CPP Programming

INDEX

	· File handling functions	
	. Different Streams	
11	Programming Practice	

1. **Beginning With CPP**

    1. **What is CPP?**

C++ is an object oriented programming (OOP) language, developed by Bjarne Stroustrup, and is an extension of C language. It is therefore possible to code C++ in a "C style" or "object-oriented style." In certain scenarios, it can be coded in either way and is thus an effective example of a hybrid language.

C++ is a general purpose object oriented programming language. It is considered to be an intermediate level language, as it encapsulates both high and low level language features. Initially, the language was called 'C with classes' as it had all properties of C language with an additional concept of 'classes'. However, it was renamed to C++ in 1983.

It is pronounced "C-Plus-Plus

C++ is one of the most popular languages primarily utilized with system/application software, drivers, client-server applications and embedded firmware.

The main highlight of C++ is a collection of pre-defined classes, which are data types that can be instantiated multiple times. The language also facilitates declaration of user defined classes. Classes can further accommodate member functions to implement specific functionality. Multiple objects of a particular class can be defined to implement the functions within the class. Objects can be defined as instances created at run time. These classes can also be inherited by other new classes which take in the public and protected functionalities by default.

C++ includes several operators such as comparison, arithmetic, bit manipulation, logical operators etc. One of the most attractive features of C++ is that it enables the overloading of certain operators such as addition.

A few of the essential concepts within C++ programming language include polymorphism, virtual and friend functions, templates, namespaces and pointers.

    2. **Why we use CPP?**

The major difference being OOPS concept, C++ is an object oriented language whereas C language is a procedural language. Apart form this there are many other features of C++ which gives this language an upper hand on C laguage.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
3. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
4. Exception Handling is there in C++.
5. The Concept of Virtual functions and also Constructors and Destructors for Objects.

6. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
7. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

3. Difference between C and CPP

S. No.	C	C++
1	C is a structural or procedural programming language.	C++ is an object oriented programming language.
2	Emphasis is on procedure or steps to solve any problem.	Emphasis is on objects rather than procedure.
3	Functions are the fundamental building blocks.	Objects are the fundamental building blocks.
4	In C, the data is not secured.	Data is hidden and can't be accessed by external functions.
5	C follows top down approach.	C++ follows bottom up approach
6	C uses scanf() and printf() function for standard input and output.	C++ uses cin>> and cout<< for standard input and output.
7	Variables must be defined at the beginning in the function.	Variables can be defined anywhere in the function.
8	In C, namespace feature is absent.	In C++, namespace feature is present.
9	C is a middle level language.	C++ is a high level language.
10	Programs are divided into modules and functions.	Programs are divided into classes and functions.
11	C doesn't support exception handling directly. Can be done by using some other functions.	C++ supports exception handling. Done by using try and catch block.
12	Features like function overloading and operator overloading is not present.	C++ supports function overloading and operator overloading.
13	C program file is saved with .C extension.	C++ program file is saved with .CPP extension.

4. Control Statements in CPP

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C++

handles decision-making by supporting the following statements,

- *if* statement
- *switch* statement
- conditional operator statement
- *goto* statement

Decision making with *if* statement

The *if* statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple *if* statement
2. *If....else* statement
3. Nested *if....else* statement
4. *else if* statement

Simple *if* statement

The general form of a simple *if* statement is,

```
if( expression )
{
 statement-inside;
}
 statement-outside;
```

If the *expression* is true, then 'statement-inside' it will be executed, otherwise 'statement-inside' is skipped and only 'statement-outside' is executed.

Example :

```
#include< iostream.h>
int main( )
{
 int x,y;
 x=15;
 y=13;
 if (x > y )
 {
  cout << "x is greater than y";
 }
}
```

Output : x is greater than y

***if...else* statement**

The general form of a simple *if...else* statement is,

```
if( expression )
{
 statement-block1;
}
else
{
```

```
 statement-block2;
}
```

If the 'expression' is true, the 'statement-block1' is executed, else 'statement-block1' is skipped and 'statement-block2' is executed.

