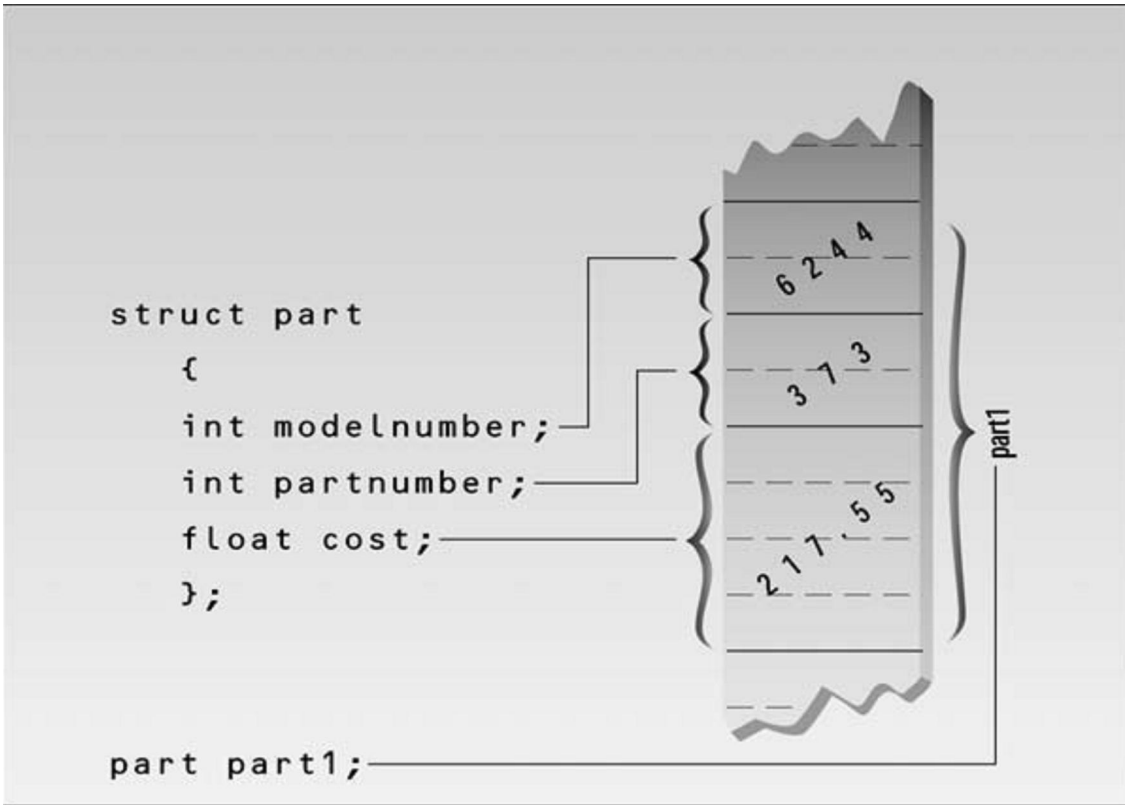


Figure 2: Structure members in memory.

Initializing Structure Members

The next example shows how structure members can be initialized when the structure variable is defined. It also demonstrates that you can have more than one variable of a given structure type.

Example 2 :

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct part                     //specify a structure
   {
   int modelnumber;             //ID number of widget
   int partnumber;              //ID number of widget part
   float cost;                  //cost of part
   };
////////////////////////////////////////////////////////////////
int main()
   {                            //initialize variable
   part part1 = { 6244, 373, 217.55F };
   part part2;                  //define variable
                                //display first variable
   cout << "Model "    << part1.modelnumber;
   cout << ", part "   << part1.partnumber;
   cout << ", costs $" << part1.cost << endl;

   part2 = part1;               //assign first variable to second
                                //display second variable
   cout << "Model "    << part2.modelnumber;
   cout << ", part "   << part2.partnumber;
   cout << ", costs $" << part2.cost << endl;
   return 0;
   }
```

This program defines two variables of type part: part1 and part2. It initializes part1, prints out the values of its members, assigns part1 to part2, and prints out its members.

Here's the output:

Model 6244, part 373, costs $217.55

Model 6244, part 373, costs $217.55

Structures as Arguments

The following example features a function that uses an argument of type structure named Distance:

Example 3

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
int feet;
float inches;
};
Void disp( Distance ); //declaration
int main()
{
Distance d1, d2; //define two lengths
cout << "Enter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
cout << "\nd1 = ";
disp(d1); //display length 1
cout << "\nd2 = ";
disp(d2); //display length 2
cout << endl;
return 0;
}
Void disp( Distance dd )      //parameter dd of type Distance
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

