Chapter ONE

Function

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function. The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location, and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows:

```
return_type  function_name( parameter list )
{
body of the function
}
```

A C++ function definition consists of a function header and a function body.

Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together establish the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list

refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body:** The function body contains a collection of statements that define what the function does.

Example: Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers
int max(int num1, int num2)
{
// local variable declaration
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts:

return type function name(parameter list);

For the above defined function max(), following is the function declaration:

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also valid declaration: int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns Program control back to the main program. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <iostream>
using namespace std;
int max(int num1, int num2); // function declaration
int main ()
{
int a = 100; int b = 200; int ret; // local variable declaration:
ret = max(a, b);// // calling a function to get max value.
cout << "Max value is : " << ret << endl;
return 0;
}
// function returning the max between two numbers
int max(int num1, int num2)
{
int result; // local variable declaration
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

I kept max () function along with main () function and compiled the source code. While running final executable, it would produce the following result: Max value is : 200

Function Arguments

If a programmer wants to use arguments, he/she must declare variables (Formal Parameter) that accept the values of the arguments (Actual Parameter). The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

Types of Function Calling

Call by Value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

Example to swap two values using calling by value

```
#include <iostream.h>
void swap(int x, int y); // function declaration
int main ()
{
int a = 100;int b = 200; // local variable declaration:
cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;
swap(a, b); // calling a function to swap the values.
cout << "After swap, value of a :" << a << endl;
cout << "After swap, value of b :" << b << endl;
return 0;
}

// function definition to swap the values.
void swap(int x, int y)
{
int temp;
```

Out put

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200

- Which shows that there is no change in the values though they had been changed inside the function?

```
temp = x; /* save the value of x */
x = y; /* put y into x */
y = temp; /* put x into y */
return;}
```

Call by Reference

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument. To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap ()**, which exchanges the values of the two integer variables pointed to by its arguments.

Example to swap two values using calling by reference

```
#include <iostream.h>
void swap(int &x, int &y) // function definition to swap the values.
{
int temp;
temp = x; //save the value at address x
x = y; // put y into x
y = temp; // put x into y
return;
 }
int main ()
{
int a = 100;int b = 200; // local variable declaration:
cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;
swap(a, b); /* calling a function to swap the values using variable reference.
cout << "After swap, value of a :" << a << endl;
```

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

- Which shows changes made to the parameter affect the passed argument? **& is called "address of" operator** is used to pass the address of a variable to the called function or variable.

```
cout << "After swap, value of b :" << b << endl;
return 0;
}
```

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

Consider the following example:

```
#include <iostream.h>
int sum(int a, int b=20){
int result;
result = a + b;
return (result);}
int main (){
int a = 100; int b = 200; int result; // local variable declaration:
result = sum(a, b); // calling a function to add the values.
cout << "Total value is :" << result << endl;
result = sum(a); // calling a function again as follows.
cout << "Total value is :" << result << endl;
return 0; }
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300
Total value is :120
```

Arrays as parameters

- ✓ At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass by value a complete block of memory as a parameter, even if it is ordered as

an array, to a function, but it is allowed to pass its address, which has almost the same practical effect and is a much faster and more efficient operation.

- ✓ In order to admit arrays as parameters the only thing that we must do when declaring the function is to specify in the argument the base **type** for the array that it contains, an identifier and a pair of brackets **[]** .

For example, void procedure (int arg[])

- ✓ admits a parameter of type "Array of int " called arg . In order to pass to this function an array declared as:

```
int myarray [40];
```

- ✓ *it would be enough with a call like this:*

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters

#include <iostream.h>

#include <conio.h>

void printarray (int arg[], int length)
{
    for (int n=0; n<length; n++)
        cout << arg[n] << " ";
    cout << "\n";
}

void mult(int arg[], int length)
{
    for (int n=0; n<length; n++)
      arg[n]=2*arg[n];
}

void main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8, 10};
```

```
    printarray (firstarray,3);
    printarray (secondarray,5);

    mult(firstarray,3);

    cout<<"first array after being doubled is\n";

    printarray (firstarray,3);
   getch()
}
```

- ✓ As you can see, the first argument (int arg[]) admits any array of type int , whatever its length is, for that reason we have included a second parameter that informs the function the length of each array that we pass to it as the first parameter so that the for loop that prints out the array can have the information about the size we are interested about. The function **mult** doubles the value of each element and the **firstarray** is passed to it. After that the display function is called. The output is modified showing that arrays are passed by reference.
- ✓ To pass an array by value, pass each element to the function

Global versus local variables

- Everything defined at the program scope level (outside functions) is said to have a *global scope*, meaning that the entire program knows each variable and has the capability to change any of them.

  Eg.

```
int year = 1994;//global variable
int max(int,int);//gloabal funcrion
int main(void)
{
        //...
}
```

- Global variables are visible ("known") from their point of definition down to the end of the program.
- Each block in a program defines a local scope. Thus the body of a function represents a local scope. The parameters of a function have the same scope as the function body.
- Variables defined within a local scope are visible to that scope only. Hence, a variable need only be unique within its own scope. Local scopes may be nested, in which case the inner scope overrides the outer scopes. Eg:

```
int xyz;//xyz is global
void Foo(int xyz)//xyz is local to the body of Foo
 {
        if(xyz > 0)
        {
```

```
            Double xyz;//xyz is local to this block
            ...
        }
    }
```

Scope Operator

- Because a local scope overrides the global scope, having a local variable with the same name as a global variable makes the latter inaccessible to the local scope.
- Eg

```
int num1;
void fun1(int num1)
{
    //...
}
```

- The global num1 is inaccessible inside fun1(), because it is overridden by the local num1 parameter.
- This problem is overcome using the scope operator '::' which takes a global entity as argument.

```
int num1 = 2;
Void fun1(int num1)
{
    //...
    num1=33;
    cout<<num1; // the output will be 33
    cout<<::num1;  //the output will be 2 which is the global
    if(::num1 != 0)//refers to global num1
        //...
}
```

Automatic versus static variables

- The terms automatic and static describe what happens to local variables when a function returns to the calling procedure. By default, all local variables are automatic, meaning that they are erased when their function ends. You can designate a variable as automatic by prefixing its definition with the term **auto**.
- In computer programming, an **automatic variable** is a local **variable** which is allocated and de allocated automatically when program flow enters and leaves the **variable's** scope.
- Eg. automatic local variables:

```
Int main(){
        int i;
        auto float x;}
```

- The opposite of an automatic is a static variable. All global variables are static and, as mentioned, all static variables retain their values. Therefore, if a local variable is static, it too retains its value when its function ends-in case this function is called a second time.
- To declare a variable as static, place the static keyword in front of the variable when you define it.
- Static variables can be declared and initialized within the function, but the initialization will be executed only once during the first call.
- If **static variables** are declared outside of all functions it will have **global** scope or it is declared within a function it will have scope within a function and it will retain until the life of program.
- If static variables are not declared explicitly, they will be declared to 0 automatically.

Eg.

```
Void my- fun ()
{
  Static int num;
  Static int count = 2;
 count=count*5;
num=num+4;
}
```

- In the above example:
  - During the first call of the function my_fun(), the static variable count will be initialized to 2 and will be multiplied by 5 at line three to have the value 10. During the second call of the same function count will have 10 and will be multiplied by 5.
  - During the first call of the function my_fun(), the static variable num will be initialized to 0 (as it is not explicitly initialized) and 4 will be added to it at line four to have the value 4. During the second call of the same function num will have 4 and 4 will be add to it again.
- **N.B**. if local variables are static, their values remain in case the function is called again.

Overloaded Functions

- Unlike C, C++ lets you have more than one function with the same name. In other words, you can have three functions called abs() in the same program.
- Functions with the same name are called overloaded functions. C++ requires that each overloaded functions differ in its argument list. Overloaded functions enable you to have similar functions that work on different types of data.
- Suppose that you wrote a function that returned the absolute value of whatever number you passed to it:

```
int iabs(int i)
{       if(i<0)
            return (i*-1);
```

```
        else
            return (i);
    }

    float fabs(float x)
    {       if(x<0.0)
            Return (x * -1.0);
            else
             Return (x);
    }
```

- Without using overloading, you have to call the function as:

```
int ans = iabs(weight);//for int arguments

float ans = fabs(weight);//for float arguments
```

- But with overloading, the above code can be used as:

```
int abs(int i);
float abs(float x);
int main()
{
    int y;
    float p;
    ...
    ians = abs(y);//calling abs with int arguments
    fans = abs(p);//calling abs with float arguments
    ...
}
int abs(int i)
{
    if(i<0)
        return i*-1;
    else
        return i;
}

float abs(flaot x)
{
    if(x<0.0)
```

```
            return x*-1.0;
        else
            return x;
    }
```

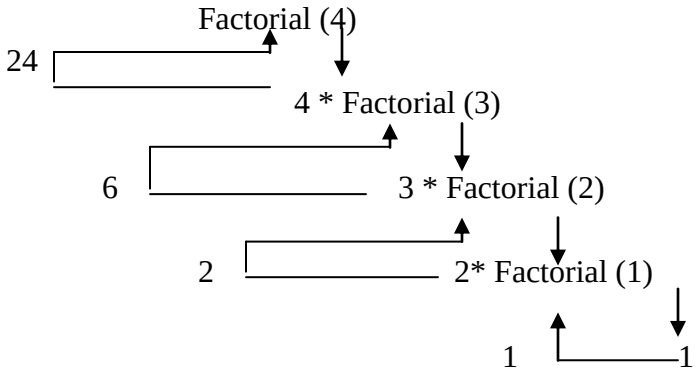
⇨ **N.B**: if two or more functions differ only in their return types, C++ can't overload them. Two or more functions that differ only in their return types must have different names and can't be overloaded

Recursion

- A function which calls itself is said to be recursive. Recursion is a general programming technique applicable to problems which can be defined in terms of themselves. Take the factorial problem, for instance which is defined as:
  - Factorial of 0 is 1
  - Factorial of a positive number n is n time the factorial of n-1.
- The second line clearly indicates that factorial is defined in terms of itself and hence can be expressed as a recursive function.

```
int Factorial(unsigned int n )
{
   return n = = 0 ? 1 : n * factorial(n-1);
}
```

- For n set to 4, the following figure shows the recursive call:



- The stack frames for these calls appear sequentially on the runtime stack, one after the other.
- A recursive function must have at least one termination condition which can be satisfied. Otherwise, the function will call itself indefinitely until the runtime stack overflows.
- The three necessary components in a recursive method are:
  1. A test to stop or continue the recursion
  2. An end case that terminates the recursion
  3. A recursive call(s) that continues the recursion
- let us implement two more mathematical functions using recursion
- e.g the following function computes the sum of the first N positive integers 1,2,…,N. Notice how the function includes the three necessary components of a recursive method.

```
int sum(int N)
{
        if(N==1)
                return 1;
        else
                Return N+sum(N-1);
}
```

- The last method computes the exponentiation A^n where A is a real number and N is a positive integer. This time, we have to pass two arguments. A and N. the value of A will not change in the calls, but the value of N is decremented after each recursive call.

```
float expo(float A, int N)
{
        if(N==1)
                return A;
        else
                return A * expo(A,N-1);
}
```

Recursion versus iteration

- Both iteration and recursion are based on control structure. Iteration uses a repetition structure (such as for, while, do…while) and recursive uses a selection structure (if, if else or switch).
- Both iteration and recursive can execute infinitely-an infinite loop occurs with iteration if the loop continuation test become false and infinite recursion occurs id the recursion step doesn't reduce the problem in a manner that coverage on a base case.
- Recursion has disadvantage as well. It repeatedly invokes the mechanism, and consequently the overhead of method calls. This can be costly in both processor time and memory space. Each recursive call creates another copy of the method (actually, only the function's variables); this consumes considerable memory.
- **N.B**: Use recursion if:
    1. A recursive solution is natural and easy to understand
    2. A recursive solution doesn't result in excessive duplicate computation.
    3. the equivalent iterative solution is too complex and
    4. Of course, when you are asked to use one in the exam!!

Chapter Two

2. Arrays and String manipulation

2.1. What is An Array?

- A collection of identical data objects, which are stored in consecutive memory locations under a common heading or a variable name. In other words, an array is a group or a table of values referred to by the same name. The individual values in array are called elements. Array elements are also variables.
- Array is data structure bigger than an individual variable.
- Set of values of the same type, which have a single name followed by an index. In C++, square brackets appear around the index right after the name
- A block of memory representing a collection of many simple data variables stored in a separate array element, and the computer stores all the elements of an array consecutively in memory.
- Array occupy space in memory, programmer specify type and no of element.

2.1.1. Properties of arrays

- **Ordered** (numbered or indexed): Arrays in C++ are zero-bounded; that is the index of the first element in the array is 0 and the last element is N-1, where N is the size of the array.
- **Array Size must inside bound**: It is illegal to refer to an element outside of the array bounds, and your program will crash or have unexpected results, depending on the compiler.
- **Homogeneous**: Array can only hold values of one type.
- **Random Access**: in array we can access any element randomly.
- **Contiguous:** Array is a location which store values sequentially.

2.1.2. Array declaration

- Declaring the name and type of an array and setting the number of elements in an array is called dimensioning the array. The array must be declared before uses like other variables. In the array declaration must define:
    1. The type of the array (i.e. integer, floating point, char etc.)
    2. Name of the array,
    3. The total number of memory locations to be allocated or the maximum value of each subscript (index). I.e. the number of elements in the array.
- So the general syntax for the declaration is:

DataTypename arrayname [array size];

- The expression array size, which is the number of elements, must be a constant such as 10 or a symbolic constant declared before the array declaration, or a constant expression such as 10*sizeof (int), for which the values are known at the time compilation takes place.
- Note: array size cannot be a variable whose value is set while the program is running.

Thus to declare an integer with size of 10 having a name of num is:

int num [10];

- This means: ten consecutive two byte memory location will be reserved with the name num. That means, we can store 10 values of type int without having to declare 10 different variables each one with a different identifier. Instead of that, using an array we can store 10 different values of the same type, int for example, with a unique identifier.

2.1.3. Initializing Arrays

- When declaring an array of local scope (within a function), if we do not specify the array variable will not be initialized, so its content is undetermined until we store some values in it.
- If we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare:

int day [5];

⇨ every element of day will be set initially to 0:

⇨

	0	1	2	3	4
day	0	0	0	0	0

- But additionally, when we declare an Array, we have the possibility to assign initial values to each one of its elements using curly brackets { } . For example:

int day [5] = { 16, 2, 77, 40, 12071 };

- The above declaration would have created an array like the following one:

	0	1	2	3	4
day	16	2	77	40	12071

- The number of elements in the array that we initialized within curly brackets { } must be equal or less than the length in elements that we declared for the array enclosed within square brackets [ ]. If we have less number of items for the initialization, the rest will be filled with zero.

- For example, in the example of the day array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element. If we ignore the last initial value (12071) in the above initialization, 0 will be taken automatically for the last array element.
- Because this can be considered as useless repetition, C++ allows the possibility of leaving empty the brackets [ ], where the number of items in the initialization bracket will be counted to set the size of the array.

  int day [] = { 1, 2, 7, 4, 12,9 };
- The compiler will count the number of initialization items which is 6 and set the size of the array day to 6 (i.e.: day[6])
- You can use the initialization form only when defining the array. You cannot use it later, and cannot assign one array to another once. I.e.