Example :

```
void main( )
{
 int x,y;
 x=15;
 y=18;
 if (x > y )
 {
  cout << "x is greater than y";
 }
 else
 {
  cout << "y is greater than x";
 }
}
```

Output : y is greater than x

Nested *if....else* statement

The general form of a nested *if...else* statement is,

```
if( expression )
{
  if( expression1 )
    {
      statement-block1;
    }
  else
    {
      statement-block 2;
    }
}
else
{
 statement-block 3;
}
```

if 'expression' is false the 'statement-block3' will be executed, otherwise it continues to perform the test for 'expression 1' . If the 'expression 1' is true the 'statement-block1' is executed otherwise 'statement-block2' is executed.

Example :

```
void main( )
{
 int a,b,c;
 clrscr();
 cout << "enter 3 number";
```

```
 cin >> a >> b >> c;
 if(a > b)
 {
  if( a > c)
  {
    cout << "a is greatest";
  }
  else
  {
    cout << "c is greatest";
  }
 }
 else
 {
  if( b> c)
   {
     cout << "b is greatest";
   }
  else
   {
     printf("c is greatest");
   }
  }
getch();
}
```

else-if ladder

The general form of else-if ladder is,

```
if(expression 1)
{
 statement-block1;
}
else if(expression 2)
{
 statement-block2;
}
else if(expression 3 )
{
 statement-block3;
}
else
 default-statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as the true condition is found, the statement associated with it is executed.

Example :

```
void main( )
{
 int a;
 cout << "enter a number";
```

```
cin >> a;
if( a%5==0 && a%8==0)
{
 cout << "divisible by both 5 and 8";
}
else if( a%8==0 )
{
 cout << "divisible by 8";
}
else if(a%5==0)
{
 cout << "divisible by 5";
}
else
{
 cout << "divisible by none";
}
getch();
}
```

Points to Remember

1. In *if* statement, a single statement can be included without enclosing it into curly braces { }

```
2. int a = 5;
3. if(a > 4)
4.    cout << "success";
```

   No curly braces are required in the above case, but if we have more than one statement inside *if* condition, then we must enclose them inside curly braces.

5. == must be used for comparison in the expression of *if* condition, if you use = the expression will always return true, because it performs assignment not comparison.
6. Other than **0(zero)**, all other values are considered as true.

```
7. if(27)
8.  cout << "hello";
```

   Note: Remaining control statements we have already seen in C programing Language**.**

Pillars of OOPS:

1. **Abstraction**
   Abstraction is a process of exposing essential feature of an entity while hiding other irrelevant detail. *abstraction reduces code complexity and at the same time it makes*

your aesthetically pleasant.

2. **Encapsulation**
   We have to take in consideration that Encapsulation is somehow related to Data Hiding. Encapsulation is when you hide your modules internal data and all other implementation details/mechanism from other modules.it is also a way of restricting access to certain properties or component. *Remember, Encapsulation is not data hiding, but Encapsulation leads to data hiding*
3. **Inheritance**
   *The ability of creating a new class from an existing class.*Like there word Inheritance literally means it is a practice of passing on property, titles, debts, rights and obligations upon the death of an individual.in OOP this is somehow true(Except the death of an individual) ,where The base class(the existing class sometimes called as the Parent class) has properties and methods that will be inherited by the sub class(somtimes called a subtype or child class) and *it can have additional properties or methods.*
   Inheritance is also a way to use code of an existing objects.
4. **Polymorphism**
   Just like in biology, Polymorphism refers to the ability to take into different forms or stages.A subclass can define its own unique behaviour and still share the same functionalities or behavior of its parent/base class.Yes, you got it right, subclass can have their own behavior and share some behaviour from its parent class BUT!! not vice versa. A parent class cannot have the behavior of its subclass.

2. Classes and Objects

3. The classes are the most important feature of C++ that leads to Object Oriented programming. Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.
4. The variables inside class definition are called as data members and the functions are called member functions.
5. **For example :** Class of birds, all birds can fly and they all have wings and beaks. So here flying is a behavior and wings and beaks are part of their characteristics. And there are many different birds in this class with different names but they all posses this behavior and characteristics.
6. Similarly, class is just a blue print, which declares and defines characteristics and behavior, namely data members and member functions respectively. And all objects of this class will share these characteristics and behavior.

Objects

Class is mere a blueprint or a template. No storage is assigned when we define a class. Objects are instances of class, which holds the data variables declared in class and the member functions

work on these class objects.

Each object has different data variables. Objects are initialised using special class functions called **Constructors**. We will study about constructors later.

And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object. C++ doesn't have Automatic Garbage Collector like in JAVA, in C++ Destructor performs this task.

Syntax:

- Class definition starts with keyword class followed by the class name, Class body enclosed by a pair of curly braces. A class definition must be followed either by a semicolon.
- Syntax-