```
Enter feet: 6
Enter inches: 4
Enter feet: 5
Enter inches: 4.25
```

```
d1 = 6'-4"
d2 = 5'-4.25"
```

Returning Structure Variables

Example 4

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
int feet;
float inches;
};
Distance add (Distance, Distance); //declarations
Void disp(Distance);
int main()
{
Distance d1, d2, d3; //define three lengths
cout << "\nEnter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
d3 = add (d1, d2); //d3 is sum of d1 and d2
cout << endl;
disp(d1); cout << " + "; //display all lengths
disp(d2); cout << " = ";
disp(d3); cout << endl;
return 0;
}
Distance add ( Distance dd1, Distance dd2 )
{
Distance dd3; //define a new structure for sum
```

```
dd3.inches = dd1.inches + dd2.inches; //add the inches
dd3.feet = 0; //(for possible carry)
if(dd3.inches >= 12.0) //if inches >= 12.0,
{ //then decrease inches
dd3.inches -= 12.0; //by 12.0 and
dd3.feet++; //increase feet
} //by 1
dd3.feet += dd1.feet + dd2.feet; //add the feet
return dd3; //return structure
}
Void disp( Distance dd )
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

```
Enter feet: 4
Enter inches: 5.5
Enter feet: 5
Enter inches: 6.5
4’ 5.5” + 5’ 6.5”= 10’
```

Overview of OOP:

When working with computers, it is very important to be fashionable!

In the **1960s**, the new fashion was what was called *high-level languages* (H.L.L.) such as ***FORTRAN*** and ***COBOL***, in which the programmer did not have to understand the ***machine instructions***.

In the **1970s**, people realized that there were better ways to program than with a jumble of GOTO statements, and the *structured programming languages* such as ***PASCAL*** were invented.

In the **1980s**, much time was invested in trying to get good results out of *fourth-generation languages* (4GLs), in which complicated programming structures could be coded in a few words. There were also schemes such as Analyst Workbenches, which made systems analysts into highly paid and overqualified programmers.

Bjarne Stroustrup at Bell Labs developed C++ during 198-

1985. The term C++ was first used in 1983. Prior to 1983, Stroustrup added features to C programming language and formed what he called “C with Classes”. In addition to the efficiency and portability of C, C++ provides number of new features. C++ programming language is basically an extension of C programming language.

The fashion of the **1990s** is most definitely *object-oriented programming*.

Read any book on object-oriented programming, and the first things you will read about are three importance OOP features:

• Encapsulation *and Data Hiding*.

• Inheritance *and Reuse*.

• Polymorphism.

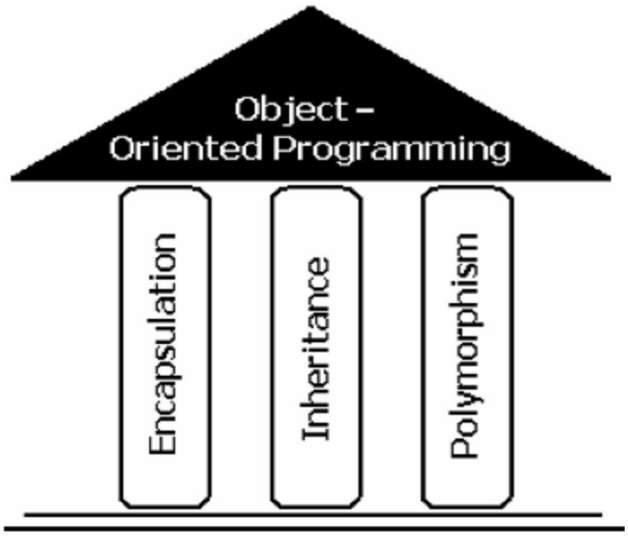


Figure 1: The three pillars of OOP.

Encapsulation and Data Hiding:

When an *engineer* needs to add a resistor to the device, he is create it, he doesn't build a new one from scratch. He walks over to a bin of resistors, examines the colored bands that indicate the properties, and picks the one he needs. The resistor is a "black box" as far as the engineer is concerned, he doesn't much care how its work as long as it conforms to his specifications, he doesn't need to look inside the box to use it in his design. When the engineer uses the resistor, he need not know anything about the internal state of the resistor. All the properties of the resistor are encapsulated in the resistor object; they are not spread out through the circuitry. It is not necessary to understand how the resistor works in order to use it effectively. Its data is hidden inside the resistor's casing.

Just as you can use a refrigerator without knowing how the compressor works, you can use a well-designed object without knowing about its internal data members.

The property of being a self-contained unit is called ***encapsulation***. With encapsulation, we can accomplish data hiding. ***Data hiding*** is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes.

Once created, class acts as a fully encapsulated entity, it is used as a whole unit. The actual inner workings of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it.

Inheritance and Reuse:

When the engineers at Acme Motors want to build a new car, they have two choices: They can start from scratch, or they can modify an existing model.