```
int arr [] = {16, 2, 77, 40, 12071};
int ar [4];
ar[]={1,2,3,4};//not allowed
arr=ar;//not allowed
```

- Note: when initializing an array, we can provide fewer values than the array elements. E.g. int a [10] = {10, 2, 3}; in this case the compiler sets the remaining elements to zero.

2.1.4. Accessing and processing array elements

- In any point of the program in which the array is visible we can access individually anyone of its elements for reading or modifying it as if it was a normal variable. To access individual elements, index or subscript is used. The format is the following:

  name [*index*] ;

Example : int day [] = { 1, 2, 7, 4, 12,9 };

- ***To access the third element***

***Cout*<<day[2];** //as the third element is found at index 2 since array index start from 0 and end with N-1,to access the last element use day[5]. But if we wrote day [6], we would be acceding to the seventh element of day and therefore exceeding the size of the array. This might give you either error or unexpected value depending on the compiler.

- In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can cause certain detectable problems, since they do not cause compilation errors but they can cause unexpected results

or serious errors during execution. The reason why this is allowed will be seen ahead when we begin to use pointers.

- At this point it is important to be able to clearly distinguish between the two uses the square brackets [ ] have for arrays.
  - ⇨ One is to set the size of arrays during declaration
  - ⇨ The other is to specify indices for a specific array element when accessing the elements of the array
- We must take care of not confusing these two possible uses of brackets [ ] with arrays:

  Eg: int day[5]; *// declaration of a new Array (begins with a type name)*
  day[2] = 7; *// access to an element of the Array.*
- Other valid operations with arrays in accessing and assigning:

```
int a=1;
day[0]=a;
day[a]=5;
b=day[a+2];
day [day[a]] = day [2] + 5;
day [day[a]] = day[2] + 5;
```

Eg: A*rrays example ,display the sum of the numbers in the array*

```
#include<iostream.h>
int day [ ] = {16, 2, 77, 40, 12071};
int n, result=0;
int main () {
    for ( n=0 ; n<5 ; n++ ) {
        result += day[n];
    }
    cout << result;
}
```

Arrays in C++

- Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer:
- **Multi-dimensional arrays:-** C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. Consists more than two dimensions.
- **Pointer to an** array:-You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
- **Passing arrays to functions**:-You can pass to the function a pointer to an array by specifying the array's name without an index.
- **Return array from functions:-** C++ allows a function to return an array.

Multi-dimensional Arrays

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration: type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional 5 . 10 . 4

integer array: int threedim[5][10][4];

Two-Dimensional Arrays

Two-dimensional arrays, the most common multidimensional arrays, are used to store information that we normally represent in table form. Two-dimensional arrays, like one-dimensional arrays, are homogeneous. This means that all of the data in a two-dimensional array is of the same type. Examples of applications involving two-dimensional arrays include:

- A seating plan for a room (organized by rows and columns),
- A monthly budget (organized by category and month), and
- A grade book where rows might correspond to individual students and columns to student scores.

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

type arrayName [x][y];

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier. A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Example: The following declarations set aside storage for a two-dimensional array called labScores which contains 40 rows and 14 columns. Rows correspond to a particular student and columns correspond to a particular lab score.

```
const int MAX_STUDENTS=40;
const int MAX_LABS=14;
int labScores [MAX_STUDENTS][MAX_LABS];
```

Exercise 1:
Show C++ statements to declare the following two-dimensional array:

1. Declare a two-dimensional array which can be used to store a yearly budget. Each row of the array corresponds to a particular budgeted item like rent, electric, etc. There are at most 15 items to be budgeted. Each column of the array corresponds to a month, January, February, etc. Of course there are 12 columns corresponding to the 12 months of the year. All the data to be placed in the array consists of real numbers.

2. Manipulation of a two-dimensional array requires the manipulation of two indices. When the two-dimensional array labScores is declared, enough storage is set aside to hold a table containing 40 rows and 14 columns for a total of 40 * 14 = 560 integer values. To access one particular value, we must specify the row and column. The

row index ranges from 0 to MAX_STUDENTS-1 (39) and the column index ranges from 0 to MAX_LABS-1 (13). Thus the table can be visualized as: This two-dimensional array may be viewed as a one-dimensional array having 40 elements where each element is an array of 14 values.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
{0, 1, 2, 3} , /* initializers for row indexed by 0 */
{4, 5, 6, 7} , /* initializers for row indexed by 1 */
{8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example: int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

int val = a[2][3];//These statement will take 4th element from the 3rd row of the array.

In our labScores example, suppose we wish to indicate that the second student (corresponding to row 1) made a 90 on lab 10 (corresponding to column 9).

We might use the statement: labScores [1][9] = 90;

Array indices may be integer constants (as in the above example), variables, or expressions. They should be within the bounds of the array.

```
#include <iostream>
using namespace std;
int main ()
{
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};.
for ( int i = 0; i < 5; i++ )
for ( int j = 0; j < 2; j++ )
{
cout << "a[" << i << "][" << j << "]: ";
cout << a[i][j]<< endl;  // output each array element's value
} return 0; }
```

Output

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]:8
```

As explained above, you can have arrays with any number of dimensions,

although it is likely that most of the arrays you create will be of one or two dimensions.

1.1 Example: Add Two Matrices using Multi-dimensional Arrays

```
#include <iostream.h>
int main(){
 int r, c, a[100][100], b[100][100], sum[100][100], i, j;
 cout << "Enter number of rows and columns (between 1 and 100): ";
 cin >> r>>c
cout << endl << "Enter elements of 1st matrix: " << endl;
   // Storing elements of first matrix entered by user.
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
      {
         cout << "Enter element a" << i + 1 << j + 1 << " : ";
         cin >> a[i][j];
      }
   // Storing elements of second matrix entered by user.
   cout << endl << "Enter elements of 2nd matrix: " << endl;
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
      {
         cout << "Enter element b" << i + 1 << j + 1 << " : ";
         cin >> b[i][j];
      }
   // Adding Two matrices
   for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
           sum[i][j] = a[i][j] + b[i][j];
   // displaying the resultant sum matrix.
   cout << endl << "Sum of two matrix is: " << endl;
   for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
           cout << sum[i][j] << "  ";
           if(j == c - 1)
               cout << endl;
       }    return 0;}
```

2.2. Strings Manipulation

2.2.1. What are Strings?

- In all programs and concepts we have seen so far, we have used only numerical variables, used to express numbers exclusively. But in addition to numerical variables there also exist strings of characters that allow us to represent successive characters, like words, sentences, names, texts, etc. Until now we have only used them as constants, but we have never considered variables able to contain them.

- In C++ there is no specific *elementary* variable type to store string of characters. In order to fulfill this feature we can use arrays of type char, which are successions of **char** elements. Remember that this data type (char**)** is the one used to store a single character, for that reason arrays of them are generally used to make strings of single characters.
- For example, the following array (or string of characters) can store a string up to 20 characters long. You may imagine it thus:

  char name [20];

  name

- This maximum size of 20 characters is not required to be always fully used. For example, **name** could store at some moment in a program either the string of characters "Hello" or the string "studying C++". Therefore, since the array of characters can store shorter strings than its total length, there has been reached a convention to end the valid content of a string with a null character, whose constant can be written as **'\0'.**
- We could represent **name** (an array of 20 elements of type **char**) storing the strings of characters "Hello" and "Studying C++" in the following way:

H	e	l	l	0	\0														

S	t	u	d	y	i	n	g		C	+	+	\0						

- Notice how after the valid content it is included a null character ('\0') in order to indicate the end of string. The empty cells (elements) represent indeterminate values.

2.2.2. Initialization of Strings

- Because strings of characters are ordinary arrays they fulfill same rules as any array. For example, if we want to initialize a string of characters with predetermined values we can do it in a similar way to any other array:
- char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
- In this case we would have declared a string of characters (array) of 6 elements of type **char** initialized with the characters that compose **Hello** plus a null character **'\0'** .
- Nevertheless, string of characters have an additional way to initialize its values: using constant strings.

- Unlike single quotes (') which allow to specify single character constants, double quotes (") are constants that specify a succession of characters. These strings enclosed between double quotes have always a null character ('\0') automatically appended at the end.
- Therefore we could initialize the string **mystring** with values by any of these two ways:

  char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };

  char mystring [] = "Hello";
- In both cases the Array or string of characters **mystring** is declared with a size of 6 characters (elements of type **char**): the 5 characters that compose **Hello** plus a final null character ('\0') which specifies the end of the string and that, in the second case, when using double quotes (") it is automatically appended.
- Before going further, you should note that the assignation of multiple constants like double-quoted constants (") to arrays are only valid when initializing the array, that is, at the moment when declared.
- The following expressions within a code are not valid for arrays

  mystring="Hello";

  mystring[] = "Hello";
- neither would be: mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };
- So remember: We can "assign" a multiple constant to an Array only at the moment of initializing it.

2.2.3. Assigning Values to Strings

- Just like any other variables, array of character can store values using assignment operators. But the following is not allowed. mystring=”Hello”;
- This is allowed only during initialization. Therefore, since the value of an assignation can only be an element of an array and not the entire array, what would be valid is to assign a string of characters to an array of char using a method like this:

  mystring[0]='H';

  mystring[1]='e';

  mystring[2]='l';

  mystring[3]='l';

  mystring[4]='o';

  mystring[5] = '\0';

Example

```
#include <iostream.h>
int main (){
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
cout << "Greeting message: ";
cout << greeting << endl;
return 0;}
```

When the above code is compiled and executed, it produces the following result: Greeting message: Hello

C++ supports a wide range of functions that manipulate null-terminated strings:

2.2.4. Converting strings to other types

- Due to that a string may contain representations of other data types like numbers it might be useful to translate that content to a variable of a numeric type. For example, a string may contain **"1977"** , but this is a sequence of 5 chars not so easily convertible to a single integer data type. The **cstdlib** (**stdlib.h**) library provides three useful functions for this purpose:
  - **atoi:** converts string to **int** type.
  - **atol:** converts string to **long** type.
  - **atof:** converts string to **float** type.
- All of these functions admit one parameter and return a value of the requested type (int , long or float). These functions combined with getline method of cin are a more reliable way to get the user input when requesting a number than the classic cin>> method:

```
// cin and ato* functions
#include <iostream.h>
#include <stdlib.h>
#include<conio.h>
int main()
{       clrscr();
            char mybuffer[100];
            float price;
            int quantity;
            cout << "Enter price: ";
            cin.getline (mybuffer,100);
            price = atof (mybuffer);
            cout << "Enter quantity: ";
            cin.getline (mybuffer,100);
            quantity = atoi (mybuffer);
            cout<<"\nafter conversion :\n";
            cout<<"\nprice is : "<<price;
            cout<<"\nquantity is : "<<quantity;
            cout << "\nTotal price: " << price*quantity;
```

```
        getch();
        return 0;
}
```

2.2.5. Functions to manipulate strings

- The **cstring** library (string.h) defines many functions to perform some manipulation operations with

a) String length

- Returns the length of a string, not including the null character (\0).

  strlen (const char* *string* **);**

b) String Concatenation:

- Appends *src* string at the end of *dest* string. Returns *dest*.
- The string concatenation can have two forms, where the first one is to append the whole content of the source to the destination the other will append only part of the source to the destination.
  - Appending the whole content of the source

    strcat (char* *dest* **, const char*** *src* **);**
  - Appending part of the source

    strncat (char* *dest* **, const char*** *src,* **int** *size* **);**

    Where size is the number characters to be appended

c) String Copy:

- Overwrites the content of the *dest* string by the *src* string. Returns *dest*.
- The string copy can have two forms, where the first one is to copying the whole content of the source to the destination and the other will copy only part of the source to the destination.
  - Copy the whole content of the source

    strcpy (char* *dest* **, const char*** *src* **);**
  - Appending part of the source

    strncpy (char* *dest* **, const char*** *src,* ***int*** *size* **);**

    Where size is the number characters to be copied

d) String Compare:

- Compares the two string *string1* and *string2*.
- The string compare can have two forms, where the first one is to compare the whole content of the two strings and the other will compare only part of the two strings.

- Compare the whole content of the source

  strcmp (const char* *string1* **, const char*** *string2* **);**

- Compare part of the source

  strncmp (const char* *string1* **, const char*** *string2, **int** size* **);**

  Where size is the number characters to be compaired

- Both string compare functions returns three different values:
  - Returns **0** is the strings are equal
  - Returns **negative** value if the first is less than the second string
  - Returns **positive** value if the first is greater than the second string

Following example makes use of few of the above-mentioned functions:

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
char str1[10] = "Hello";
char str2[10] = "World";
char str3[10];
int len ;
int compare =strcmp(str1, str2);
cout<<”string1 compare with string2 is ”<<compare<<endl;
strcpy( str3, str1); // copy str1 into str3
cout << "strcpy( str3, str1) : " << str3 << endl;
strcat( str1, str2); // concatenates str1 and str2
cout << "strcat( str1, str2): " << str1 << endl;
len = strlen(str1); // total lenghth of str1 after concatenation
cout << "strlen(str1) : " << len << endl;
return 0;}
```

When the above code is compiled and executed, it produces result something as follows:

```
string1 compare with string2 is -1
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

Chapter Three
Pointers

3. Background

3.1. Variables and Memory

When you declare a variable, the computer associates the variable name with a particular location in memory and stores a value there. When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

C++ allows us to perform either one of these steps independently on a variable with the & and * operators:

1. &x evaluates to the address of x in memory.
2. *(&x) takes the address of x and dereferences it – it retrieves the value at that location in memory. *(&x) thus evaluates to the same thing as x.

