Object Oriented Programming in CPP

Programming in C++

Introduction to C++

- C++ programming language developed by AT&T Bell Laboratories in 1979 by Bjarne Stroustrup. C++ is fully based on Object Oriented Technology i.e. C++ is ultimate paradigm for the modeling of information.
- C++ is the successor of C language.
- It is a case sensitive language.

Character Set- Set of characters which are recognized by c++compiler i.e
Digits (0-9), Alphabets (A-Z & a-z) and special characters + - * , . “ ‘ < > = { ( ]) space etc i.e 256 ASCII characters.

Tokens- Smallest individual unit. Following are the tokens

- **Keyword-** Reserve word having special meaning the language and can’t be used as identifier.
- **Identifiers-**Names given to any variable, function, class, union etc. Naming convention (rule) for writing identifier is as under:
  i. First letter of identifier is always alphabet.
  ii. Reserve word cannot be taken as identifier name.
  iii. No special character in the name of identifier except under score sign ‘_’.
- **Literals**-Value of specific data type assign to a variable or constant. Four type of Literals:
  i. Integer Literal i.e int x =10
  ii. Floating point Literal i.e float x=123.45
  iii. Character Literal i.e char x= ‘a’, enclosed in single quotes and single character only.
  iv. String Literal i.e cout<< “Welcome” , anything enclosed in double quotes
- **Operator** – performs some action on data
  Arithmetic(+,-,*,/,%)
  Assignment operator (=)

Increment / Decrement (++, --)
Relational/comparison (<,>,<=,>=,==,!=).
Logical(AND(&&),OR(||),NOT(!).
Conditional (? :)

Precedence of Operators:

++(post increment),--(post decrement)
++(pre increment),--(pre decrement),sizeof !(not),-(unary),+unary plus)
*(multiply), / (divide), %(modulus)
+(add),-(subtract)
<(less than),<=(less than or equal),>(greater than), >=(greater than or equal to)
==(equal),!=(not equal)
&& (logical AND)
||(logical OR)
?:(conditional expression)
=(simple assignment) and other assignment operators(arithmetic assignment operator)
, Comma operator

- **Punctuation** – used as separators in c++ e.g. [{ ( ) }] , ; # = : etc

Data type- A specifier to create memory block of some specific size and type. C++offers two types of data types:

1. Fundamental type : Which are not composed any other data type i.e. int, char, float and void
2. Derived data type : Which are made up of fundamental data type i.e array, function, class, union etc

Data type conversion- Conversion of one data type into another data type. Two type of conversion i.e

i. Implicit Conversion – It is automatically taken care by complier in the case of lower range

to higher range e.g. int x, char c='A' then x=c is valid i.e character value in c is automatically converted to integer.

ii. Explicit Conversion- It is user-defined that forces an expression to be of specific type. e.g. double x1,x2 and int res then res=int(x1+x2)

Variable- Memory block of certain size where value can be stored and changed during program execution. e.g. int x, float y, float amount, char c;

Constant- Memory block where value can be stored once but can't changed later on during program execution. e.g. const int pi =3.14;

cout – It is an object of ostream_with assign class defined in iostream.h header file and used to display value on monitor.

cin – It is an object of istream_with assign class defined in iostream.h header file and used to read value from keyboard for specific variable.

comment- Used for better understanding of program statements and escaped by the compiler to compile . e.g. – single line (//) and multi- line(/*....*/)

Cascading– Repeatedly use of input or output operators(">>" or "<<") in one statement with cin or cout.

Control Structure:

Sequence control statement(if)	**conditional statement (if else)**	**Multiple Choice Statement If –else-if**	**Switch Statement (Alternate for if else- if) works for only exact match**	**loop control statement (while ,do... while, for)**
Syntax	Syntax	Syntax	Syntax	Syntax

			switch(int / char variable)
			{
			case literal1:
			[statements
			break;]
				while(expression)
				{
				statements;
				}
				Entry control loop

if(expression)
{
statements;
}
	If(expression)
	{
	statements;
	}
	else
	{
	statements;
	}
		If (expression)
		{
		statements
		}
		else if(expression)
		{
		statement
		}
		else
		{
		statement
		}
			case literal2:
			[statements, **break**;]
			default:statements;
			}
			Break is compulsory
			
			statement with every **case** because if it is not included then the controls executes next case statement until next
			break encountered or end of **swtich** reached.
			Default is optional, it gets executed when no match is found
				works for true condition.
				do
				{
				statements;
				} **while(expression);**
				Exit Control Loop
				execute at least once if the condition is false at beginning.
				for loop
				for(expression1;expression2;expression3)
				{
				statement;}
				Entry control loop
				works for true condition
				and preferred for fixed no.of times.