```
class class_name
 {
   access_modifier:
                declaration code
    access_modifier:
                declaration code
 };
```

- Member functions can be defined within the class definition or separately using **scope resolution operator**(::)
- **Syntax-**

```
return_type class_name::Function_name(para.list)
 {
      //code
 }
```

- **E.g.-**

```
void Employee::getdata()
 {
    cout<<"\n Enter id and name:";
 }
```

```
class Abc
{
 int x;
 void display(){} //empty function
};
```

```
in main()
{
 Abc obj;   // Object of class Abc created
}
```

Any object should satisfy four characteristics

1. State
2. Behavior
3. Identity
4. Responsibility

Access Modifier

public members:

a. A public member is accessible from anywhere outside the class but within a program.

private members:

b. A private member variable or function cannot be accessed, or even viewed from outside the class.

c. Only the class and friend functions can access private members.

d. By default all the members of a class would be private

protected members:

e. A protected member variable or function is very similar to a private member

f. It provided one additional benefit that they can be accessed in child classes which are called derived classes.

g. Used at the time of inheritance.

e.g:

```
class Student
{
 public:
 int rollno;
 string name;
};

int main()
{
 Student A;
 Student B;
 A.rollno=1;
 A.name="Adam";

 B.rollno=2;
 B.name="Bella";

 cout <<"Name and Roll no of A is :"<< A.name << A.rollno;
 cout <<"Name and Roll no of B is :"<< B.name << B.rollno;
}
```

3. Functions:

Member Functions in Classes

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name alng with function name.

Example :

```
class Cube
{
 public:
 int side;
 int getVolume();     // Declaring function getVolume with no argument and return type int.
};
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
class Cube
{
 public:
 int side;
 int getVolume();
}

int Cube :: getVolume()     // defined outside class definition
{
 return side*side*side;
}
```

5. Constructor

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object.

```
class A
{
 int x;
 public:
 A();  //Constructor
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
 int i;
 public:
 A(); //Constructor declared
};

A::A()   // Constructor definition
{
 i=1;
}
```

Types of Constructors

Constructors are of three types :

1. Default Constructor
2. Parametrized Constructor
3. Copy COnstructor

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax :

```
class_name ()
{ Constructor Definition }
```

Example :

```
class Cube
{
int side;
public:
Cube()
 {
  side=10;
 }
};

int main()
{
Cube c;
cout << c.side;
}
```

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 int side;
};

int main()
{
 Cube c;
 cout << c.side;
}
```

Output : 0

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

Example :

```
class Cube
{
 int side;
 public:
 Cube(int x)
  {
   side=x;
  }
};

int main()
{
 Cube c1(10);
 Cube c2(20);
 Cube c3(30);
 cout << c1.side;
 cout << c2.side;
 cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

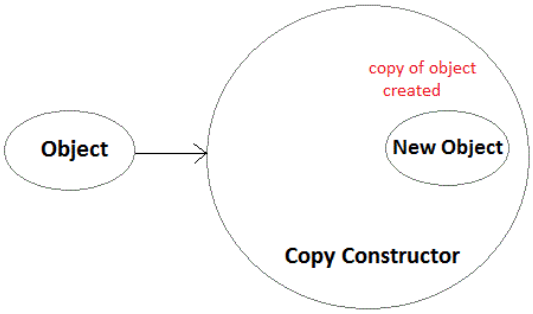
Copy Constructor

Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name.he compiler provides a default Copy Constructor to all the classes.

Syntax of Copy Constructor

class-name (*class-name &*)
{
. . . .
}

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.



Constructor Overloading

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

You can have any number of Constructors in a class that differ in parameter list.

```
class Student
{
 int rollno;
 string name;
 public:
 Student(int x)
 {
  rollno=x;
  name="None";
 }
 Student(int x, string str)
 {
  rollno=x ;
  name=str ;
 }
};

int main()
{
 Student A(10);
 Student B(11,"Ram");
}
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

Destructors

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

```
class A
{
 public:
 ~A();
};
```

Destructors will never have any arguments.

6. Inline and Friend Function

All the member functions defined inside the class definition are by default declared as Inline. Let us have some background knowledge about these functions.

You must remember Preprocessors from C language. Inline functions in C++ do the same thing what Macros do in C. Preprocessors were not used in C++ because they had some drawbacks.