Perhaps their ***Star model*** is nearly perfect, but the engineers don't like to add a turbocharger and a six-speed transmission. The chief engineer would prefer not to start from the ground up, but rather to say, "Let's build another Star, but let's add these additional capabilities. We'll call the new model a ***Quasar model***". A Quasar model is a kind of Star model, but one with new features. C++ supports the idea of ***reuse*** through ***inheritance***. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The Quasar model is derived from the Star model and thus inherits all its qualities, but can add to them as needed.

Polymorphism:

The new Quasar model might respond differently than a Star model does when you press down on the accelerator. The Quasar model might engage fuel injection and a turbocharger, while the Star model would simply let gasoline into its carburetor. A user, however, does not have to know about these differences. He can just "floor it," and the right thing will happen, depending on which car he's driving. C++ supports the idea that different objects do "the right thing" through what is called function ***polymorphism*** and class polymorphism.

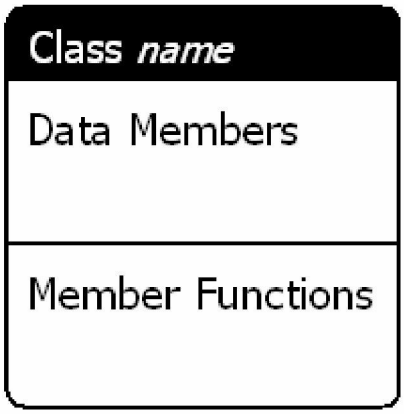
Poly means many, and morph means form. Polymorphism refers to the same name taking *many forms*.

Class Definition:

Class is a keyword, whose functionality is similar to that of the ***struct*** keyword, but with the possibility of including functions as members, instead of only data.

Classes are collections of variables and functions that operate on those variables. The variables in a class definition are called data members, and the functions are called member functions.

Note: Class is a specification for number of objects.



A **class definition** consists of two parts: header and body. The class **header** specifies the class **name** and its **base classes**. The class **body** defines the class **members**. Two types of members are supported:

- **Data members** have the syntax of variable definitions and specify the representation of class objects.
- **Member functions** have the syntax of function prototypes and specify the class operations, also called the class **interface**.

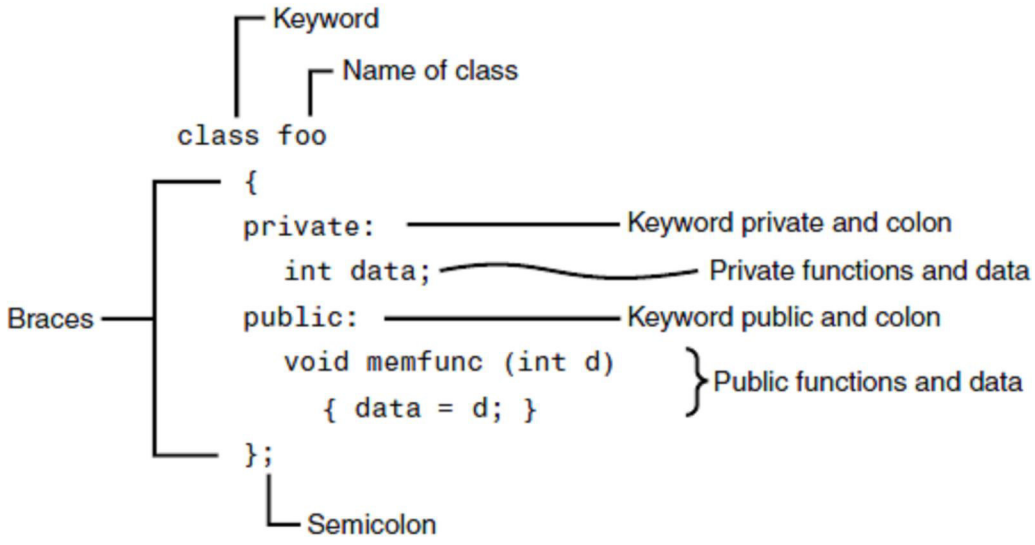


Figure2: Syntax of a class definition

We will **encapsulate** the data member which found in an object with member functions, and as you extend a class you will **"overload"** and **"override"** the functions of the *base class*.

When the permanently associate all related functions to the data of the class. This process is known as **encapsulation**.

Classes can be used to produce more specialized versions through a concept called **Inheritance**.

We can inherit features of the "**base class**", when we create a new "**derived class**". That is, we will create general object classes and then make more specific classes from them, deriving the particular from the general.

With such concept; new classes can be built on top of existing ones and extending their base classes; therefore allowing software designers to ***reuse code*** easily, which result in reducing the ***development time***.

For example, if we look at the class of a ***rectangle*** then we can think of data members to be the ***width*** and ***length***. Now if we are to define a new class of a ***square***, then we will have similar data members with the special case of having the ***width*** = ***length***, a member function to calculate the area of a square is also needed. Instead of defining the class of square from scratch, we can think of the square as the special case of a rectangle, hence, making use of the class of a rectangle by inheriting its behavior and redefining the area function to work for the class of squares.

New classes can ***modify the behavior of their base classes*** by redefining member functions to define new behaviors with different number and/or types of parameters, a capability known as **polymorphism**.

With such concept, the correct function is called at runtime based on the type and number of parameters passed.

For example, the operation "+" in the following expression:

A = B + C , thought of as an Integer addition if both B and C are integers. If both B and C are double however, the operation will be thought of as a Double addition. This is polymorphism in its simple way.

General form of class declaration:

```
class class-name
{
        public:
          public-data-members;
          public-functions;

        private:
          private-data-members;
          private -functions;

        protected:
          protected-data-members;
          protected -functions;
};
```

Class members fall under one of three different access permission categories:

- **Public** members are accessible by all class users.
- **Private** members are only accessible by the class members.
- ·**Protected** members are only accessible by the class members and the members of a derived class.

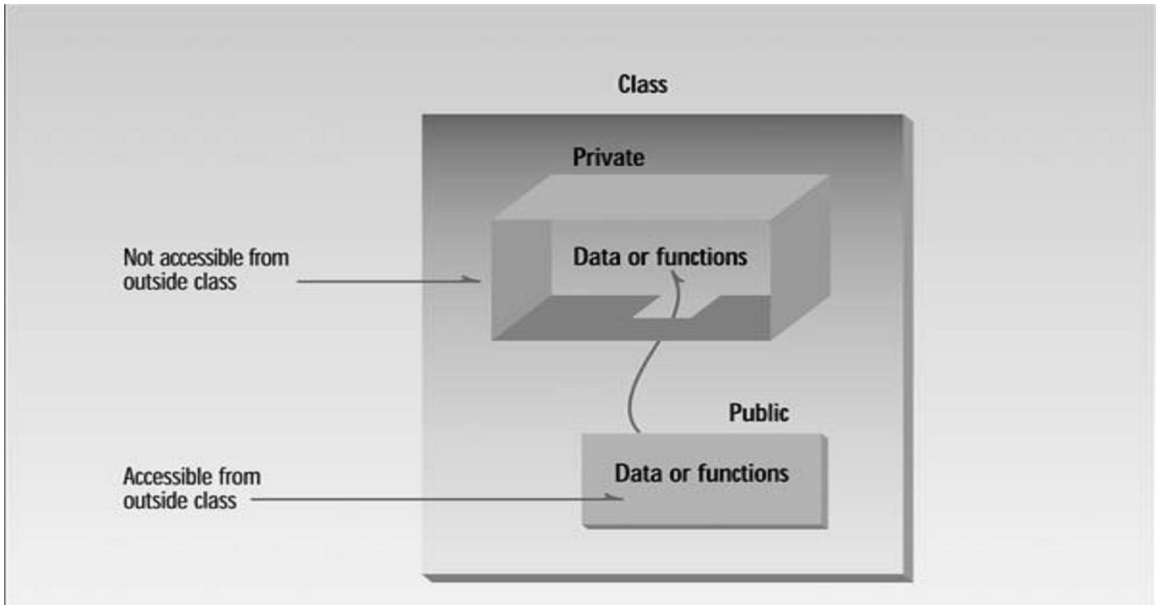


Figure 3: Public and Private Definition

Example 1:

Write a C++ program to modeling the Rectangle class.

```
# include <iostream.h>

class Rectangle
{
    public:
        int length , width;
        int area( )
        {
           return length * width;
        }
};


int main( )
{
  Rectangle my_rectangle;

  my_rectangle.length = 6;
  my_rectangle.width = 7;
  cout<< my_rectangle.area( );

  return 0;
}
```

Example 2:A Simple Class

```
#include <iostream>
using namespace std;
class smallobj      //define a class
{
private:
int   somedata; //class data
public:
void   setdata(int d)              //member function to set data
{
somedata = d;
 }
void showdata()                  //member function to display data
{ cout << "Data is " << somedata << endl; }
};
int main()
{
smallobj s1, s2;              //define two objects of class smallobj
s1.setdata(1066);       //call member function to set data
s2.setdata(1776);
s1.showdata();            //call member function to display data
s2.showdata();
return 0;
}
```

Example 3: write an oo program to define the coordinate of point and change the values of point.

```
#include<iostream.h>
class point {
         int xval , yval;
public:
   void setpt(int x , int y)
{
        xval=x;
        yval=y;
}
   void offsetpt(int x , int y)
{
        xval+=x;
        yval+=y;
                cout<<xval<<yval;

}
};
void main()
{
 point pt;
pt.setpt(10,20);
pt.offsetpt(2,2);
}
```

Class Constructors and Destructors:

A **class constructor** is a function that is executed automatically whenever a new instance of a given class is declared.

The main purpose of a class constructor is to perform any initializations related to the class instances via passing of some parameter values as initial values and allocate proper memory locations for that object.

Note1: A class constructor must have the same name as that of the associated class.

Note2: A class constructor has not return type not even void.

Note3: A class constructor can be overloaded

Example 1: Write an oo program to represent simple constructor.