Motivating Pointers

Memory addresses, or pointers, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. Just a taste of what we'll be able to do with pointers:

- ✓ More flexible pass-by-reference
- ✓ Manipulate complex data structures efficiently, even if their data is scattered in different memory locations

3.1.1. The Nature of Pointers

We have already seen (in fundamentals programming I) how variables are memory cells that we can access by an identifier. But these variables are stored in concrete places of the computer memory. For our programs, the computer memory is only a succession of 1 byte cells (the minimum size for a datum), each one with a unique address. A pointer is a variable which stores the address of another variable. The only difference between pointer variable and regular variable is the data they hold.

- ➢ There are two pointer operators in C++:

  &→ the address of operator

  * →the dereference operator

Whenever you see the & used with pointers, think of the words “address of.” The & operator always produces the memory address of whatever it precedes. The * operator, when used with pointers, either declares a pointer or dereferences the pointer’s value. The dereference operator can be literally translated to ***"value pointed by"*** .

A pointer is simply the address of an object in memory. Generally, objects can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to an object via a pointer to it, is called dereferencing the pointer. Pointer variables are defined to point to objects of a specific type so that when the pointer is dereferenced, a typed object is obtained.

At the moment in which we declare a variable this one must be stored in a concrete location in this succession of cells (the memory). We generally do not decide where the variable is to be placed - fortunately that is something automatically done by the compiler and the operating system on runtime, but once the operating system has assigned an address there are some cases in which we may be interested in knowing where the variable is stored.

This can be done by preceding the variable identifier by an ampersand sign (&), which literally means, "address of”. For example:

```
ptr= &var;
```

This would assign to variable ptr the address of variable var , since when preceding the name of the variable var with the ampersand (&) character we are no longer talking about the content of the variable, but about its address in memory.

We are going to suppose that var has been placed in the memory address 1776 and that we write the following:

```
var=25;
x=var;
ptr = &var;
```

The result will be the one shown in the following diagram:

We have assigned to **x** the content of variable **var** as we have done in many other occasions in previous sections, but to **ptr** we have assigned the address in memory where the operating system stores the value of **var**, that we have imagined that it was **1776** (it can be any address). The reason is that in the allocation of **ptr** we have preceded **var** with an *ampersand* (&) character.

The variable that stores the address of another variable (like **ptr** in the previous example) is what we call a **pointer.**

3.2. Pointer Syntax/Usage

3.2.1. Declaring Pointers

It is reserving a memory location for a pointer variable in the heap. The general scheme for declaring pointers is:

```
data_type *pointer_name; // Add "= initial_value " if applicable
```

Example: to declare a pointer variable called p_age, do the following:

*int * p_age;*

Whenever the dereference operator, *, appears in a variable declariation, the variable being declared is always a pointer variable.

3.2.2. Assigning *values to pointers*:

- p_age is an integer pointer. The type of a pointer is very important. p_age can point only to integer values, never to floating-point or other types.
- To assign p_age the address of a variable, do the following:

```
int age = 26;
int * p_age;
p_age = &age;
```

OR

```
int age = 26;
int * p_age = & age;
```

- Both ways are possible.
- If you wanted to print the value of age, do the following:

```
cout<<age;//prints the value of age
Or by using pointers you can do it as follows
cout<<*p_age;//dereferences p_age;
```

- The dereference operator produces a value that tells the pointer where to point. Without the *, (i.e cout<<p_age), a cout statement would print an address (the address of age). With the *, the cout prints the value at that address.
- You can assign a different value to age with the following statement:

age = 13; //assigns a new value to variable age

*p_age = 13 //assigns 13 as a value to the memory p_age points at.

N.B: the * appears before a pointer variable in only two places: when you declare a pointer variable and when you dereference a pointer variable (to find the data it points to).

- The following program is one you should study closely. It shows more about pointers and the pointer operators, & and *, than several pages of text could do.

int main()

```
{
intnum = 123; // a regular integer variable
int *p_num; //declares an integer pointer
cout<< "num is "<<num<<endl;
cout<< "the address of num is "<<&num<<endl;
p_num = &num;// puts address of num in p_num;
cout<< "*p_num is "<<*p_num<<endl; //prints value of num
cout<< "p_num is "<<p_num<<endl; //prints value of P_num(address of num)
getch();
}
```

3.2.3. Pointer to void

- Note that we can't assign the address of a float type variable to an integer pointer variable and similarly the address of an interger variable can not be stored in a float or character pointer.

```
flaot y;
int x;
int *ip;
float *fp;
ip = &y; //illegal statement
fp = &x; //illegal statement
```

- That means, if a variable type and pointer to type is same, then only we can assign the address of variable to pointer variable. And if both are different type then we can't assign the address of variable to pointer variable but this is also possible in C++ by declaring pointer variable as a void as follows:

```
void *p;
```

- Let us see an example:

```
void *p;
int x;
float y;
p = &x; //valid assignment
p = &y; //valid assignment
```

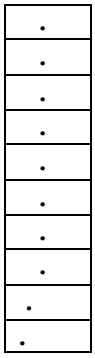
- The difficulty on void pointers is that, void pointers can not be de referenced. They are aimed only to store address and the dereference operator is not allowed for void pointers.

3.2.4. Arrays of Pointers

- If you have to reserve many pointers for many different values, you might want to declare an array of pointers.
- The following reserves an array of 10 integer pointer variables:

```
int *iptr[10]; //reserves an array of 10 integer pointers
```

- The above statement will create the following structure in RAM



iptr[4] = &age;// makes iptr[4] point to address of age.

3.3. Pointer and arrays

- The concept of array goes very bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing. For example, supposing these two declarations:

  int numbers [20];
  *int * p;*

- the following allocation would be valid:

  p = numbers;

- At this point **p** and **numbers** are equivalent and they have the same properties, with the only difference that we could assign another value to the pointer **p** whereas **numbers** will always point to the first of the 20 integer numbers of type int with which it was defined. So, unlike **p,** that is an ordinary *variable pointer*, **numbers** is a *constant pointer* (indeed that is an Array: a constant pointer). Therefore, although the previous expression was valid, the following allocation is not:

  numbers = p;

- Because **numbers** is an array (constant pointer), and no values can be assigned to constant identifiers.

- N.B: An array name is just a pointer, nothing more. The array name always points to the first element stored in the array. There for , we can have the following valid C++ code:

```
Int ara[5] = {10,20,30,40,50};
cout<< *(ara + 2); //prints ara[2];
```

- The expression *(ara+2) is not vague at all if you remember that an array name is just a pointer that always points to the array's first element. *(ara+2) takes the address stored in ara, adds 2 to the address, and dereferences that location.
- Consider the following character array:

  char name[] = "C++ Programming";

- What output do the following cout statements produce?

```
cout<<name[0];      // ____C__
cout<<*name;        // _____C__
cout<<*(name+3); //_________
cout<<*(name+0); //____C____
```

3.3.1. Pointer Advantage

- IT allows management of structures which are allocated memory dynamically.
- IT allows passing arrays and strings to functions more efficiently
- It makes possible to pass address of structure instead of entire structure to function.
- You can't change the value of an array, because you can't change constants. This explains why you can't assign an array a new value during a program's execution: eg: if Cname is array of characters then: Char Cname[20];

```
Cname = "Football"; //invalid array assignment;
Cname[]="football";//invalid array assignment;
```

OR

```
Int array[6];
array={1,2,3,4,5};//invalid array assignment
array[]={1,2,3,4,5};//invalid array assignment
array=1;//invalid
```

- Unlike arrays, you can change a pointer variable. By changing pointers, you can make them point to different values in memory. Have a look at the following code:

```
#include<iostream.h>
#...
int main()
{
```

```
        float v1 = 679.54;
        float v2 = 900.18;
        float * p_v;
        p_v = &v1;
        cout<< "\nthe first value is "<<*p_v;
        p_v = &v2;
        cout<< "\nthe second value is "<<*p_v;
        getch();
}
```

- You can use pointer notation and reference pointers as arrays with array notation. Study the following program carefully. It shows the inner workings of arrays and pointer notation.

```
void main()
{
clrscr();
int ctr;
int iara[5] = {10,20,30,40,50};
int *iptr;
iptr = iara; //makes iprt point to array's first element. Or iprt = &iara[0]
cout<< "using array subscripts:\n"
cout<< "iara\tiptr\n";
for(ctr=0;ctr<5;ctr++)
  {
cout<<iara[ctr]<< "\t"<<iptr[ctr]<< "\n";
  }
cout<< "\nUsing pointer notation\n";
for(ctr=0;ctr<5;ctr++)
  {
cout<< *(iara+ctr) << "\t" << *(iptr+ctr)<< "\n";
  }
getch();
}
```

- Suppose that you want to store a persons name and print it. Rather than using arrays, you can use a character pointer. The following program does just that.

```
intmain()
{
char *c = "MeseretBelete";
cout<< "your name is : "<<c;
```

```
}
```

- Suppose that you must change a string pointed to by a character pointer, if the persons name in the above code is changed to MeseterAlemu: look at the following code:

```
int main()
{
char *c = "MeseretBelete";
cout<< "youe name is : "<<c;
c = "MeseretAlemu";
cout<< "\nnew person name is : "<<c;
getch();
}
```

- If c were a character array, you could never assign it directly because an array name can't be changed.

3.4. Pointer Arithmetic

- To conduct arithmetical operations on pointers is a little different than to conduct them on other integer data types. To begin, only *addition* and *subtraction* operations are allowed to be conducted, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point to.

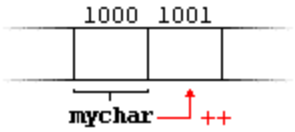
- When we saw the different data types that exist, we saw that some occupy more or less space than others in the memory. For example, in the case of integer numbers, *char* occupies 1 byte,*short* occupies 2 bytes and *long* occupies 4.
- Let's suppose that we have 3 pointers:

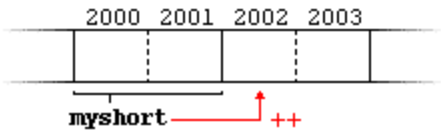
```
char *mychar;
short *myshort;
long *mylong;
```

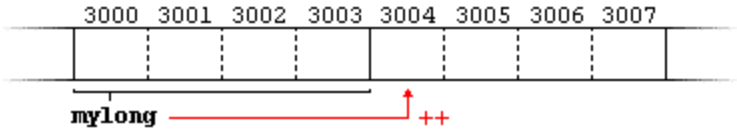
- And that we know that they point to memory locations **1000** ,**2000** and **3000** respectively. So if we write:

```
mychar++;
myshort++;
mylong++;
```

- mychar , as you may expect, would contain the value 1001 . Nevertheless, myshort would contain the value 2002 , and mylong would contain 3004 . The reason is that when adding 1 to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in *bytes* of the *type* pointed is added to the pointer.







- This is applicable both when adding and subtracting any number to a pointer.
- It is important to warn you that both increase (++) and decrease (--) operators have a greater priority than the reference operator asterisk (*), therefore the following expressions may lead to confusion:

```
*p++;
*p++ = *q++;
```

- The first one is equivalent to *(p++) and what it does is to increase **p** (the address where it points to– not the value that contains).
- The second, because both increase operators (++) are after the expressions to be evaluated and not before, first the value of *q is assigned to *p and then they are both q and p increased by one. It is equivalent to:

```
*p=*q;
p++;
q++;
```

- Now let us have a look at a code that shows increments through an integer array:

```
int main()
{
intiara[] = {10,20,30,40,50};
int * ip = iara;
cout<<*ip<<endl;
ip++;
cout<<*ip<<endl;
ip++;
cout<<*ip<<endl;
ip++;
cout<<*ip<<endl;
}
```

3.5. Pointer and String

- If you declare a character table with 5 rows and 20 columns, each row would contain the same number of characters. You can define the table with the following statement.
- *char names[5][20] ={{"George"},{"Mesfin"},{"John"},{"Kim"},{"Barbara"}};*

- The above statement will create the following table in memory:

G	E	O	R	g	e	\0													
M	E	S	F	i	n	\0													
J	O	H	N	\0															
K	I	M	\0																
B	A	R	B	a	r	a	\0												

- Notice that much of the table is waster space. Each row takes 20 characters, even though the data in each row takes far fewer characters.
- To fix the memory-wasting problem of fully justified tables, you should declare a single-dimensional array of character pointers. Each pointer points to a string in memory and the strings do not have to be the same length.
- Here is the definition for such an array:

  *char *name [5] = {{"George"},{"Mesfin"},{"John"}, {"Kim"},{"Barbara"}};*
- This array is a single-dimension array. The asterisk before names makes this array as an array of pointers. Each string takes only as much memory as is needed by the string and its terminating zero. At this time, we will have this structure in memory:

  To print the first string, we should use:
  *cout<<*names; //prints George.*
  To print the second use:
  *cout<< *(names+1); //prints Mesfin*
- Whenever you dereference any pointer element with the * dereferencing operator, you access one of the strings in the array.

3.6. Pointer to pointer:

- As the memory address where integer, float or character is stored in can be stored into a pointer variable, the address of a pointer can also be stored in another pointer. This pointer is said to be pointer to a pointer.
- An array of pointer is conceptually same as pointer to pointer type. The pointer to pointer type is declared as follows:

  *Data_type ** pointer_name;*

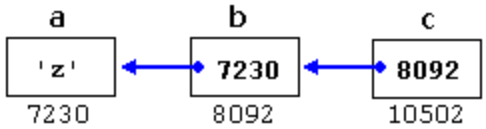
- Note that the asterisk is double here.