Note: any non-zero value of an expression is treated as true and exactly 0 (i.e. all bits contain 0) is treated as false.

Nested loop- loop within loop.

exit()- defined in process.h and used to terminate the program depending upon certain condition.

break- exit from the current loop depending upon certain condition.

continue- to skip the remaining statements of the current loop and passes control to the next loop control statement.

goto- control is unconditionally transferred to the location of local label specified by <identifier>.

For example
A1:
cout<<"test";
goto A1;

Some Standard C++ libraries

Header	**Nome Purpose**
iostream.h	Defines stream classes for input/output streams
stdio.h	Standard input and output
ctype.h	Character tests
string.h	String operations
math.h	Mathematical functions such as sin() and cos()
stdlib.h	Utility functions such as malloc() and rand()

Some functions

- isalpha(c)-check whether the argument is alphabetic or not.
- islower(c)- check whether the argument is lowecase or not.
- isupper(c) - check whether the argument is upercase or not.
- isdigit(c)- check whether the argument is digit or not.
- isalnum(c)- check whether the argument is alphanumeric or not.
- tolower()-converts argument in lowercase if its argument is a letter.
- toupper(c)- converts argument in uppercase if its argument is a letter.
- strcat()- concatenates two string.
- strcmp-compare two string.
- pow(x,y)-return x raised to power y.
- sqrt(x)-return square root of x.
- random(num)-return a random number between 0 and (num-1)
- randomize- initializes the random number generator with a random value.

Array- Collection of element of same type that are referred by a common name.

One Dimensional array

- An array is a continuous memory location holding similar type of data in single row or single column. Declaration in c++ is as under:
  const int size =20;
  int a[size] or int a[20]. The elements of array accessed with the help of an index.
  For example : for(i=0;i<20;i++) cout<<a[i];
- **String (Array of characters)–** Defined in c++ as one dimensional array of characters as char s[80]= “Object oriented programming”;

Two-dimensional array

- A two diamensional array is a continuous memory location holding similar type of data arranged in row and column format (like a matrix structure).
  Declaration – int a[3][4], means ‘a’ is an array of integers are arranged in 3 rows & 4 columns.

Function- Name given to group of statements that does some specific task and may return a value. Function can be invoked(called) any no. of time and anywhere in the program.

Function prototypes- Function declaration that specifies the function name, return type and parameter list of the function.
syntax: return_type function_name(type var1,type var2,….,type varn);

Actual Parameters
Variables associated with function name during function call statement.

Formal Parameters
Variables which contains copy of actual parameters inside the function definition.

Local variables

- Declared inside the function only and its scope and lifetime is function only and hence accessible only inside function.

Global variables

- Declared outside the function and its scope and lifetime is whole program and hence accessible to all function in the program from point declaration.
  Example:
  #include <iostream.h>
  int a=20; // global
  void main()
  {
  int b=10; // local
  cout<<a<<b;
  }

Passing value to function-

- **Passing by value-**In this method separate memory created for formal arguments and if any changes done on formal variables, it will not affect the actual variables. So actual variables are preserved in this case
- **Passing by address/reference-**In this method no separate memory created for formal variables i.e formal variables share the same location of actual variables and hence any change on formal variables automatically reflected back to actual variables.

Example :

```
void sample( int a, int &b)
{
a=a+100;
b=b+200;
cout<<a<<b;
}
void main()
{
int a=50, b=40;
cout<<a<<b; // output 50 40
sample(a,b) // output 150 240
cout<<a<<b; // output 50 240
}
```

Function overloading

- Processing of two or more functions having same name but different list of parameters

Function recursion

- Function that call itself either directly or indirectly.