```
#include<iostream.h>
class point {
     int xval,yval;
public:
    point(int x,int y)    //constructor
{
    xval=x;
    yval=y;
}
    void offsetpt(int x,int y)
{
    xval+=x;
    yval+=y;
        cout<<xval<<yval;

}
};

void main()
{
 point pt(10,20);
pt.offsetpt(2,2);
}
```

Example 2: Write an oo program to represent a rectangle constructor.

```
# include <iostream.h>

class Rectangle
{
    int length , width;
    public:
     Rectangle(int x, int y)
{
    length = x;
    width = y;
}

int area( )
{
    return (length *width);
}

};


void main( )
{
    Rectangle rect1(6,7);
    cout<< rect1.area( ) << endl;
}
```

A class may have more than one constructor. To avoid ambiguity, however, each of these must have a unique signature.

Example 3: Write an oo program to represent multiple constructors.

```
# include <iostream.h>
class Rectangle
{
    int length , width;
    public:
     Rectangle( )                    //constructor1
     {
          length = 7;
          width = 9;
     }

     Rectangle(int x, int y)     //constructor2
{
    length = x;
    width = y;
}
     int area( )
     {    return (length *width);
     }

};

void main( )
{
    Rectangle rect1(6,7);
    Rectangle rect2;

    cout<< rect1.area( ) << endl;
    cout<< rect2.area( ) << endl;
}
```

Example 4

```
#include <iostream>
class Counter
{
private:
int count; //count
public:
Counter() : count(0)      //constructor
{ /*empty body*/ }
void inc_count()          //increment count
{ count++; }
int get_count()          //return count
{ return count;  }
};
```

```
int main()
{
Counter c1, c2; //define and initialize
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count();
c1.inc_count(); //increment c1
c2.inc_count(); //increment c2
c2.inc_count(); //increment c2
cout << "\nc1=" << c1.get_count(); //display again
cout << "\nc2=" << c2.get_count();
cout << endl;
return 0;
};
```

Output:

C1=0

C2=0

C1=1

C2=2

Destructor

Just as a constructor is used to initialize an object when it is created, a destructor is used to clean up the object just before it is destroyed. A destructor always has the same name as the class itself, but is preceded with a ~ symbol. Unlike constructors, a class may have at most one destructor. A destructor never takes any arguments and has no explicit return type.

Destructors are generally useful for classes which have pointer data members which point to memory blocks allocated by the class itself.

In such cases it is important to release member-allocated memory before the object is destroyed. A destructor can do just that.

Example 1: Write an oo program to represent a simple destructor.

```
# include <iostream.h>

class Rectangle
{
   int length , width;
   public:
    Rectangle(int x, int y)             //constructor
{
   length = x;
   width = y;
}
    ~ Rectangle()                      //destructor
    {
        cout<<"destructor delete data"<<endl;
    }


int area( )
{
   return (length *width);
}

};


void main( )
{
   Rectangle rect1(6,7);
   cout<< rect1.area( ) << endl;
}
```

Example2: Write an oo program to represent destructor of pointer members.

```
# include <iostream.h>

class Rectangle
{
   int *length , *width;
   public:
    Rectangle(int x, int y)
{
   length= new int;
   width= new int;
   *length = x;
   *width = y;

}

    ~Rectangle()
{
   delete length;
   delete width;
}

int area( )
{
   return (*length **width);
}

};


void main( )
{
   Rectangle rect1(6,7);
   cout<< rect1.area( ) << endl;
}
```

Note: A class destructor proceeded with a tilde (~).

Friend function

Occasionally we may need to grant a function access to the nonpublic members of a class. Such an access is obtained by declaring the function a friend of the class.

There are two possible reasons for requiring this access:

• It may be the only correct way of defining the function.

• It may be necessary if the function is to be implemented efficiently.

Example1: Write an oo program to find the summation of point using friend function.

```
#include <iostream.h>

class point
{
private:
int xval,yval;
public:
    point()
    {
        xval=0;
        yval=0;
    cout<<"xval= "<<xval<<"yval= "<<yval<<endl;

    }
point(int x,int y)
    {
        xval=x+2;
        yval=y+3;
    cout<<"xval= "<<xval<<" "<<"yval= "<<yval<<endl;

    }

 friend int sum(point p);                    //friend function

};

int sum(point p)
{
int ss = p.xval + p.yval;
return (ss);
}

void main( )
{ point p2;
    point p(2,2);
    int dd;
        dd=sum(p);
    cout<<" the summation of coordinate x & y = "<<dd<<endl;
}
```

Example2: Write an oo program to represent a friend function .