  *int **p; //p is a pointer which holds the address of another pointer.*

  E.g.:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

- This, supposing the randomly chosen memory locations of **7230 , 8092** and **10502** , could be described thus:



(inside the cells there is the content of the variable; under the cells its location)

- Have a look at the following code:

```
int main()
{
int data;
int *p1;
int **p2;
data = 15;
cout<< "data = "<<data<<endl;
p1 = &data;
p2 = &p1;
cout<< "data through p1 = "<<*p1<<endl;
cout<< "data through p2 = "<< **p2<<endl;
}
```

3.7. Higher-order Functions

The functions we have seen so far have accepted only data as parameters. Since functions can contain Conditional statements and loops, they can do different things based on the data they receive. The code Within each function, however, is fixed at compile time. Consider the following function:

```
int evaluate(int x, int y) {
return x + y;
}
```

The call

evaluate(10, 2)

evaluates to 12. Since the evaluate function returns the sum of its two parameters. The call

evaluate(-4, 6);

returns 2. The function returns a different result this time because we passed different parameters to it . Evaluate function in a sense behaves differently depending on the arguments passed by its caller ;however, it always adds its two parameters. There is no way we can call evaluate and expect it to **multiply** its two parameters instead. The evaluate function is hard-coded to perform addition .What if we wanted the evaluate function to be able to perform different arithmetic operations at different times during a program's execution? Unfortunately, evaluate as it currently is written cannot adapt to perform a different arithmetic operation. The good news is that we can rewrite evaluate so that it can flexibly adapt to a caller's changing arithmetic needs.

C++ allows programmers to pass functions as parameters to other functions. A function even may return a function as a result. A function that accepts one or more functions as parameters or returns a function as a result is known as a higher-order function. As we will see, higher-order functions open up new programming possibilities enabling us to customize the behavior of a function by plugging into it different functions to achieve different effects.

C++ achieves higher-order functions via function pointers. During a program's execution, the compiled machine language code for a function must reside in the computer's memory for the program toto invoke the function. That means every function has a memory address just as each variable has its own specific memory address. A pointer to a function holds the starting address for the compiled code of a particular function.

The following code snippet provides an example of a higher-order function in action.

```
#include <iostream>
Using namespace std;
Int add(intx, inty) {
returnx + y;
}
Int multiply(intx, inty) {::
returnx * y;
}
```

```
Int evaluate(int(*f)(int, int), intx, inty) {
returnf(x, y);
}
intmain() {
cout<< add(2, 3) <<'\n';
cout<< multiply(2, 3) <<'\n';
cout<< evaluate(&add, 2, 3) <<'\n';
cout<< evaluate(&multiply, 2, 3) <<'\n';
}
```

int(*f)(int, int)
represents a function pointer. The parameter's name is f, and f is a pointer to a function that accepts two integer parameters and returns an integer result. In Listing 10.20 (arithmeticeval.cpp), the first parameter a caller must pass to evaluate is the address of a function with a prototype that matches the formal parameter specified in **evaluate's** definition. Both **add** and **multiply** functions qualify because they accept two integer parameters and return an integer result. The expression

evaluate(&add, 2, 3)

passes the address of the add function to evaluate. In the body of the evaluate function we see that evaluate invokes the function specified by its first parameter (f), passing to this function its second (x)and third (y) parameters as the two parameters that function expects. The net effect, therefore, of the call evaluate(&add, 2, 3)is the same as

add(2, 3)

C++ has a somewhat relaxed syntax for function pointers that cannot be applied to pointers to data.When invoking evaluate in source code we may omit the ampersand in front of the function argument,as is

cout<< evaluate(add, 2, 3) <<'\n';

Since the compiler knows that add is a function and since parentheses do not follow the word add, the compiler deduces that this is not a call to add but rather the address of the add function.

NB: When the name of a previously declared function appears by itself within C++source code it represents a pointer to that function.

Function pointers are not restricted to function parameters. Given the definition of add above, the following code fragment is legal C++:

```
// Declare func to be a pointer to a function that accepts
// two integer parameters and returns an integer result
int(*func)(int, int);
// Assign add to func
func = add;
// Call add through func
cout<<func(7, 2) <<'\n';
```

This code fragment as part of a complete C++ program would print 9.

Another Example

```
// pointer to functions
#include <iostream.h>
int addition (int a, int b)
{
    return (a+b);
}

int subtraction (int a, int b)
{
    return (a-b);
}

   int (*minus)(int,int) = subtraction;

int operation (int x, int y, int (*functocall)(int,int))
{
```

```
    int g;
    g = (*functocall)(x,y);
    return (g);
}

int main ()
{
     Int m,n;
    m = operation (7, 5, addition);
     n = operation (20, m, minus);
     cout<<n;
return 0;
}
```

The output of the above program is 8. In the example, minusis a global pointer to a function that has two parameters of type int, it is immediately assigned to point to the function subtraction, all in a single line:

```
int (* minus)(int,int) = subtraction;
```

3.8. Dynamic memory:

- Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them to be fixed before the execution of the program. But, What if we need a variable amount of memory that can only be determined during the program execution (runtime)? For example, in case that we need a user input to determine the necessary amount of space. The answer is *dynamic memory,* for which C++ integrates the operators *new* and *delete.*
- Pointers are useful for creating **dynamic** objects during program execution. Unlike normal (global and local) objects which are allocated storage on the runtime stack, a dynamic object is allocated memory from a different storage area called the **heap**. Dynamic objects do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.

a) The New Operator

- In C++ *new* operator can create space dynamically i.e at run time, and similarly *delete* operator is also available which releases the memory taken by a variable and return memory to the operating system.
- When the space is created for a variable at compile time this approach is called static. If space is created at run time for a variable, this approach is called dynamic. See the following two lines:

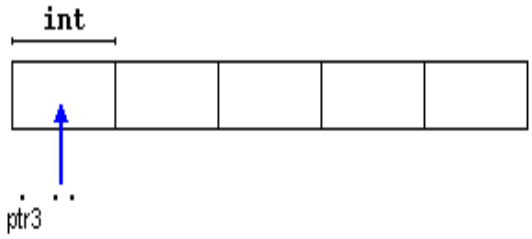
```
int a[10];//creation of static array
int *a;
```

```
a = new int[10];//creation of dynamic array
```

- Lets have another example:

```
int * ptr3;
ptr3 = new int [5];
```

- In this case, the operating system has assigned space for 5 elements of type **int**in the heap and it has returned a pointer to its beginning that has been assigned to **ptr3** . Therefore, now, **ptr3** points to a valid block of memory with space for 5 **int**elements.



- You could ask whatis the difference between declaring a normal array and assigning memory to a pointer as we have just done. The most important one is that the size of an array must be a constant value, which limits its size to what we decide at the moment of designing the program before its execution, whereas the dynamic memory allocation allows assigning memory during the execution of the program using any variable, constant or combination of both as size.

- The dynamic memory is generally managed by the operating system, and in the multi-task interfaces can be shared between several applications, so there is a possibility that the memory exhausts. If this happens and the operating system cannot assign the memory that we request with the operator **new**, a null pointer will be returned. For that reason it is recommendable to always verify if after a call to instruction **new** the returned pointer is null:

```
int * ptr3;
ptr3 = new int [5];
if (ptr3 == NULL) {
    // error assigning memory. Take measures.
  };
```

- if ptr3 is **NULL**, it means that there is no enough memory location in the heap to be given for ptr3.

b) *Operator delete*

- Since the necessity of dynamic memory is usually limited to concrete moments within a program, once this one is no longer needed it shall be freed so that it become available for future requests of dynamic memory. For this exists the operator **delete** , whose form is:

deletepointer ***;***

or

delete [] pointer ***;***

- The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for multiple elements (arrays).
- In most compilers both expressions are equivalent and can be used without distinction, although indeed they are two different operators and so must be considered for operator overloading.
- In the following simple example, a program that memorizes numbers, does not have a limited amount of numbers that can be introduced, thanks to the concept and power of pointer that we request to the system as much space as it is necessary to store all the numbers that the user wishes to introduce.

```
#include <iostream.h>
#include <stdlib.h>
#include<conio.h>
int main ()
{
char input [100];
inti,n;
long * num;// total = 0;
cout<< "How many numbers do you want to type in? ";
cin.getline (input,100);
i=atoi (input);
num= new long[i];
if (num == NULL)
 {
cout<<"\nno enough memory!";
getch();
exit (1);
      }
for (n=0; n<i; n++)
      {
cout<< "Enter number: ";
cin.getline (input,100);
num[n]=atol(input);
      }
cout<< "You have entered: ";
for (n=0; n<i; n++)
cout<<num[n] << ", ";
delete[] num;
getch();
```

```
return 0;
}
```

- **NULL** is a constant value defined in C++ libraries specially designed to indicate null pointers. In case that this constant is not defined you can do it yourself by defining it to 0:

Chapter Four
Structures

What is a Structure?

Structure is a collection of variables under a **single name**. Variables can be of any type: int, float, char etc. The main difference between **structure** and **array** is that arrays are collections of the same data type and structure is a collection of variables under a **single name**.

Note:

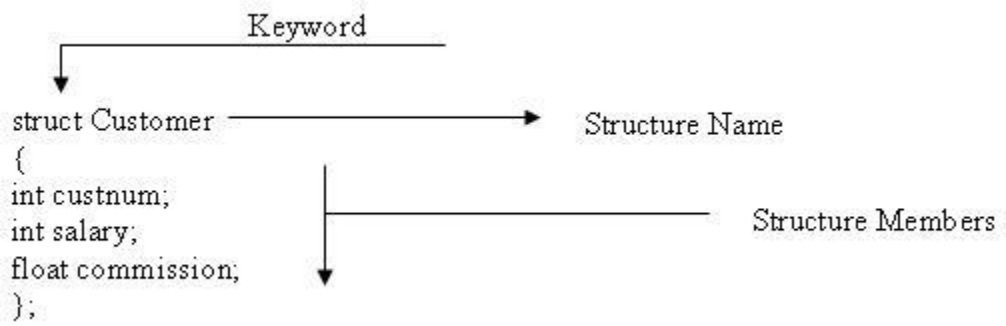
- A Structure is a collection of related data items, possibly of different types.
- A structure type in C++ is called **struct**.
- A structure is **heterogeneous** (composed of data of different types.)
- In contrast, array is **homogeneous** since it can contain only data of the same type.
- Structures hold data that belong **together**.
- Examples:
  - Student record: student id, name, major, gender, start year , ..
  - Bank account: account number, name, currency, balance ,…
  - Address book: name, address, telephone number, …
- In database applications, structures are called **records.**
- Individual components of a struct type are called **members (or fields).**
- Members can be of **different types** (simple, array or struct).
- A struct is named as a whole while individual members are named using field identifiers.
- Complex data structures can be formed by defining **arrays of structs.**

Declaring a Structure:

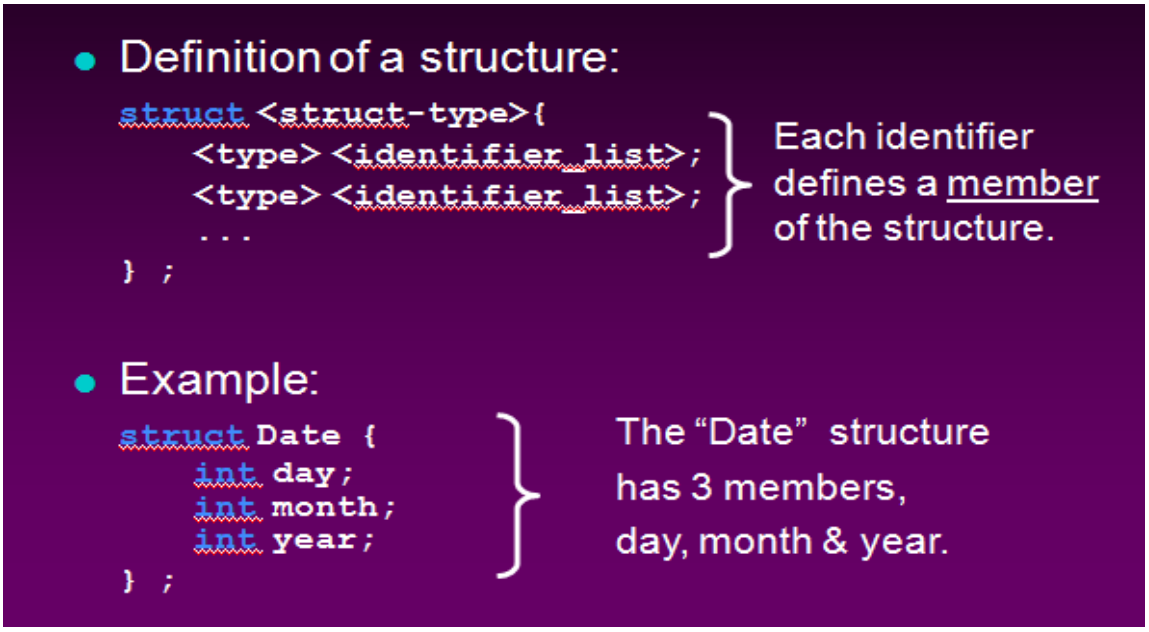
The structure is declared by using the keyword **struct** followed by structure name, also called a **tag**. Then the structure members (**variables)** are defined with their type and variable names inside the open and close braces { and }. Finally, the closed braces end with a semicolon denoted as ; following the statement. The above structure declaration is also called a Structure Specifier.

Example:

Three variables: *custnum*of type int, *salary* of type int, *commission* of type float are structure members and the structure name is **Custome**r. This structure is declared as follows:

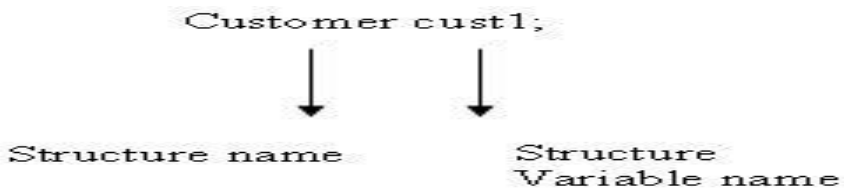


In the above example, it is seen that variables of different types such as int and float are grouped in a **single structure** name Customer. Arrays behave in the same way, **declaring structures** does not mean that memory is allocated. Structure declaration gives a skeleton or template for the structure. After declaring the structure, the next step is to **define** a structure variable.



Declaration of Structure Variable?