For an inline function, declaration and definition must be done together. For example,

```
inline void fun(int a)
{
 return a++;
}
```

Some Important points about Inline Functions

1. We must keep inline functions small, small inline functions have better efficiency.
2. Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to **code bloat**, and might affect the speed too.
3. Hence, it is adviced to define large functions outside the class definition using scope resolution :: operator, because if we define such functions inside class definition, then they become inline automatically.
4. Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

Example:

```
inline int Max(int x, int y)
 {
     return (x > y)? x : y;
 }
 // Main function for the program
 int main( )
{
 cout << "Max (20,10): " << Max(20,10) << endl;
 cout << "Max (0,200): " << Max(0,200) << endl;
 cout << "Max (100,1010): " << Max(100,1010) << endl;
 return 0;
 }
```

Output: 20,200,1010

Friend Function:

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition

with keyword **friend** as follows:

```
class Box
{
  double width;
public:
  double length;
  friend void printWidth( Box box );
  void setWidth( double wid );
};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:

```
friend class ClassTwo;
```

Consider the following program:

```
#include <iostream>

using namespace std;

class Box
{
  double width;
public:
  friend void printWidth( Box box );
  void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid )
{
   width = wid;
}

// Note: printWidth() is not a member function of any class.
void printWidth( Box box )
{
  /* Because printWidth() is a friend of Box, it can
   directly access any member of this class */
  cout << "Width of box : " << box.width <<endl;
}

// Main function for the program
int main( )
{
  Box box;

  // set box width without member function
  box.setWidth(10.0);

  // Use friend function to print the wdith.
  printWidth( box );

  return 0;
```

```
}
```

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

```
Width of box : 10
```

```
class Distance
{
   private:
      int meter;
   public:
      Distance(): meter(0){ }
      friend int func(Distance);  //friend function
};
int func(Distance d)           //function definition
{
   d.meter=5;        //accessing private data from non-member function
   return d.meter;
}
int main()
{
   Distance D;
   cout<<"Distace: "<<func(D);
   return 0;
}
```

Output: Distance: 5

Here, friend function func() is declared inside Distance class. So, the private data can be accessed from this function.Though this example gives you what idea about the concept of friend function, this program doesn't give you idea about when friend function is helpful.

Suppose, you need to operate on objects of two different class then,friend function can be very helpful. You can operate on two objects of different class without using friend function but, you program will be long, complex and hard to understand.

5. Static Data Member and Member Functions:

Static Keyword

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following,

1. Static variable in functions
2. Static Class Objects
3. Static member Variable in class
4. Static Methods in class

Static variables inside Functions

Static variables when used inside function are initialized only once, and then they hold there value even through function calls.

These static variables are stored on static storage area , not in stack.

```
void counter()
{
 static int count=0;
 cout << count++;
}

int main(0
{
 for(int i=0;i<5;i++)
 {
   counter();
 }
}
```

Output : 0 1 2 3 4

Let's se the same program's output **without using static** variable.

```
void counter()
{
 int count=0;
 cout << count++;
}

int main(0
{
 for(int i=0;i<5;i++)
 {
   counter();
 }
}
```

Output : 0 0 0 0 0

If we do not use static keyword, the variable count, is reinitialized everytime when counter() function is called, and gets destroyed each time when counter() functions ends. But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.

If you don't initialize a static variable, they are by default initialized to zero.

Static class Objects

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized using constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc
{
 int i;
 public:
 Abc()
 {
  i=0;
  cout << "constructor";
```

```
 }
 ~Abc()
 {
   cout << "destructor";
 }
};

void f()
{
 static Abc obj;
}

int main()
{
 int x=0;
 if(x==0)
 {
  f();
 }
 cout << "END";
}
```

Output : constructor END destructor

You must be thinking, why was destructor not called upon the end of the scope of if condition. This is because object was static, which has scope till the program lifetime, hence destructor for this object was called when main() exits.

Static data member in class

Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.

Static member variables (data members) are not initialized using constructor, because these are not dependent on object initialization.

Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

```
class X
{
 static int i;
 public:
 X(){};
};

int X::i=1;

int main()
{
 X obj;
 cout << obj.i;   // prints value of i
}
```

Once the definition for static data member is made, user cannot redefine it. Though, arithmetic operations can be performed on it.

Static Member Functions

These functions work for the class as whole rather than for a particular object of a class.