```
#include <iostream.h>

class beta;                         //needed for frifunc declaration

class alpha
{
private:
int data;
public:
alpha() : data(3) { }
friend int frifunc(alpha, beta);      //friend function
};

class beta
{
private:
int data;
public:
beta() : data(7) { }
friend int frifunc(alpha, beta);      //friend function
};

int frifunc(alpha a, beta b)        //function definition
{
return( a.data + b.data );
}

int main()
{
alpha aa;
beta bb;
cout << frifunc(aa, bb) << endl; //call the function
return 0;
}
```

Example3:Write an oo program to find the square of distance using friend function with overloading constructors

```
#include <iostream.h>
class Distance
{
private:
int feet;
float inches;
public:
Distance()                          //constructor (no arguments)
{
    feet=0;
 inches=0.0 ;
}

Distance(int ft, float in)          //constructor (two arguments)
{
    feet=ft;
 inches=in;
}
void showdist()                              //display distance
{ cout<<feet <<" "<<inches <<" "; }
friend float square(Distance);                //friend function
};
//-----------------------------------------------------------
float square(Distance d)                     //return square of
{                                           //this Distance
float fltfeet = d.feet + d.inches/12;       //convert to float
float feetsqrd = fltfeet * fltfeet;          //find the square
return feetsqrd;                            //return square feet
}
////////////////////////////////////////////////////////////////
int main()
{
Distance dist(3, 6.0);                        //two-arg constructor (3'-6")
float sqft;
sqft = square(dist);                           //return square of dist
                                               //display distance and square
cout << "\nDistance = "; dist.showdist();
cout << "\nSquare = " << sqft << " square feet\n";
return 0;
}
```

Example 4:

```
#include <iostream.h>
class D3;
class point
{
     int xVal;
     int yVal;
 public:
   point(int x, int y): xVal(x), yVal(y) {}
   friend void func1(point a, D3 b);
};
class D3
{
private:
int zVal;
public:
    D3(int zz): zVal(zz) {}
  friend void func1(point a, D3 b);
};
void func1(point a, D3 b) {
cout << "\nxVal=" << a.xVal; cout << "\nyVal=" << a.yVal; cout << "\nzVal=" << b.zVal; }
int main()
{
point d1(3,2);
D3 d2(6);
func1(d1,d2);
cout << endl;
return 0;
}
```

friend class

The member functions of a class can all be made friends at the same time when you make the entire class a friend.

The extreme case of having all member functions of a class A as friends of another class B can be expressed in an *abbreviated* form:

```
class A;
class B {
//...
friend class A; // abbreviated form
};
```

Example 1:Write an oo program to represent a friend class

```
#include <iostream.h>
//////////////////////////////////////////////////////////////////
class alpha
{
private:
int data1;
public:
    alpha()
    {
        data1=99;
    }                          //constructor beta is a friend class
friend class beta;
};
//////////////////////////////////////////////////////////////////
class beta
{                                        //all member functions can access private alpha data
public:
void func1(alpha a) { cout<<"\n data1="<< a.data1; }
void func2(alpha a) { cout<<"\n data1= "<<a.data1; }
};
//////////////////////////////////////////////////////////////////
int main()
{
alpha a;
beta b;
b.func1(a);
b.func2(a);
cout<<endl;
return 0;
}
```

Example 2:Write an oo program to represent a friend class to class point

```
#include <iostream.h>

class point
{
private:
int xval,yval;
public:
    point(int x,int y)
    {
        xval=x+2;
        yval=y+3;

    }
    friend class display;
};

class display
{
public:
    void disdata(point p)
    {
        cout<<"xval= "<<p.xval<<" "<<"yval= "<<p.yval<<endl;
    }

};


void main( )
{   point p(2,2);
    display d;
        d.disdata(p);
    cout<<endl;
}
```

Example 3:

```
#include <iostream.h>
class point
{
int xVal;
int yVal;
public:
point(int x, int y): xVal(x), yVal(y)
{}
friend class D3;
};
class D3
{
private:

int  zVal;
public:
D3(int  zz): zVal(zz)
{}
void func1(point a)
{ cout << "\nxVal=" << a.xVal; cout << "\nyVal=" << a.yVal; cout <<
"\nzVal=" << zVal; }
};

int main()
{
D3 d1(6);
point d2(3,2);
d1.func1(d2);
cout << endl;
  return 0;
}
```

Output:

```
3
2
6
```

Default Arguments

As with global functions, a member function of a class may have default arguments. The same rules apply: all default arguments should be trailing arguments, and the argument should be an expression consisting of objects defined within the scope in which the class appears.

Surprisingly, a function can be called without specifying all its arguments. This won't work on just any function: The function declaration must provide default values for those arguments that are not specified.

Example 1:Write a simple program to represent a default argument