This is similar to variable declaration. For variable declaration, data type is defined followed by variable name. For **structure variable declaration**, the data type is the **name of the structure** followed by the structure variable name. In the above example, structure variable **cust1** is defined as:



Here are examples of declaring structure variable

```
#include<iostream.h>
struct  customer
{
int custnum;
char name[10];
int phonenum;
}cust1,cust2,cust3;//structure variable declaration
int main()
{
….
}
```

```
#include<iostream.h>
struct  customer
{
int custnum;
char name[10];
int phonenum;
};
int main()
{
struct  customer  cust1,cust2,cust3;  //structurevariable declaration
}
```

What happens when this is defined? When structure is defined, it **allocates or reserves space** in memory. The memory space allocated will be cumulative of all defined structure members. In the above example, there are 3 structure members: custnum, name and phonenum. If integer space allocated by a system is 2 bytes and char one bytes the above would allocate 2bytes for custnum, 2 bytes for phonenum and 1byte for name.

How to access structure members in C++?

- ✓ Use the dot operator to access or **initialize members** of structures.
- ✓ More importantly, you usually do not even know the contents of the structure variables. Generally, the user enters data to be stored in structures, or you read them from a **disk file**.
- ✓ A better approach to initializing structures is to use the dot operator (.) The dot operator is one way to initialize individual members of a structure variable in the body of your program. With the dot operator, you can treat each structure member almost as if it were a regular non structure variable.
- ✓ The General syntax to access members of a structure variable would be:

structurevariablename.membername

- ✓ A structure **variable name** must always precede the dot operator, and a **member name** must always appear after the dot operator. Using the dot operator is easy, as the following examples show.

Here are some examples

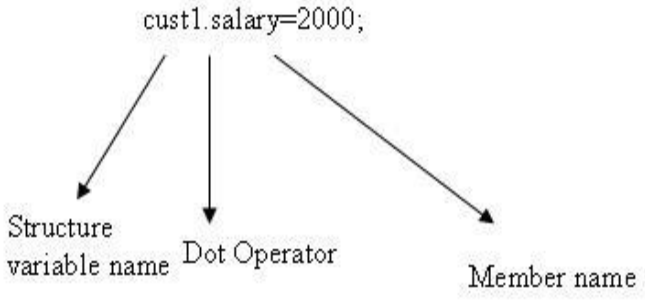
#include<iostream.h>
struct customer{
int custnum;
char name[10];
int phonenum;}**cust1;**
int main(){
cout<<"Enter customer number"<<endl;
cin>>cust1.custnum;
cout<<"Enter customer name"<<endl;
cin>>cust1.name;
cout<<"Enter customer phone number"<<endl;
cin>>cust1.phonenum;
cout<<"Customer Number:"<<**cust1.custnum**;
cout<<"\nCustomer Name:"<<**cust1.name**;
cout<<"\nCustomerPhone:"<<**cust1.phonenum**;}
	#include<iostream.h>
	struct customer{
	int custnum;
	char name[10];
	int phonenum; **};**
	int main(){
	struct customer cust1;
	cout<<"Enter customer number"<<endl;
	cin>>cust1.custnum;
	cout<<"Enter customer name"<<endl;
	cin>>cust1.name;
	cout<<"Enter customer phone number"<<endl;
	cin>>cust1.phonenum;
	cout<<"Customer Number:"<<**cust1.custnum**;
	cout<<"\nCustomer Name:"<<**cust1.name**;
	cout<<"\nCustomerPhone:"<<**cust1.phonenum**;}

Initializing structure members

You cannot initialize individual members because they are **not variables**. You can assign only values to variables. The braces must enclose the data you initialize in the structure variables, just as they enclose data when you initialize arrays. This method of initializing structure variables becomes tedious when there are several structure variables (as there usually are). Putting the data in several variables, each set of data enclosed in braces, becomes messy and takes too much space in your code. You can initialize members when you **declare a structure**, or you can initialize a structure in the body of the program.

For example:

A programmer wants to assign 2000 for the structure member *salary* in the above example of structure *Customer* with structure variable ***cust1*** this is written as:



For example

Initializing members at declaration	Initializing members at the body of the program

#include<iostream.h>
struct employee
{
char Emp_id[10];
char Name[20];
float Salary;
}**emp1={"DMU/001","Zerihun",2808};**
int main()
{
cout<<"Your id:"<<emp1.Emp_id<<endl;
cout<<"Your name:"<<emp1.Name<<endl;
cout<<"Your salay"<<emp1.Salary<<endl;
}
	#include<iostream.h>
	#include<string.h>
	struct employee
	{
	char Emp_id[10];
	char Name[20];
	float Salary;
	}emp1;
	int main()
	{
	strcpy(emp1.Emp_id,"DMU/001/");
	strcpy(emp1.Name,"Zerihun");
	emp1.Salary=2808;
	cout<<"Your id:"<<emp1.Emp_id<<endl;
	cout<<"Your name:"<<emp1.Name<<endl;
	cout<<"Your salay:"<<emp1.Salary<<endl;
	}

Arrays of Structures

- ✓ Arrays of structures are good for storing a complete employee file, inventory file, or any other set of data that fits in the structure format.
- ✓ Consider the following structure declaration:

```
struct Company
    {
int employees;
int registers;
double sales;
}store[1000];
```

- ✓ In one quick declaration, this code creates 1,000 ***store*** structures with the definition of the ***Company*** structure, each one containing three members.
- ✓ **NB.** Be sure that your computer does not run out of memory when you create a large number of structures. Arrays of structures quickly consume valuable information.
- ✓ You can also define the array of structures after the declaration of the structure.

```
struct Company
    {
    int employees;
    int registers;
    double sales;
    }; // no structure variables defined yet
```

```
#include<iostream.h>
void main()
{
struct Company store[1000]; //the variable store is array of the structure Company
        ...
}
```

Referencing the array structure

- ✓ The ***dot operator (.)*** works the same way for structure array element as it does for regular variables.
- ✓ Look the following example

```
#include<iostream.h>
struct student
{
int id;
float mark;
char name[15];
}stud[2];
int main()
{
for(int i=0;i<=1;i++)
{
cout<<"Enter the id, name and mark of student"<<i+1<<endl;
cin>>stud[i].id>>stud[i].name>>stud[i].mark;
}
for(int i=0;i<=1;i++)
{
cout<<"Id of Student "<<i+1<<"="<<stud[i].id<<endl;
cout<<"Name of Student "<<i+1<<"="<<stud[i].name<<endl;
cout<<"Mark of student "<<i+1<<"="<<stud[i].mark<<endl;
}
}
```

Pointers to structures

Like any other type, structures can be pointed by pointers. The rules are the same as for any fundamental data type: The pointer must be declared as a pointer to the structure:

```
struct Book {
char title[50];
int year;
};
Book b1;
Book * pb1;
```

Here **b1** an object of struct type **Book** and **pb1** is a pointer to point to objects of struct type Book. So, the following, as with fundamental types, would also be valid: pb1 = &b1;

Ok, we will now go with another example that will serve to introduce a new operator

// pointers to structures

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

```
#include <stdlib.h>
struct Book{
char title[50];
int year;};
int main (){
char buffer[50];
Book b1;
Book * pb1;
Pb1 = &b1;
cout << "Enter title: ";
cin.getline (pb1->title,50);
cout << "Enter year: ";
cin.getline (buffer,50);
pb1->year = atoi (buffer);
cout << "\nYou have entered:\n";
cout << pb1->title;
cout << " (" << pb1->year << ")\n";
return 0;}
```

The previous code includes an important introduction: operator→. This is a reference operator that is used exclusively with pointers to structures and pointers to classes. It allows us not to have to use parenthesis on each reference to a structure member. In the example we used:

Pb1->title

that could be translated to:

(*pb1).title

both expressions pb1->title and (*pb1).title are valid and mean that we are evaluating the element title of the structure pointed by pb1. You must distinguish it clearly from:

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

```
#include <iostream>
#include <cstring>
using namespace std;
void printBook( struct Books book );
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
};
int main( )
{
struct Books Book1; // Declare Book1 of type Book
struct Books Book2; // Declare Book2 of type Book
// book 1 specification
strcpy( Book1.title, "Learn C++ Programming");
strcpy( Book1.author, "Chand Miyan");
strcpy( Book1.subject, "C++ Programming");
```

OUTPUT

Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming

```
Book1.book_id = 6495407;
// book 2 specification
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Yakit Singha");
strcpy( Book2.subject, "Telecom");
Book2.book_id = 6495700;
// Print Book1 info
printBook( Book1 );
// Print Book2 info
printBook( Book2 );
return 0;
}
void printBook( struct Books book )
{
cout << "Book title : " << book.title <<endl;
cout << "Book author : " << book.author <<endl;
cout << "Book subject : " << book.subject <<endl;
cout << "Book id : " << book.book_id <<endl;
}
```

Nesting structures

Structures can also be nested so that a valid element of a structure can also be another structure.

```
struct Book {
char title[50];
int year;
};
struct student{
char name[50];
char email[50];
Book favourite_Book;
} S1, S2;
student * pstud = &S1;
```

Therefore, after the previous declaration we could use the following expressions:

```
S1.name
S2.favourite_Book.title
S1.favourite_Book.year
pstud->favourite_Book.year
(*pstud). favourite_Book.year
```

NB the last three expressions are equivalent.

```
(*pstud). favourite_Book.year
```

NB the last three expressions are equivalent.

Example 2:

```
#include<iostream.h>
struct Distance {
int feet;
float inches;};
struct Room //rectangular area
{
```

```
Distance length; //length of rectangle
Distance width; //width of rectangle
};
int main(){
Room dining; //define a room
dining.length.feet = 2; //assign values to room
dining.length.inches = 3.1;
dining.width.feet = 3;
dining.width.inches = 2.0;
//convert length & width
float l = dining.length.feet + dining.length.inches;
float w = dining.width.feet + dining.width.inches;
//find area and display it
cout <<"Dining room area is " << l * w;
return 0;}
```

expression	can be read as
*x	pointed to by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed to by x
(*x).y	member y of object pointed to by x (equivalent to the previous one)
x[0]	first object pointed to by x
x[1]	second object pointed to by x
x[n]	(n+1)th object pointed to by x

Chapter Five

5. Streams and External Files

- **Introduction**
  - The data created by the user and assigned to variables with an assignment statement is sufficient for some applications. With large volume of data most real-world applications use a better way of storing that data. For this, disk files offer the solution.
  - When working with disk files, C++ does not have to access much RAM because C++ reads data from your disk drive and processes the data only parts at a time.
- **Stream**
  - Stream is a general name given to *flow of data*. In C++, there are different types of streams. Each stream is associated with a particular class, which contains member function and definition for dealing with file. Let's have a look at the figure:

  - According to the above hierarchy, the class iostream is derived from the two classes' istream and ostream and both istream and ostream are derived from ios. Similarly the class fstream is derived from iostream. Generally two main header files are used iostream.h and fstream.h. The classes used for input and output to the video display and key board are declared in the header file iostream.h and the classes used for disk file input output are declared in fstream.h.
  - Note that when we include the header file fstream.h in our program then there is no need to include iostream.h header file. Because all the classes which are in fstream.h they are derived from classes which are in iostream.h therefore, we can use all the functions of iostream class.

- **Operation with File**
  - First we will see how files are opened and closed. A file object can be defined by following class ifstream, ofstream, fstream, all these are defined in fstream.h header file.

- if a file object is declared by *ifstream* class, then that object can be used for *reading* from a file.
- if a file object is declared by *ofstream* class, then that object can be used for *writing* onto a file.
- If a file object is declared by *fstream* class then, that object can be used for both *reading from* and *writing to* a file.

- **Types of Disk File Access**
  - Your program can access files either in *sequential* manner or *random* manner. The access mode of a file determines how one can read, write, change, add and delete data from a file.
  - A sequential file has to be accessed in the same order as the file was written. This is analogues to cassette tapes: you play music in the same order as it was recorded.
  - Unlike the sequential files, you can have random-access to files in any order you want. Think of data in a random-access file as being similar to songs on compact disc (CD): you can go directly to any song you want to play without having to play or fast-forward through the other songs.
  - SYNTAX:First the heder fileis fstream . **#include<iostream.h>;**
  - -Declaration either if or of stream file object name; **ifstream/ofstream i;**
  - /Open file ether open or constracter method. **i.open(file name ::ios::in/out);**
  - /Read when declaration is ifstream use >> **i>>x; cout<<x<<endl;**
  - /Write when declaration is ofstream use << **i<<(Dessie Tarekegn);**
  - /Close all data are closed using **i. close ();**

- ***Sequential File Concepts***
  - You can perform three operations on sequential disk files. You can *create* disk files, *add* to disk files, and *read* from disk files.

- ***Opening* and Closing Sequential Files**
  - When you open a disk file, you only have to inform C++, the file name and what you want to do with it. C++ and your operating system work together to make sure that the disk is ready, and they create an entry in your file directory for the filename (if you are creating from the program, and updates the file directory to reflect the file's new size.
  - You can use either of the two methods to open a file in C++:
    - using a ***Constructor*** or
    - using the ***open function***
  - The following C++ statement will create an object with the name ***fout*** of ofstream class and this object will be associated with file name "hello.txt".
  - **using constractor method both declareation and file name written in the same line** ***ofstream fout ("d:\\hello.txt");***
  - This statement uses the constructor method.

- The following C++ statement will create an object with the name ***fout*** of ofstream class and this object will be associated with file name "hello.txt".

  ofstream fout;declaration

  fout.open("d:\\hello.txt"); using open

- If you open a file for writing (out access mode), C++ creates the file. If a file by that name already exists, *C++ overwrite* the old file with no warning. You must be careful when opening files not to overwrite existing data that you want.
- If an error occurs during opening of a file, C++ does not create a valid file pointer (file object). Instead, C++ creates a file pointer (object) equal to zero. For example if you open a file for output, but use an invalid disk name, C++ can't open the file and therefore makes the file object equal to zero.
- You can also determine the file access mode when creating a file in C++. If you want to use the open function to open a file then the syntax is:

  fileobject.open(filename,accessmode);

  - File name is a string containing a valid file name for your computer.
  - Access mode is the sought operation to be taken on the file and must be one of the values in the following table.