It can be called using an object and the direct member access . operator. But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.

Example :

```
class X
{
 public:
 static void f(){};
};

int main()
{
 X::f();   // calling member function directly with class name
}
```

These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

It doesn't have any "this" keyword which is the reason it cannot access ordinary members. We will study about "this" keyword later.

7. References in C++

References are like constant pointers that are automatically dereferenced. It is a new name given to an existing storage. So when you are accessing the reference, you are actually accessing that storage.

```
int main()
{ int y=10;
  int &r = y;  // r is a reference to int y
  cout << r;
}
```

Output : 10

There is no need to use the * to dereference a reference variable.

Difference between Reference and Pointer

References	Pointers
Reference must be initialized when it is created.	Pointers can be initialized any time.
Once initialized, we cannot reinitialize a reference.	Pointers can be reinitialized any number of time.
You can never have a NULL reference.	Pointers can be NULL.
Reference is automatically dereferenced.	* is used to dereference a pointer.

References in Funtions

References are generally used for function argument lists and function return values, just like pointers.

Rules for using Reference in Functions

1. When we use reference in argument list, we must keep in mind that any change to the reference inside the function will cause change to the original argument outside th function.
2.
3. When we return a reference from a function, you should see that whatever the reference is connected to shouldn't go out of scope when fnction ends. Either make that **global** or **static**

Example to explain use of References

```
int* first (int* x)
{ (*x++);
  return x;   // SAFE, x is outside this scope
}

int& second (int& x)
{ x++;
```

```
 return x;   // SAFE, x is outside this scope
}

int& third ()
{ int q;
 return q;   // ERROR, scope of q ends here
}

int& fourth ()
{ static int x;
 return x;   // SAFE, x is static, hence lives till the end.
}

int main()
{
 int a=0;
 first(&a);   // UGLY and explicit
 second(a);   // CLEAN and hidden
}
```

We have four different functions in the above program.

- **first()** takes a pointer as argument and returns a pointer, it will work fine. The returning pointer points to variable declared outside first(), hence it will be valid even after the first() ends.
- Similarly, **second()** will also work fine. The returning reference is connected to valid storage, that is `int a` in this case.
- But in case of **third()**, we declare a variable `q` inside the function and try to return a reference connected to it. But as soon as function third() ends, the local variable q is destroyed, hence nothing is returned.
- To remedify above problem, we make x as **static** in function **fourth()**, giving it a lifetime till main() ends, hence now a reference connected to x will be valid when returned.

8. OOPS

1. Abstraction:

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.

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.

2. Encapsulation

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
```

4. Inheritance

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

NOTE : All members of a class except Private, are inherited

Purpose of Inheritance

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

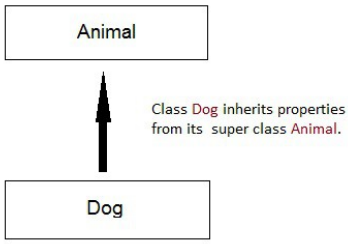
Basic Syntax of Inheritance

```
class Subclass_name : access_mode Superclass_name
```

While defining a subclass like this, the super class must be already defined or atleast declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

Example of Inheritance




```
class Animal
{ public:
  int legs = 4;
};
```

```
class Dog : public Animal
{ public:
  int tail = 1;
};

int main()
{
 Dog d;
 cout << d.legs;
 cout << d.tail;
}
```

Output : 4 1

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : **public** Superclass

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // By default its private inheritance

3) Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : **protected** Superclass

Table showing all the Visibility Modes

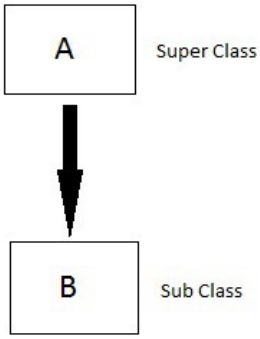
Base class	**Derived Class**		
	Public Mode	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.

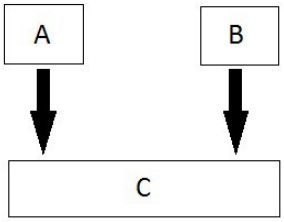


- Each class or instance object has a single parent.
- Only one base class which is derived in only one derived class.
- **Syntax:**