```
// missarg.cpp
// demonstrates missing and default arguments
#include <iostream>
using namespace std;

void repchar(char='*', int=45);    //declaration with
                                   //default arguments
int main()
   {
   repchar();                      //prints 45 asterisks
   repchar('=');                   //prints 45 equal signs
   repchar('+', 30);               //prints 30 plus signs
   return 0;
   }
//-------------------------------------------------------------
// repchar()
// displays line of characters
void repchar(char ch, int n)       //defaults supplied
   {                               //   if necessary
   for(int j=0; j<n; j++)          //loops n times
      cout << ch;                  //prints ch
   cout << endl;
   }
```

In this program the function repchar() takes two arguments. It's called three times from main(). The first time it's called with no arguments, the second time with one, and the third time with two. Why do the first two calls work? Because the called function provides default arguments, which will be used if the calling program doesn't supply them. The default arguments are specified in the declaration for repchar():

void repchar(char='*', int=45); //declaration

The default argument follows an equal sign, which is placed directly after the type name. You can also use variable names, as in

void repchar(char reptChar='*', int numberReps=45);

If one argument is missing when the function is called, it is assumed to be the last argument. The repchar() function assigns the value of the single argument to the ch parameter and uses the default value 45 for the n parameter. If both arguments are missing, the function assigns the default value '*' to ch and the default value 45 to n. Thus the three calls to the function all work, even though each has a different number of arguments.

Implicit Member Argument

When a class member function is called, it receives an implicit argument which denotes the particular object (of the class) for which the function is invoked. For example, in

Point pt(10,20);

pt.OffsetPt(2,2);

pt is an implicit argument to **OffsetPt**. Within the body of the member function, one can refer to this implicit argument explicitly as **this**, which denotes a pointer to the object for which the member is invoked. Using this, OffsetPt can be rewritten as:

Point::OffsetPt (int x, int y)

{

this->xVal += x; // equivalent to: xVal += x;

this->yVal += y; // equivalent to: yVal += y;}

The **this** pointer can be used for referring to member functions in exactly the same way as it is used for data members. It is important to bear in mind, however, that **this** is defined for use within member functions of a class only.

Example 1:Write an oo program to represent the *this* pointer

```
#include<iostream.h>
class point {
      int xval,yval;
public:

     void offsetpt(int x,int y)
{
     this->xval=x+2;
     cout<<this->xval;
        this->yval=y+5;
         cout<<this->yval;
         cout<<'\n';

}
};
void main()
{
 point pt;
pt.offsetpt(2,2);
}
```

Example 2:Write an oo program to represent the *this* pointer

```
#include <iostream.h>
//////////////////////////////////////////////////////////////////
class what
{
private:
int alpha;
public:
void tester()
{
this->alpha = 11; //same as alpha = 11;
cout<<this->alpha; //same as cout << alpha;
}
};
//////////////////////////////////////////////////////////////////
int main()
{
what w;
w.tester();
cout<<endl;
return 0;
}
```

Scope Operator

When calling a member function, we usually use an abbreviated syntax. For example:

pt.OffsetPt(2,2); // abbreviated form

This is equivalent to the full form:

pt.Point::OffsetPt(2,2); // full form

The full form uses the binary **scope operator ::** to indicate that OffsetPt is a member of Point.

In some situations, using the scope operator is essential. For example, the case where the name of a class member is hidden by a local variable (e.g., member function parameter) can be overcome using the scope operator:

```
class Point {
public:
Point (int x, int y) { Point::x = x; Point::y = y; }
//...
private:
int x, y;
}
```

Here **x** and **y** in the constructor (inner scope) hide **x** and **y** in the class (outer scope). The latter are referred to explicitly as **Point::x** and **Point::y.**

Example 1:

Write a C++ program under the concept of class constructor to modeling the Rectangle class, using scope resolution operator with the member function.
Note: all data member are private.

```
# include <iostream.h>

class Rectangle
{
   int length , width;
   public:
         Rectangle(int, int);
         int area( );
};

Rectangle :: Rectangle(int x, int y)
{
  length = x;

  width = y;
}

int Rectangle::area( )
{
  return length * width;
}

int main( )
{
  Rectangle my_rectangle(6,7);

  cout<< my_rectangle.area( );

  return 0;
}
```

Member Initialization List

There are two ways of initializing the data members of a class.

1) The first approach involves initializing the data members using assignments in the body of a constructor. For example:

```
class Image {
public:
Image (const int w, const int h);
private:
int width;
int height;
//...
};
Image::Image (const int w, const int h)
{
width = w;
height = h;
//...
}
```

Example 1: Write an oo program to initialize the data member of a class using assignments in the body of constructor.

```
# include <iostream.h>
class Rectangle
{
    int length , width;
public:
    Rectangle(const int l, const int w);
    int area( );
};

Rectangle :: Rectangle(const int l, const int w)
{
  length = l;
  width = w;
}

int Rectangle::area( )
{
   return length * width;
}

void main( )
{
    Rectangle my_rectangle(6,7);
    cout<<'\n';
    cout<< my_rectangle.area( );
    cout<<'\n';

}
```

2) The second approach uses a member initialization list in the definition of a constructor. For example:

class Image {

public:

Image (const int w, const int h);

private:

int width;

int height;

//...

};

Image::Image (const int w, const int h) : width(w), height(h)

{

//...

}

Example 2: Write an oo program to initialize the data member of a class in the definition of constructor.