Structure-Collection of logically related different data types (Primitive and Derived) referenced under one name.

e.g. struct employee
{
int empno;
char name[30];
char design[20];
char department[20];
}

Declaration: employee e;

Input /Output : cin>>e.empno; // members are accessed using dot(.) operator.
cout<<e.empno;

Nested structure

- A Structure definition within another structure.
- A structure containing object of another structure.

e.g. struct address
{ int houseno;
char city[20];
char area[20];
long int pincode;}
struct employee
{
int empno;

char name[30];
char design[20];
char department[20];
address ad; // **nested structure**
}
Declaration: employee e;
Input /Output : cin>>e.ad.houseno; // members are accessed using dot(.) operator.
cout<<e.ad.houseno;

typedef
Used to define new data type name.
e.g. typedef char Str80[80]; Str80 str;

#define Directives

- Use to define a constant number or macro or to replace an instruction.

Function overloading in C++

- A function name having several definitions that are differentiable by the number or types of their arguments is known as function overloading.

Example : A same function print() is being used to print different data types:

```
#include <iostream.h>
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 following result:

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

Object Oriented Programming Concepts

Object Oriented Programming follows bottom up approach in program design and emphasizes on safety and security of data.

FEATURES OF OBJECT ORIENTED PROGRAMMING:

Inheritance:

- Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class.
- Derived class is also known as a child class or sub class. Inheritance helps in reusability of code , thus reducing the overall size of the program

Data Abstraction:

- It refers to the act of representing essential features without including the background details .Example : For driving , only accelerator, clutch and brake controls need to be learnt rather than working of engine and other details.

Data Encapsulation:

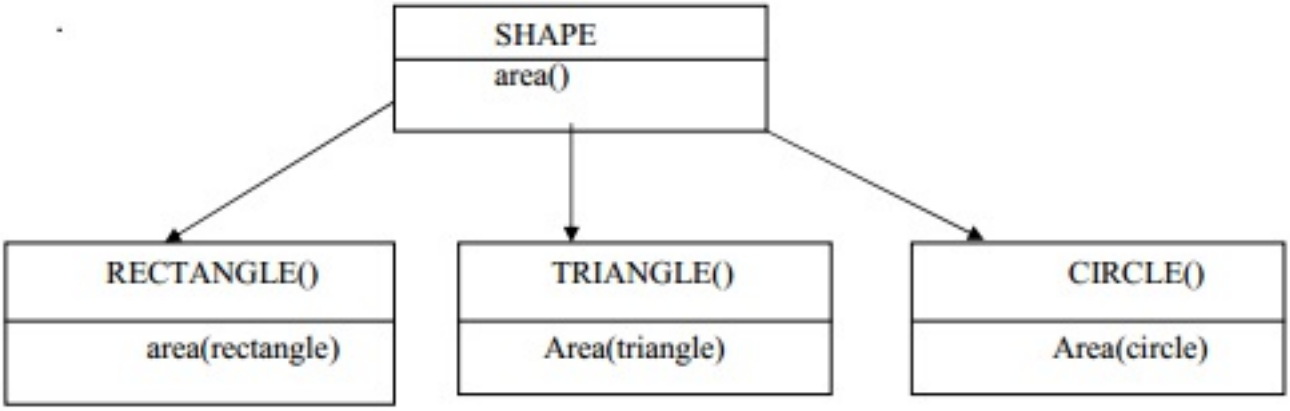
- It means wrapping up data and associated functions into one single unit called class..
- A class groups its members into three sections :public, private and protected, where private and protected members remain hidden from outside world and thereby helps in implementing data hiding.

Modularity :

- The act of partitioning a complex program into simpler fragments called modules is called as modularity.
- It reduces the complexity to some degree and
- It creates a number of well defined boundaries within the program.

Polymorphism:

- Poly means many and morphs mean form, so polymorphism means one name multiple forms.
- It is the ability for a message or data to be processed in more than one form.
- C++ implements Polymorhism through Function Overloading, Operator overloading and Virtual functions.



Objects and Classes:

The major components of Object Oriented Programming are . Classes & Objects

A Class is a group of similar objects . Objects share two characteristics: They all have state and behavior. For example : Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, applying brakes). Identifying the state and behavior for real world objects is a great way to begin thinking in terms of object-oriented programming. These real-world observations all translate into the world of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through functions

Classes in Programming :

- It is a collection of variables, often of different types and its associated functions.
- Class just binds data and its associated functions under one unit there by enforcing encapsulation.
- Classes define types of data structures and the functions that operate on those data

structures.

- A class defines a blueprint for a data type.

Declaration/Definition :

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.

class class_name
{
access_specifier_1:
member1;
access_specifier_2:
member2;
...
} object_names;

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members that can either be data or function declarations, and optionally access specifiers.

[Note: the default access specifier is private.

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

Access specifiers in Classes:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

- private

- public
- protected

Member-Access Control

Type of Access	Meaning
Private	Class members declared as **private** can be used only by member functions and friends (classes or functions) of the class.
Protected	Class members declared as **protected** can be used by member functions and friends (classes or functions) of the class. Additionally, they can be used by classes derived from the class.
Public	Class members declared as **public** can be used by any function.

- Importance of Access Specifiers

Access control helps prevent you from using objects in ways they were not intended to be used. Thus it helps in implementing data hiding and data abstraction.

OBJECTS in C++:

Objects represent instances of a class. Objects are basic run time entities in an object oriented system.

Creating object / defining the object of a class:

The general syntax of defining the object of a class is:-

Class_name object_name;

In C++, a class variable is known as an object. The declaration of an object is similar to that of a variable of any data type. The members of a class are accessed or referenced using object of a class.

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.

Accessing / calling members of a class All member of a class are private by default.

Private member can be accessed only by the function of the class itself.Public member of a class can be accessed through any object of the class. They are accessed or called using object of that class with the help of dot operator (.).
The general syntax for accessing data member of a class is:-
Object_name.Data_member=value;
The general syntax for accessing member function of a class is:-
Object_name. Function_name (actual arguments);
The dot ('. ') used above is called the **dot operator or class member access operator**. The dot operator is used to connect the object and the member function. The private data of a class can be accessed only through the member function of that class.

Class methods definitions (Defining the member functions)

Member functions can be defined in two places:-

- Outside the class definition

The member functions of a class can be defined outside the class definitions. It is only declared inside the class but defined outside the class. The general form of member function definition outside the class definition is:

Return_type Class_name:: function_name (argument list)
{
Function body
}
Where symbol :: is a scope resolution operator.

The scope resolution operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition

class sum
{