Mode	**Description**
app	Opens file for appending
ate	Open and Seeks to the end of file while opening the file
in	Opens the file for reading
out	Opens the file for writing
binary	Opens the file in binary mode
nocreate	If file doesn't exist, don't create it
noreplace	*Don't delete the file(open fails if file exists unless you specify app or ate)*
trunc	Truncate file to zero length if it already exists(default if file exist and app or ate is not specified)

- You should always check for the successful opening of a file before starting file manipulation on it. You use the fail() function to do the task.
- Lets have an example here:

```
ifstream indata;
indata.open("c:\\myfile.txt",ios::in);
if(indata.fail())
{
    //error description here
}
```

- In this case, the open operation will fail (i.e the fail function will return true), if there is no file named myfile.txt in the directory C:\
- After you are done with your file manipulations, you should use the close function to release any resources that were consumed by the file operation. Here is an example

  indata.close();
- The above close() statement will terminate the relationship between the ifstream object indata and the file name "c:\myfile.txt", hence releasing any resource needed by the system.

- **Writing to a sequential File**
- The most common file I/O functions are
  - get()
  - and put()
- You can also use the output redirection operator (<<) to write to a file.
- The following program creates a file called names.txt in C:\ and saves the name of five persons in it:

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
ofstream fp;
int main()
{
        fp.open("c:\\names.txt" ,ios::out);
        if(fp.fail())
        {
        cerr<< "\nError opening file";
        getch();
        exit(1);
        }
        fp<< "Abebe Alemu"<<endl;
        fp<< "Lemelem Berhanu"<<endl;
        fp<< "Tesfaye Mulugeta"<<endl;
        fp<< "Mahlet Kebede"<<endl;
        fp<< "Assefa Bogale"<<endl;
        fp.close();}//end main
```

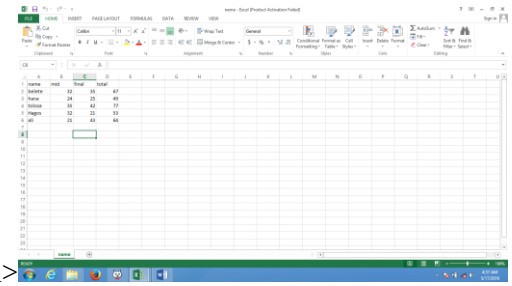
- **Writing characters to sequential files**
- A character can be written onto a file using the *put()* function. See the following code:

```
#include<fstream.h>
#include<stdlib.h>// for exit() function
int main()
{
char c;
ofstream outfile;
outfile.open("d:\\name.txt",ios::out);
        if(outfile.fail())
        {
        cerr<< "\nError opening test.txt";
```

```
        getch();
        exit(1);
        }
        for(int i=1;i<=15;i++)
        {
        cout<< "\nEnter a character : ";
        cin>>c;
        outfile.put(c);
        }
        outfile.close();
}//end main
```

- The above program reads 15 characters and stores in file name.txt. You can easily add data to an existing file, or create new files, by opening the file in append access mode. Files you open for append access mode (using ios::app) do not have to exist. If the file exists, C++ appends data to the end of the file (as it done when you open a file for write access). If the file names.txt *does not exist*, C++ creates it and stores the characters to the file.
- Basically, you have to change only the open() function's access mode to turn a file-creation program into a file-appending program.

  Example: the following C++ program is reads name, mid and final result of 5 students from the user. Then it calculates total result. Finally, it writes the result or data into **name.csv** file.



name.csv

```
#include<iostream.h>
#include<fstream>
using namespace std;
int main(){
char fname[30],name[5][30];
float mid[5],final[5],total[5];
int i;
ofstream f;
cout<<"enter file name\n";
cin>>fname;
f.open(fname,ios::out);
```

```
cout<<"enter name,mid and final result of 5 students\n";
for(i=0;i<5;i++){
cout<<"<enter name of student"<<i+1<<":";
cin>>name[i];
cout<<"<enter mid of student"<<i+1<<":";
cin>>mid[i];
cout<<"<enter final of student"<<i+1<<":";
cin>>final[i];
total[i]=mid[i]+final[i];}
f<<"name,mid,final,total\n";//comma is used to write data into each cell
for(i=0;i<5;i++)
f<<name[i]<<","<<mid[i]<<","<<final[i]<<","<<total[i]<<endl;
f.close();}
```

- **Reading from a File**
- Files you open for read access (using ios::in) *must exist already*, or C++ gives you an error message. You can't read a file that does not exist. Open () returns zero if the file does not exist when you open it for read access. Another event happens when you read files. Eventually, you read all the data. Subsequently reading produces error because there is *no more* data to read. C++ provides a solution to the end-of-file occurrence.
- If you attempt to read a file that you have completely read the data from, C++ returns the value zero. To find the end-of-file condition, be sure to check for zero when reading information from files.
  - The following code is count number of characters saved in hh.txt file name.

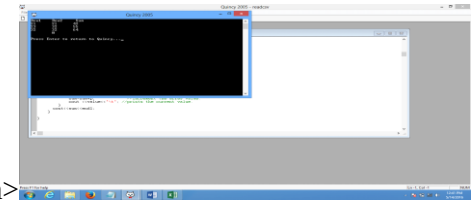
```
 #include<fstream.h>
#include<iostream.h>
   int main(){
     ifstream fin;
  fin.open("d:\\hh.txt");
   if(outfile.fail()){
  cerr<< "\nError opening test.txt";
        getch();
        exit(1);}
      char ch;
     int count=0;
  while(!fin.eof()){
    fin.get(ch);
   count++; }
cout << "Number of characters in file is " << count;
```

```
        fin.close();
    return 0;}
```

2. Write a c++ program that reads the following List.CSV file, calculate the sum of numbers in each row and print the result as follows.

List.CSV

21	21
23	32
32	32



```
#include<iostream>
#include<fstream>
#include<string>

#include<cstdlib>

#include<sstream>

int a[50],p;

using namespace std;

int main(){

    ifstream file;

    file.open("e:\\list.csv",ios::in);

    string row, value;

            int sum=0;

            cout<<"Row1\tRow2\t Sum\n";

    while(file.good())

    {//checks whether file is in good condition  {

            sum=0;

        getline(file, row); //gets the row

        istringstream tmp(row); //saves the row as the string

        while(tmp.good())   {

            getline(tmp, value, ','); //extracts the comma for the values

            p=atoi(value.c_str());
```

```
                    sum=sum+p;          //increment the array value.
        cout <<value<<"\t"; //prints the current value.
    }
        cout<<sum<<endl;
  }}
```

- **Reading characters from a sequential file**
- You can read a characters from a file using *get()* function. The following program asks for a file name and displays *each character* of the file to the screen. NB. A space among characters is considered as a character and hence, the exact replica of the file will be shown in the screen.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
int main()
{
  char c,filename[15];
  ifstream indata;
  cout<<"\nEnter the file name : ";
  cin.getline(filename,15);
  indata.open(filename,ios::in);
  if(indata.fail())// check id open succeeded
  {
     cerr<<"\nError opening file : "<<filename;
     getch();
     exit(1);}
  while(!indata.eof())// check for eof
  {
     indata.get(c);
//c=indata.get();
     cout<<c;
  }
  indata.close();
  getch();}
```

- **Text Files and Binary Files (Comparison)**
- The default access mode for file access is *text mode*. A text file is an ASCII file, compatible with most other programming languages and applications. Programs that read ASCII files can read data you create as C++ text files.
- If you specify binary access, C++ creates or reads the file in binary format. Binary data files are *"squeezed"*- that is, they take less space than text files. The disadvantage of using binary files is that other programs can't always read the data files. Only C++ programs written to access binary files can read and write them. The advantage of binary files is that you save disk space because your data files are more compact.

- The binary format is a *system-specific* file format. In other words, not all computers can read a binary file created on another computer.
- ***Random Access File Concepts***
- Random access enables you to read or write any data in your disk file without having to read and write every piece of data that precedes it.
- Generally you read and write file records. A record to a file is analogues to a C++ structure. A record is a collection of one or more data values (called fields) that you read and write to disk. Generally you store data in the structures and write structures to disk. When you read a record from disk, you generally read that record into a structure variable and process it with your program.
- Most random access files are fixed-length records. Each record (a row in the file) takes the same amount of disk space. With fixed length records, your computer can better calculate where on the disk the desired record is located.

0	100	200	300	400	500	Byte offsets
100 Bytes	100 bytes	……		100 bytes	100 bytes	

- **Opening Random-Access Files**
- There is really no difference between sequential files and random files in C++. The difference between the files is not physical, but lies in the method that you use to access them and update them.
- The ostream member function write outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream.
- When the stream is associated with a file, function write writes the data at the location in specified by the “put” file position pointer. The istream member function read inputs a fixed number of bytes from the specified stream into an area in memory beginning at the specified address.
- When the stream is associated with a file, function read inputs bytes at the location in the file specified by the “get” file poison pointer.

- **File Pointer and their Manipulators to access files randomly**
- Each file has two pointers one is called *input pointer* and second is *output pointer*. The input pointer is called *get pointer* and the output pointer is called *put pointer*.
- When input and output operation take places, the appropriate pointer is automatically set according to the access mode. For example when we open a file in reading mode, file pointer is automatically set to the start of the file. When we open a file in append mode, the file pointer is automatically set to the end of file.
- In C++ there are some manipulators by which we can control the movement of the pointer. The available manipulators are.

•seekg(): this moves get pointer i.e input pointer to a specified location.

For eg. infile.seekg(5); move the file pointer to the byte number 5 from starting point.

•seekp(): this move put pointer (output pointer) to a specified location for example: outfile.seekp(5);

•tellg(): this gives the current position of get pointer (input pointer)

•tellp(): this gives the current position of put pointer (output pointer)

The other prototype for these functions is:
seekg(offset, refposition);
seekp(offset, refposition);
The parameter offset represents the number of bytes the file pointer is to be moved from the
location specified by the parameter refposition.
The refposition takes one of the following three constants defined in the ios class.

- **ios::beg** start of the file
- **ios::cur** current position of the pointer
- **ios::end** end of the file

- By the above statement in length, the total number bytes of the file are assigned to the integer variable length. Because the file is opened in append mode that means, the file pointer is the last part of the file.
- Now lets see the seekg() function in action

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
int main()
{
  fstream fileobj;
  char ch; //holds A through Z
  //open the file in both output and input mode
  fileobj.open("c:\\alph.txt",ios::out|ios::in);
  if(fileobj.fail())
  {
     cerr<<"\nError opening alph.txt";
     getch();
     exit(1);
```

```
    }
    //now write the characters to the file
    for(ch = 'A'; ch <= 'Z'; ch++)
    {
       fileobj<<ch;
    }
    fileobj.seekg(8L,ios::beg);//skips eight letters, points to I
    fileobj>>ch;
    cout<<"\nThe 8th character is : "<<ch;
    fileobj.seekg(16L,ios::beg);//skips 16 letters, points to Q
    fileobj>>ch;
    cout<<"\nThe 16th letter is : "<<ch;
    fileobj.close();
    getch();
  }
```

- To point to the end of a data file, you can use the seekg() function to position the file pointer at the last byte. This statement positions the file pointer to the last byte in the file.

  Fileobj.seekg(0L,ios::end);
- This seekg() function literally reads “move the file pointer 0 bytes from the end of the file.” The file pointer now points to the end-of-file marker, but you can seekg() backwards to find other data in the file.
- The following program is supposed to read “c:\alph.txt” file backwards, printing each character as it skips back in the file.
- Be sure that the seekg() in the program seeks two bytes backwards from the *current* position, not from the beginning or the end as the previous programs. The for loop towards the end of the program needs to perform a “skip-two-bytes-back”, read-one-byte-forward” method to skip through the file backwards.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
int main()
{
  ifstream indata;
  int ctr=0;
  char inchar;
  indata.open("c:\\alph.txt",ios::in);
  if(indata.fail())
  {
     cerr<<"\nError opening alph.txt";
     getch();
     exit(1);
  }
  indata.seekg(-1L,ios::end);//points to the last byte in the file
  for(ctr=0;ctr<26;ctr++)
  {
     indata>>inchar;
     indata.seekg(-2L,ios::cur);
     cout<<inchar;
  }
```

```
    indata.close();
    getch();
  }
```

- ***Some other useful functions to access files randomly***
  - Write()

    Syntax of write: *fileobject.write((char*)* & *NameOfObject, sizeof(name of object))*

    - Function write expects data type const char* as its first argument. The second argument (sizeoff(object))of write is an integer of type size specifying the number of bytes to be written.
    - Used with output pointer object to write the block of data.
    - Uses binary version of data instead of text version.
    - Each value is stored in its binary format and each recorded is created to have a fixed length .
  - Read()

    Syntax:-fileobject.read (char*)&object , sizeoff(object);

    - Used with input pointer object to read the block of data.

Chapter 6

An introduction to Object Oriented Programming using C++

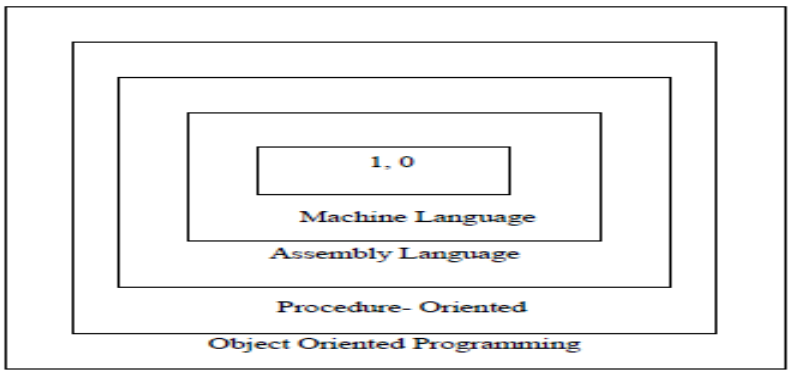
6.1. Introduction

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession. This has forced the software engineers and industry to continuously look for new approaches to **software design** and **development**, and they are becoming more and more critical in view of the increasing complexity of **software systems** as well as the highly competitive nature of the **industry**. These rapid advances appear to have created a situation of **crisis** within the industry. The following issued need to be addressed to face the crisis:

- ✓ How to represent real-life entities of problems in system design?
- ✓ How to ensure reusability and extensibility of modules?
- ✓ How to develop modules that are tolerant of any changes in future?
- ✓ How to improve software productivity and decrease software cost?
- ✓ How to improve the quality of software?