```
# include <iostream.h>
class Rectangle
{
public:
    Rectangle(const int l, const int w);
    int area( );

private:
    int length , width;

};

Rectangle :: Rectangle(const int l, const int w):length(l),width(w)
{
  cout<<"i am in rectangle constructor";
}

int Rectangle::area( )
{
   return length * width;
}

void main( )
{
    Rectangle my_rectangle(6,7);
    cout<<'\n';
    cout<< my_rectangle.area( );
    cout<<'\n';

}
```

Constant member

A class data member may define as constant. For example:

```
class Image {
const int width;
const int height;
//...};
```

However, data member constants cannot be initialized using the same syntax as for other constants:

```
class Image {
const int width = 256; // illegal initializer!
const int height = 168; // illegal initializer!
//...
};
```

The correct way to initialize a data member constant is through a member initialization list:

```
class Image {
public:
Image (const int w, const int h);
private:
const int width;
const int height;
//...
};
Image::Image (const int w, const int h) : width(w), height(h)
```

{

//...

}

As one would expect, no member function is allowed to assign to a constant data member.

Constant Function Argument

If an argument is large, passing by reference is more efficient because, behind the scenes, only an address is really passed, not the entire variable.

Suppose you want to pass an argument by reference for efficiency, but not only do you want the function not to modify it, you want a guarantee that the function *cannot* modify it.

To obtain such a guarantee, you can apply the const modifier to the variable in the function declaration.

Example 1:Write a simple program to represent a constant member

```
//constarg.cpp
//demonstrates constant function arguments

void aFunc(int& a, const int& b);  //declaration

int main()
   {
   int alpha = 7;
   int beta = 11;
   aFunc(alpha, beta);
   return 0;
   }
//--------------------------------------------------------------
void aFunc(int& a, const int& b)   //definition
   {
   a = 107;   //OK
   b = 111;   //error: can't modify constant argument
   }
```

Here we want to be sure that aFunc() can't modify the variable beta. (We don't care if it modifies alpha.) So we use the const modifier with beta in the function declaration (and definition):

void aFunc(int& alpha, const int& beta);

Constant Member Functions

We can apply const to variables of basic types such as int to keep them from being modified. In a similar way, we can apply const to objects of classes. When an object is declared as const, you can't modify it.

Example 2:Write an oo program to represent a constant member function

```
// constObj.cpp
// constant Distance objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                        //2-arg constructor
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //user input; non-const func
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance; const func
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   const Distance football(300, 0);

// football.getdist();            //ERROR: getdist() not const
   cout << "football = ";
   football.showdist();           //OK
   cout << endl;
   return 0;
   }
```

Static Members

A data member of a class can be defined to be static. This ensures that there will be exactly one copy of the member, shared by all objects of the class.

Example 1: Write an oo program to represent a static members.

```
// statfunc.cpp
// static functions and ID numbers for objects
#include <iostream>

////////////////////////////////////////////////////////////////
class gamma
   {
   private:
      static int total;         //total objects of this class
                                //   (declaration only)
      int id;                   //ID number of this object
   public:
      gamma()                   //no-argument constructor
         {
         total++;               //add another object
         id = total;            //id equals current total
         }
      ~gamma()                  //destructor
         {
         total--;
         cout << "Destroying ID number " << id  << endl;
         }
      static void showtotal()   //static function
         {
         cout << "Total is " << total << endl;
         }
      void showid()             //non-static function
         {
         cout << "ID number is " << id << endl;
         }
   };
//--------------------------------------------------------------
int gamma::total = 0;            //definition of total
////////////////////////////////////////////////////////////////
int main()
   {
   gamma g1;
   gamma::showtotal();

   gamma g2, g3;
   gamma::showtotal();

   g1.showid();
   g2.showid();
   g3.showid();
   cout << "----------end of program----------\n";
   return 0;
   }
```

Now the function can be accessed using only the class name. Here's the output:

Total is 1
Total is 3
ID number is 1
ID number is 2
ID number is 3
----------end of program--------
Destroying ID number 3
Destroying ID number 2
Destroying ID number 1

Example 2: Write an oo program to represent a static member.

```
# include <iostream.h>
class test{

static int count;//count is static
int code;
public:
void setcode()
{
     cout<<"i am in set code"<<endl;
     code=++count;
}
void showcode()

{    cout<<"i am in show code"<<endl;

     cout<<"object number"<<code<<"\n";
}
static void showcount()
{    cout<<"i am in showcount"<<endl;

     cout<<"count:"<<count<<"\n";
}
};
int test::count;//count defined
void main ()
{ test t1,t2;
t1.setcode ();
t2.setcode ();
test::showcount ();
test t3;
t3.setcode ();
test::showcount ();
t1.showcode();
t2.showcode();
t3.showcode();
}
```