```
class base_class
{
  //defination
};
class derived_class:<access modifier> base_class
{
 //defination
};
```

Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.

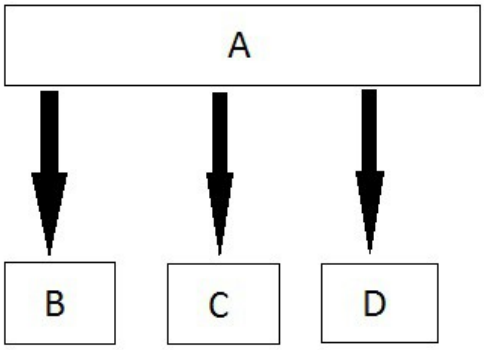


- It is possible for a derived class to inherit two or more base classes.
- **Syntax:**

```
class base_class1
{
  //defination
};
class base_class2
{
  //defination
};
class derived_class:<access modifier> base_class1,<access modifier> base_class2,...
{
 //defination
};
```

Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.

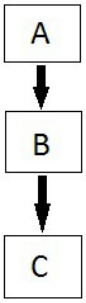


- **Hierarchical Inheritance is a method of inheritance where one or more derived classes is derived from common base class.**
- **Syntax-**

```
class base_class
{
  //defination;
};
class derived1_class:<access modifier>base_class
{
    //defination;
};
class derived2_class:<access modifier>base_class
{
  //defination;
}
```

Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.

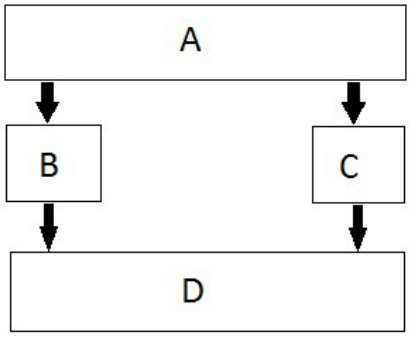


- Intermediate derived class derived from another derived class.
- **Syntax-**

```
class base_class
{
 //defination;
};
class intermidiate_class:<access modifier>base_class
{
    //defination;
};
class derived:<access modifier>intermidiate_class
{
  //defination;
}
```

Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



- **Syntax-**

```
class A
{
 //defination;
```

```
};
class B: virtual <access modifier>A
{
   //defination;
};
class C: virtual<access modifier>A
{
  //defination;
}
Class D:public A, public B
{
  //defination;
}
```

4. **Polymorphism**

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by 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:

Types of Polymorphism

- Compile time polymorphism(False polymorphism)
  - What is Overloading
    - » Function Overloading
    - » Operator Overloading

Run time polymorphism

--What is overriding

Virtual Function

Pure Virtual Function

Function Overloading

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to

perform one single operation but with different number or types of arguments, then you can simply overload the function.

Ways to overload a function

1. By changing number of Arguments.
2. By having different types of argument.

Number of Arguments different

In this type of function overloading we define two functions with same names but different number of parameters of the same type. For example, in the below mentioned program we have made two sum() functions to return sum of two and three integers.

```
int sum (int x, int y)
{
 cout << x+y;
}

int sum(int x, int y, int z)
{
 cout << x+y+z;
}
```

Here sum() function is overloaded, to have two and three arguments. Which sum() function will be called, depends on the number of arguments.

```
int main()
{
sum (10,20);  // sum() with 2 parameter will be called

sum(10,20,30);  //sum() with 3 parameter will be called
}
```

Different Datatype of Arguments

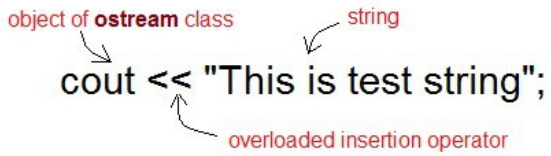
In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different. For example in this program, we have two sum() function, first one gets two integer arguments and second one gets two double arguments.