6.2. Software Evaluation

Ernest Tello, A well-known writer in the field of **artificial intelligence**, compared the evolution of **software technology** to the growth of the tree. Like a tree, the software evolution has had distinct phases "layers" of growth. These layers were building up one by one over the last five decades as shown in fig. 5.1, with each layer representing and improvement over the previous one. However, the analogy fails if we consider the life of these layers. In software system each of the layers continues to be functional, whereas in the case of trees, only the uppermost layer is functional



Alan Kay, one of the promoters of the object-oriented paradigm and the principal designer of Smalltalk, has said: "*As* ***complexity increases, architecture dominates the basic materials***". To build today's complex software it is just not enough to put together a sequence of programming statements and sets of

procedures and **modules**; we need to incorporate sound construction techniques and program structures that are easy to comprehend implement and modify.

With the advent of languages such as c, structured programming became very popular and was the main technique of the 1980's. **Structured programming** was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired result in terms of bug-free, easy-to- maintain, and reusable programs.

Object Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

6.3. Procedure-Oriented Programming

In the procedure oriented approach, the problem is viewed as the sequence of things to be done such as reading, calculating and printing such as cobol, fortran and c. The primary focus is on functions. A typical structure for procedural programming is shown in fig.5.2. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

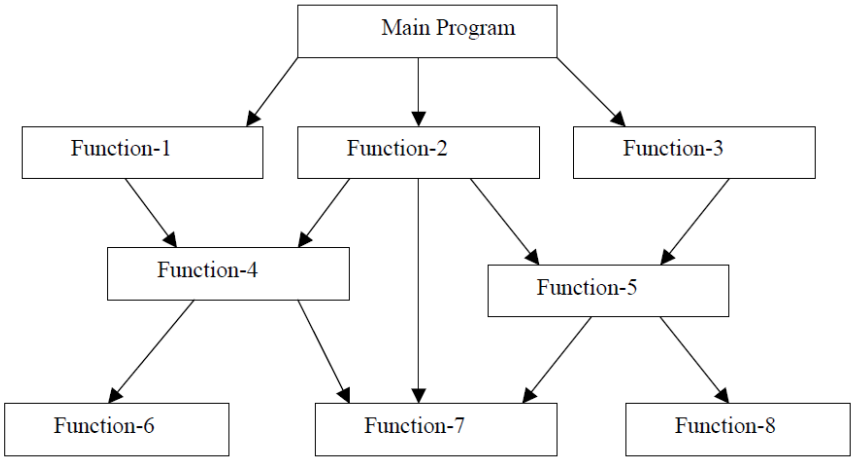


Fig. 5.2 Typical structure of procedural oriented programs

Procedure oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions. We normally use flowcharts to organize these actions and represent the flow of control from one action to another.

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that we do not model real world problems very well. This is because functions are action-oriented and do not really corresponding to the element of the problem.

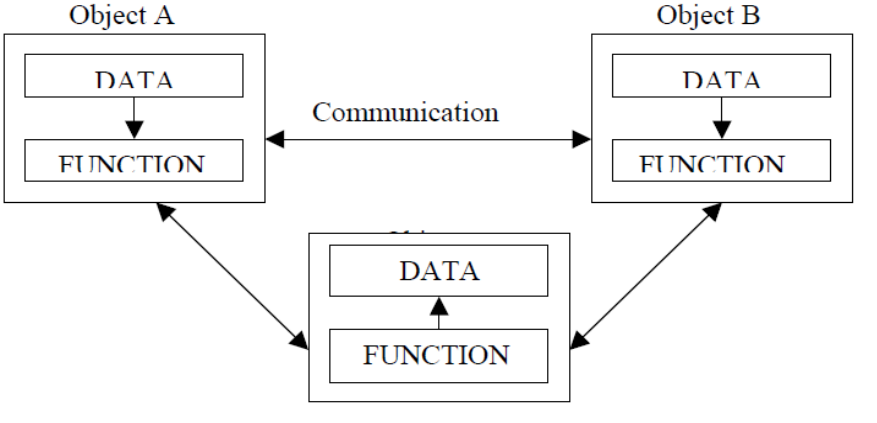
Some Characteristics exhibited by **procedure-oriented programming** are:

- ✓ Emphasis is on doing things (algorithms).
- ✓ Large programs are divided into smaller programs known as functions.
- ✓ Most of the functions share global data.
- ✓ Data move openly around the system from function to function.
- ✓ Functions transform data from one form to another.
- ✓ Employs top-down approach in program design.

6.4. Object Oriented Paradigm

The major motivating factor in the invention of **object-oriented** approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from **outside** function. **OOP** allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in **object-oriented** programs is shown in fig.5.3. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.

Organization of data and function in OOP



Some of the features of **object oriented programming** are:

- ✓ Emphasis is on data rather than procedure.
- ✓ Programs are divided into what are known as objects.
- ✓ Data structures are designed such that they characterize the objects.
- ✓ Functions that operate on the data of an object are ties together in the data structure.
- ✓ Data is hidden and cannot be accessed by external function.
- ✓ Objects may communicate with each other through function.
- ✓ New data and functions can be easily added whenever necessary.
- ✓ Follows bottom up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

6.5. Basic Concepts of Object Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- ✓ Objects
- ✓ Classes
- ✓ Data abstraction and encapsulation
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Dynamic binding
- ✓ Message passing

6.5.1.Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as **vectors, time and lists**. Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

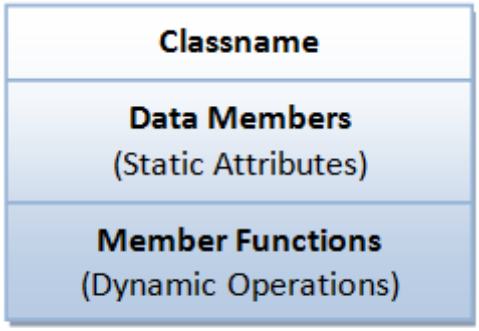
When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are to object in a program, then the customer object may send a message to the count object requesting for the bank balance. Each object contain data, and code to manipulate data. Objects can interact without having to know details of each other's data or code. It is a sufficient to know the type of message accepted, and the type of response returned by the objects. Although different

author represent them differently the following figure shows two notations that are popularly used in object-oriented analysis and design

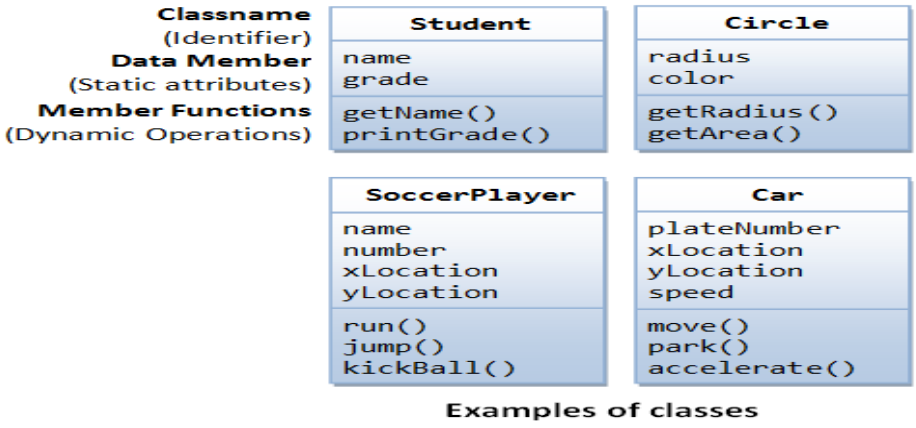
```
OBJECTS: STUDENT

DATA
        Name
        Date-of-birth
        Marks

FUNCTIONS
        Total
        Average
        Display
        ........
```

6.5.2. Classes

A class is a definition of objects of the same kind. In other words, a class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviors common to all objects of the same kind. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class.



A class is a 3-compartment box encapsulating data and functions



A class can be visualized as a three-compartment box, as illustrated:

- **Classname** (or identifier): identifies the class.
- **Data Members** or **Variables** (or attributes, states, fields): contains the static attributes of the class.
- **Member Functions** (or methods, behaviors, operations): contains the dynamic operations of the class.

In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

Class Members: The data members and member functions are collectively called **class members.**

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

Brief Summary

- A class is a programmer-defined, abstract, self-contained, reusable software entity that mimics a real-world thing.

A class is a 3-compartment box containing the name, data members (variables) and the member functions.

A class encapsulates the data structures (in data members) and algorithms (member functions). The values of the data members constitute its state. The member functions constitute its behaviors.

An instance is an instantiation (or realization) of a particular item of a class.

6.5.2.1. C++ Class Definitions:

- When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

- A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword class as follows:

```
class Box
{
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};
```

- The keyword public determines the **access attributes** of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

6.5.2.2. Define C++ Objects:

- A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;        // Declare Box1 of type Box
Box Box2;        // Declare Box2 of type Box
```

- Both of the objects Box1 and Box2 will have their own copy of data members.

6.5.2.3. Accessing the Data Members:

- The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

```
#include <iostream>
using namespace std;
class Box
{
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};
int main( )
{
  Box Box1;        // Declare Box1 of type Box
  Box Box2;        // Declare Box2 of type Box
```

```
  double volume = 0.0;     // Store the volume of a box here

 // box 1 specification
  Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;
  // box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;
  // volume of box 1
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;
  // volume of box 2
  volume = Box2.height * Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
 }
```

- When the above code is compiled and executed, it produces the following result:

  Volume of Box1 : 210

  Volume of Box2 : 1560

- It is important to note that private and protected members cannot be accessed directly using direct member access operator (.). We will learn how **private and protected** members can be accessed.

Defining member functions

Class member function can be defined inside the class or outside the class. If the member function of a class is **defined outside the** class it follows the following form.

```
Return type ClassName :: member function(parameter list)
{
Statements;
}
```

Example

```
#include <iostream.h>
class CRectangle {
int x, y;
public:
void set_values (int,int);
int area (void)
{
return (x*y);
}//member definition inside the class
};
//define member function set_Values
void CRectangle::set_values (int a, int b) {
x = a;
y = b;
}
int main () {
CRectangle  rect;
rect.set_values (3,4);
cout << "area: " << rect.area();
}
```

Classes inside classes

A part from member data and member functions, classes can also define their own data types. These new class definitions, inside the given class, can then be used by the class's member functions as well as from outside the class. In the latter case, one has to use the scope operator. See the following example for details.

```
#include <iostream>
using namespace std;
class myclass {
int a;
public:
class local_type {
double t;
public:
local_type(double x)
```

```
{
 t = x;
}
void print() const { cout << "Local type: " << t << endl; }
};
void print() const { cout << a << endl; }
};
int main() {
myclass::local_type z(2.3);
z.print();
return 0;
}
```

Note that the same access restriction for member data and member functions also applies to classes defined inside other classes. I.e. these classes can only be used from outside the class in which they are defined, if they were declared as public.

6.5.3. Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become totally operative and to avoid returning unexpected values during their execution For example, what would happen if in the previous example (in class member definition) we called the function area() before having called function set values? Probably an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function: **a constructor**, which can be declared by naming a member function with the same name as the class. This constructor function will be called automatically when a new instance of the class is created (when declaring a new object or allocating an object of that class) and only then. We are going to implement **CRectangle** including a constructor:

Example: - // classes example

```
#include <iostream.h>
class CRectangle {
int width, height;
public:
CRectangle (int,int);//constructor is created
int area (void)
 {
return (width*height);
}
};
CRectangle::CRectangle (int a, int b) {
```

```
rect
area: 12
rectb
area: 30
```

```
width = a;
height = b;
}
int main () {
CRectangle rect (3,4);// object rect with initial value
CRectangle rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
}
```

The **Destructo**r fulfills the opposite functionality of **constructor**. It is automatically called when an object is released from the memory, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using operator **delete**.

The destructor must have the same name as the class with a tilde (**~**) as prefix and it must return no value.

The use of destructors is specially suitable when an object assigns dynamic memory during its life and at the moment of being destroyed we want to release the memory that it has used.

Example on **constructors and destructors**

```
#include <iostream.h>
class CRectangle {
int *width, *height;
public:
CRectangle (int,int);
~CRectangle ();
int area (void) {return (*width * *height);}
};
CRectangle::CRectangle (int a, int b) {
width = new int;
height = new int;
*width = a;
*height = b;
}
CRectangle::~CRectangle () {
delete width;
delete height;
}
```

```
int main () {
CRectangle rect (3,4), rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
```

6.5.3.1. Overloading Constructors

Like any other function, a constructor can also be overloaded with several functions that have the same name but different types or numbers of parameters. Remember that the compiler will execute the one that matches at the moment at which a function with that name is called. In this case, at the moment at which a class object is declared.

In fact, in the cases where we declare a class and we do not specify any constructor the compiler automatically assumes two overloaded constructors ("**default constructor**" and "**copy constructor").**

For example, for the class:

```
class CExample {
public:
int a,b,c;
void multiply (int n, int m) { a=n; b=m;
c=a*b; };
};
```

with no constructors, the compiler automatically assumes that it has the following constructor member functions:

- **Empty constructor**

It is a constructor with no parameters defined as nop(empty block of instructions). It does nothing.

```
CExample::CExample () { };
```

- **Copy constructor**

It is a constructor with only one parameter of its same type that assigns to every non static class member variable of the object a copy of the passed object.

```
CExample::CExample (const CExample& rv) {
a=rv.a; b=rv.b; c=rv.c;
}
```

It is important to realize that both default constructors: the empty construction and the copy constructor exist only if no other constructor is explicitly declared. In case that any constructor with any number of parameters is declared, none of these two default constructors will exist. So if you want them to be there, you must define your own ones.

Of course, you can also overload the class constructor providing different constructors for when you pass parameters between parenthesis and when you do not (empty):

// overloading class constructors

```
#include <iostream.h>
class CRectangle {
int width, height;
public:
CRectangle ();
CRectangle (int,int);
int area (void) {return (width*height);}
};
CRectangle::CRectangle () {
width = 5;
height = 5;
}
CRectangle::CRectangle (int a, int b) {
width = a;
height = b;
}
int main () {
CRectangle rect (3,4);
CRectangle rectb;
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
}
output
```

```
rect
area: 12
rectb
area: 25
```

In this case **rectb** was declared without parameters, so it has been initialized with the constructor that has no parameters, which declares both width and height with a value of 5.