```
int sum(int x,int y)
{
 cout<< x+y;
}

double sum(double x,double y)
{
 cout << x+y;
}

int main()
{
 sum (10,20);
```

```
 sum(10.5,20.5);
}
```

Operator Overloading

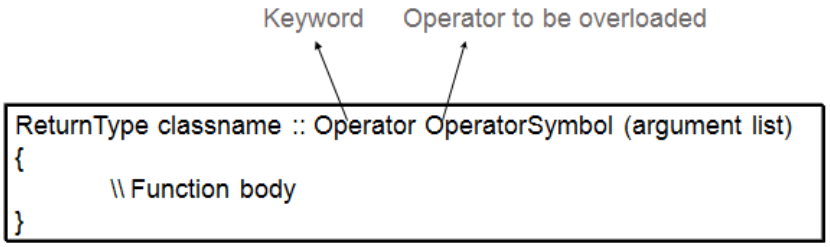
Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.



Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Example of Function Overriding

```
class Base
{
 public:
 void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}
```

In this example, function **show()** is overridden in the derived class. Now let us study how these

overridden functions are called in **main()** function.

Overloading Arithmetic Operator

Arithmetic operator are most commonly used operator in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below example we have overridden the + operator, to add to Time(hh:mm:ss) objects.

Example: overloading '+' Operator to add two time object

```
#include< iostream.h>
#include< conio.h>
class time
{
 int h,m,s;
 public:
  time()
  {
   h=0, m=0; s=0;
  }
  void getTime();
  void show()
  {
   cout<< h<< ":"<< m<< ":"<< s;
  }
 time operator+(time);   //overloading '+' operator
};
time time::operator+(time t1)    //operator function
{
 time t;
 int a,b;
 a=s+t1.s;
 t.s=a%60;
 b=(a/60)+m+t1.m;
 t.m=b%60;
 t.h=(b/60)+h+t1.h;
 t.h=t.h%12;
 return t;
}
void time::getTime()
{
  cout<<"\n Enter the hour(0-11) ";
  cin>>h;
  cout<<"\n Enter the minute(0-59) ";
  cin>>m;
  cout<<"\n Enter the second(0-59) ";
  cin>>s;
}
void main()
{
 clrscr();
 time t1,t2,t3;
 cout<<"\n Enter the first time ";
```

```
 t1.getTime();
 cout<<"\n Enter the second time ";
 t2.getTime();
 t3=t1+t2;  //adding of two time object using '+' operator
 cout<<"\n First time ";
 t1.show();
 cout<<"\n Second time ";
 t2.show();
 cout<<"\n Sum of times ";
 t3.show();
 getch();
}
```

Virtual Function

Virtual Functions

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

Problem without Virtual Keyword

```
class Base
{
 public:
 void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}

int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Early Binding Ocuurs
}
```

Output : Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Using Virtual Keyword

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

```
class Base
{
 public:
 virtual void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}

int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Late Binding Ocuurs
}
```

Output : Derived class

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

Using Virtual Keyword and Accessing Private Method of Derived class

We can call **private** function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

```
#include
using namespace std;

class A
{
   public:
   virtual void show()
   {
      cout << "Base class\n";
```

```
    }
};

class B: public A
{
private:
    virtual void show()
    {
        cout << "Derived class\n";
    }
};

int main()
{
    A *a;
    B b;
    a = &b;
    a -> show();
}
```

Output : Derived class

Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the copiler uses **VPTR**(vpointer) to point to the Virtual Function.

Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,

```
virtual void f() = 0;
```

Example of Abstract Class

```
class Base         //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
 Base obj;       //Compile Time Error
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output : Implementation of Virtual Function in Derived class

In the above example Base class is abstract, with pure virtual **show()** function, hence we cannot create object of base class.

Why can't we create Object of Abstract Class ?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.

Pure Virtual definitions

- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

```
class Base         //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};
```

```
void Base :: show()        //Pure Virtual definition
{
 cout << "Pure Virtual definition\n";
}

class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output :

Pure Virtual definition
Implementation of Virtual Function in Derived class

Generic Functions

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how do they work:

Function Template:

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
  // body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

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

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
   return a < b ? b:a;
}
int main ()
{

   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

  return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Class Template:

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name {
.
.
```

.
}

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
  private:
    vector<T> elems;     // elements

  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const{       // return true if empty.
       return elems.empty();
    }
};

template <class T>
void Stack<T>::push (T const& elem)
{
    // append copy of passed element
    elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
       throw out_of_range("Stack<>::pop(): empty stack");
    }
          // remove last element
    elems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
    if (elems.empty()) {
       throw out_of_range("Stack<>::top(): empty stack");
```

```
    }
        // return copy of last element
    return elems.back();
}

int main()
{
    try {
        Stack<int>         intStack;  // stack of ints
        Stack<string> stringStack;    // stack of strings

        // manipulate int stack
        intStack.push(7);
        cout << intStack.top() <<endl;