```
CRectangle rectb;  // right
CRectangle rectb(); // wrong!
```

6.5.4. Data Abstraction in C++

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the sort() function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use classes to define our own abstract data types (ADT). You can use the cout object of class ostream to stream data to standard output like this:

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

int main( )
{
```

```
cout << "Hello C++" <<endl;
return 0; }
```

Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

Access Labels Enforce Abstraction:

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction:

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example:

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

```
#include <iostream>
using namespace std;
class Adder{
  public:
    // constructor
    Adder(int i = 0)
    {
      total = i;
    }
```

```
    // interface to outside world
    void addNum(int number)
    {
        total += number;
    }
    // interface to outside world
    int getTotal()
    {
        return total;
    };
  private:
    // hidden data from outside world
    int total;
};
int main( )
{
  Adder a;
    a.addNum(10);
  a.addNum(20);
  a.addNum(30);
  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Total 60

Above class adds numbers together, and returns the sum. The public members addNum and getTotal are the interfaces to the outside world and a user needs to know them to use the class. The private member total is something that the user doesn't need to know about, but is needed for the class to operate properly

6.5.5. Data Encapsulation in C++

All C++ programs are composed of the following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

- Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.
- Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.
- C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain private, protected and public members. By default, all items defined in a class are private. For example:

```
class Box
{
  public:
    double getVolume(void)
    {
      return length * breadth * height;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};
```

- The variables length, breadth, and height are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.
- To make parts of a class public (i.e., accessible to other parts of your program), you must declare them after the public keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.
- Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example:

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

```
#include <iostream>
using namespace std;
class Adder{
  public:
```

```
    // constructor
    Adder(int i = 0)
    {
      total = i;
    }
    // interface to outside world
    void addNum(int number)
    {
        total += number;
    }
    // interface to outside world
    int getTotal()
    {
        return total;
    };
  private:
    // hidden data from outside world
    int total;
};
int main( )
{
  Adder a;
    a.addNum(10);
  a.addNum(20);
  a.addNum(30);
  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

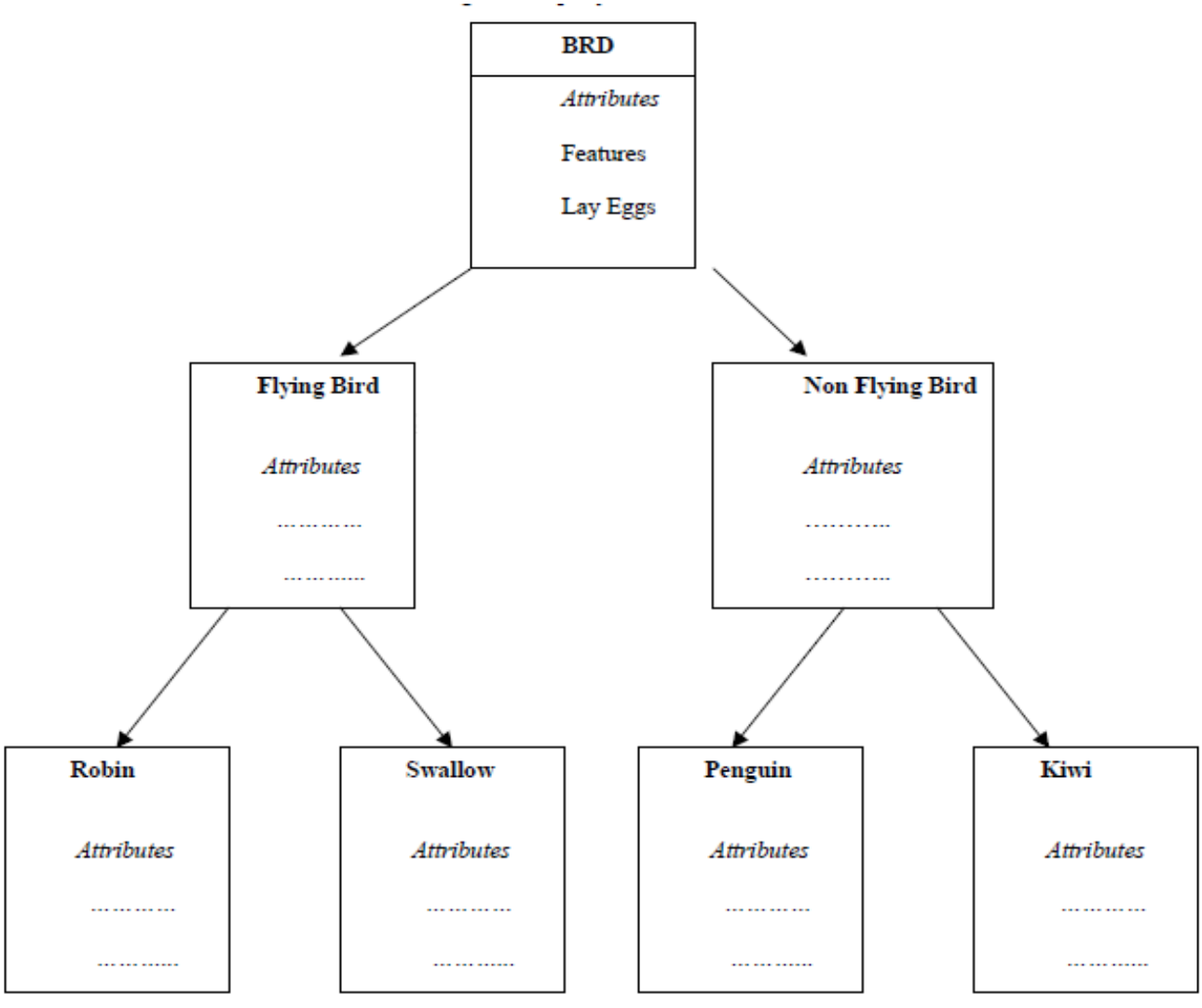
When the above code is compiled and executed, it produces the following result:

Total 60

Above class adds numbers together, and returns the sum. The public members addNum() and getTotal() are the interfaces to the outside world and a user needs to know them to use the class. The private member total is something that is hidden from the outside world, but is needed for the class to operate properly

6.5.6. C++ Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of *hierarchical classification.* For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig below.



In OOP, the concept of inheritance provides the idea of *reusability.* This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class i.e almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduced any undesirable side-effects into the rest of classes.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

class derived-class: access-specifier base-class

Where **access-specifier** is one of **public**, **protected**, or **private**, and **base-class** is the name of a previously defined class. If the access-specifier is not used, then it is **private** by default.

Consider a base class Shape and its derived class Rectangle as follows

```
#include <iostream>
using namespace std;
// Base class
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
    int height;
};
class Rectangle: public Shape // Derived class
{
  public:
    int getArea()
    {
      return (width * height);
    }
};
```

```
int main(void)
{
   Rectangle Rect;
    Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Total area: 35

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

6.5.6.1. Type of Inheritance:

When deriving a class from a base class, the base class may be inherited through public, protected or **private inheritance**. The type of inheritance is specified by the access-specifier as explained above. We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

- ✓ **Public Inheritance:** When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
- ✓ **Protected Inheritance:** When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

- ✓ **Private Inheritance:** When deriving from a private base class, public and protected members of the base class become private members of the derived class.

6.5.6.2. Multiple Inheritances:

A C++ class can inherit members from more than one class and here is the extended syntax:

class derived-class: access baseA, access baseB....

Where access is one of public, protected, or private and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

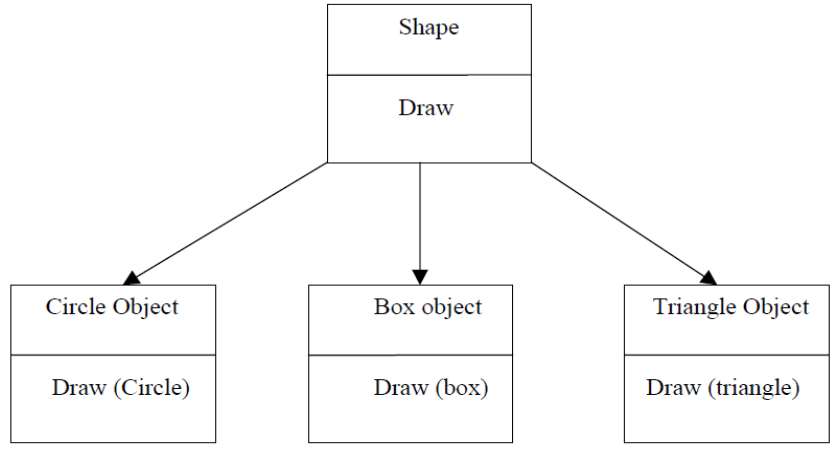
```
#include <iostream>
 using namespace std;
// Base class Shape
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
    int height;
};

// Base class PaintCost
class PaintCost
{
  public:
    int getCost(int area)
    {
      return area * 70;
    }
};

// Derived class
```

```
class Rectangle: public Shape, public PaintCost
{
  public:
    int getArea()
    {
      return (width * height);
    }
};
int main(void)
{
  Rectangle Rect;
  int area;
   Rect.setWidth(5);
  Rect.setHeight(7);
  area = Rect.getArea();
    // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;
  // Print the total cost of painting
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;
  return 0;
}
```

6.5.7. Polymorphism

- Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

- The following figure illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as function overloading.



- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function. Consider the following example where a base class has been derived by other two classes:

```
#include <iostream>
using namespace std;
 class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
```

```
    int area ()
    {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};
class Triangle: public Shape{
  public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
int main( ) // Main function for the program
{
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);
  // store the address of Rectangle
  shape = &rec;
  // call rectangle area.
  shape->area();
  // store the address of Triangle
  shape = &tri;
  // call triangle area.
  shape->area();
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Parent class area

Parent class area

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or

static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword virtual so that it looks like this:

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    virtual int area()
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

6.5.8. Virtual Function:

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

6.5.8.1. Pure Virtual Functions:

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

6.5.9. C++ Overloading (Operator and Function)

- C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively. An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
- When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

6.5.9.1. Function overloading in C++:

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type. Following is the example where same function print() is being used to print different data types:

```
#include <iostream>
using namespace std;
 class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};
int main(void)
{
   printData pd;
    // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

6.5.9.2. Operators overloading in C++:

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below:

```
#include <iostream>
using namespace std;
class Box
{
  public:
    double getVolume(void)
    {
      return length * breadth * height;
    }
    void setLength( double len )
    {
       length = len;
    }
    void setBreadth( double bre )
    {
       breadth = bre;
    }

    void setHeight( double hei )
    {
       height = hei;
    }
    // Overload + operator to add two Box objects.
    Box operator+(const Box& b)
    {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
```

```
        box.height = this->height + b.height;
        return box;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
// Main function for the program
int main( )
{
   Box Box1;              // Declare Box1 of type Box
   Box Box2;              // Declare Box2 of type Box
   Box Box3;              // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here
    // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
    // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);
    // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
    // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   // Add two object as follows:
   Box3 = Box1 + Box2;
   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
```

Volume of Box2 : 1560

Volume of Box3 : 5400

6.5.10. Overriding

Defining a function in the derived class with same name as in the parent class is called overriding. In C++, the base class member can be overridden by the derived class function with the same signature as the base class function. Method overriding is used to provide different implementations of a function so that a more specific behavior can be realized.

6.5.11. Relationships between classes

5.5.11.1 Friend functions (friend keyword)

In the previous section we have seen that there were three levels of internal protection for the different members of a class: public, protected and private. In the case of members Protected and private, these could not be accessed from outside the same class at which they are declared. Nevertheless, this rule can be transgressed with the use of the friend keyword in a class, so we can allow an external function to gain access to the protected and private members of a class.

In order to allow an external function to have access to the private and protected members of a class we have to declare the prototype f the external function that will gain access preceded by the keyword friend within the class declaration that shares its members. In the following example we declare the friend function duplicate:

Example for friend functions

```
#include <iostream.h>
class CRectangle {
int width, height;
public:
void set_values (int, int);
int area (void) {return (width * height);}
friend CRectangle duplicate (CRectangle);
};
void CRectangle::set_values (int a, int b) {
width = a;
height = b;
}
CRectangle duplicate (CRectangle rectparam)
{
```

```
CRectangle rectres;
rectres.width = rectparam.width*2;
rectres.height = rectparam.height*2;
return (rectres);
}
int main () {
CRectangle rect, rectb;
rect.set_values (2,3);
rectb = duplicate (rect);
cout << rectb.area();
}
```

From within the duplicate function, that is a friend of CRectangle, we have been able to access the members width and height of different objects of type CRectangle. Notice that neither in the declaration of duplicate() nor in its later use in main()have we considered duplicate as a member of class CRectangle.

The friend functions can serve, for example, to conduct operations between two different classes. Generally the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to make the process. Such as in the previous example, it would have been shorter to integrate duplicate () within the class CRectangle.

5.5.11.2. Friend classes (friend)

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, allowing that the second one access to the protected and private members of the first one.

Example for friend class

```
#include <iostream.h>
class CSquare;
class CRectangle {
int width, height;
public:
int area (void)
{return (width * height);}
void convert (CSquare a);
};
class CSquare {
```

```
private:
int side;
public:
void set_side (int a)
{side=a;}
friend class CRectangle;
};
void CRectangle::convert (CSquare a) {
width = a.side;
height = a.side;
}
int main () {
CSquare sqr;
CRectangle rect;
sqr.set_side(4);
rect.convert(sqr);
cout << rect.area();
return 0;
}
```

In this example we have declared **CRectangle** as a friend of **CSquare** so that **CRectangle** can access the protected and private members of **CSquare**, more concretely **CSquare::side**, that defines the square side width.

You may also see something new in the first instruction of the program, that is the empty prototype of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare(as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous definition for CSquareth is class would not be visible from within the definition of CRectangle. Consider that friendships are not corresponded if we do not explicitly specify it. In our CSquare example CRectangle is considered as a class friend, but CRectangle does not do the proper thing with CSquare, so CRectangle can access to the protected and private members of CSquare but not the reverse way. Although nothing prevents us from declaring CSquareas a friend of CRectangle.