        // manipulate string stack
        stringStack.push("hello");
        cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (exception const& ex) {
        cerr << "Exception: " << ex.what() <<endl;
        return -1;
    }
} If we compile and run above code, this would produce the following result:
7
hello
Exception: Stack<>::pop(): empty stack
```

10.

10. File Handling

File Handling concept in C++ language is used for store a data permanently in computer. Using file handling we can store our data in Secondary memory (Hard disk).

Why use File Handling

- For permanet storage.
- The transfer of input - data or output - data from one computer to another can be easily done by using files.

For read and write from a file you need another standard C++ library called **fstream**, which defines three new data types:

Datatype	Description
ofstream	This is used to create a file and write data on files
ifstream	This is used to read data from files
fstream	This is used to both read and write data from/to files

How to achieve File Handling

For achieving file handling in C++ we need follow following steps

- Naming a file
- Opening a file
- Reading data from file
- Writing data into file
- Closing a file

Functions use in File Handling

Function	Operation
open()	To create a file
close()	To close an existing file
get()	Read a single character from a file
put()	write a single character in file.
read()	Read data from file
write()	Write data into file.

Defining and Opening a File

The function open() can be used to open multiple files that use the same stream object.

Syntax

```
file-stream-class   stream-object;
stream-object.open("filename");
```

Example

```
ofstream  outfile;      // create stream
outfile . open ("data1");  // connect stream to data1
```

Closing a File

A file must be close after completion of all operation related to file. For closing file we need **close()** function.

Syntax

```
outfile.close();
```

File Opening mode

Mode	Meaning	Purpose
ios :: out	Write	Open the file for write only.
ios :: in	read	Open the file for read only.
ios :: app	Appending	Open the file for appending data to end-of-file.

ios :: ate	Appending	take us to the end of the file when it is opened.

Both ios :: app and ios :: ate take us to the end of the file when it is opened. The difference between the two parameters is that the ios :: app allows us to add data to the end of file only, while ios :: ate mode permits us to add data or to modify the existing data any where in the file.

The mode can combine two or more parameters using the bitwise OR operator (symbol |)

Example

```
fstream file;
file.Open("data . txt", ios :: out | ios :: in);
```

File pointer

Each file have two associated pointers known as the file pointers. One of them is called the input pointer (or get pointer) and the other is called the output pointer (or put pointer). The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.

Function for manipulation of file pointer

When we want to move file pointer to desired position then use these function for manage the file pointers.

Function	Operation
seekg()	moves get pointer (input) to a specified location.
seekp()	moves put pointer (output) to a specified location.
tellg()	gives the current position of the get pointer.
tellp()	gives the current position of the put pointer.
fout . seekg(0, ios :: beg)	go to start
fout . seekg(0, ios :: cur)	stay at current position
fout . seekg(0, ios :: end)	go to the end of file
fout . seekg(m, ios :: beg)	move to m+1 byte in the file
fout . seekg(m, ios :: cur)	go forward by m bytes from the current position
fout . seekg(-m, ios :: cur)	go backward by m bytes from the current position
fout . seekg(-m, ios :: end)	go backward by m bytes from the end

put() and get() function

The function put() write a single character to the associated stream. Similarly, the function get() reads a single character from the associated stream.

read() and write() function

These function take two arguments. The first is the address of the variable V , and the second is the length of that variable in bytes. The address of variable must be cast to type char * (i.e pointer to character type).

Example

```
file . read ((char *)&V , sizeof (V));
file . Write ((char *)&V , sizeof (V));
```

Data Writing into a File

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main() {
  ofstream ofile; // declaring an object of class ofstream
  ofile.open("info.txt"); // open "info.txt" for writing data
  /* write to a file */
 ofile << "spark institute" << endl;
  ofile << "This is fifth line" << endl;
  /* write to a console */
    cout << "Data written to file" << endl;
         ofile.close(); // close the file
  getch();
  return 0;

  }
```

Reading and Writing with File

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main() {
  char line[100];
  fstream file; // declare an object of fstream class
  file.open("info.txt", ios :: out | ios :: app); // open file in append mode
  if (file.fail()) { // check if file is opened successfully
    // file opening failed
    cout << "Error Opening file ... " << endl;   }
  else {      // proceed with further operations
    cout << "Enter a line : ";
    cin.getline(line, 100);
```

```
    file << line << endl; // Append the line to the file
    cout << "Line written into the file" << endl;   }
  getch();
  return 0;
}
```

****