```
int A, B, Total;
public:
void getdata ();
void display ();
};
void sum:: getdata () // Function definition outside class definition Use of :: operator
{
cout<<" \n enter the value of A and B";
cin>>A>>B;
}
void sum:: display () // Function definition outside class definition Use of :: operator
{
Total =A+B;
cout<<"\n the sum of A and B="<<Total;
}
```

- Inside the class definition

The member function of a class can be declared and defined inside the class definition.

```
class sum
{
int A, B, Total;
public:
void getdata ()
{
cout< "\n enter the value of A and B";
cin>>A>>B;
}
void display ()
{
total = A+B;
cout<<"\n the sum of A and B="<<total;
}
};
```

Differences between struct and classes in C++

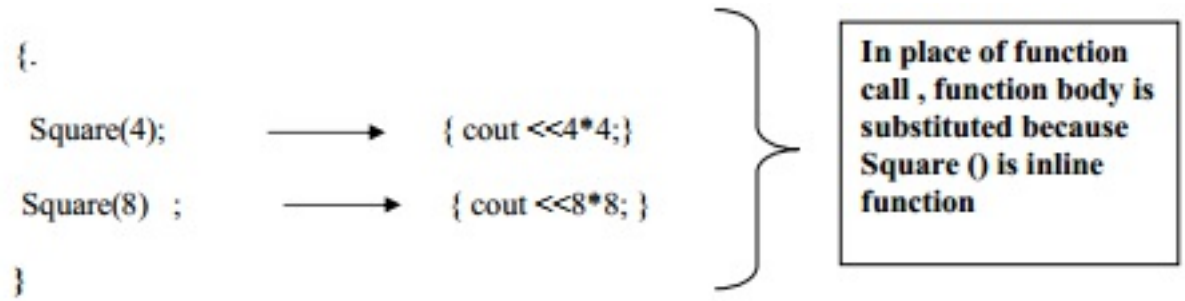
In C++, a structure is a class defined with the struct keyword.Its members and base classes are public by default. A class defined with the class keyword has private members and base classes by default. This is the only difference between structs and classes in C++.

INLINE FUNCTIONS

- Inline functions definition starts with keyword inline
- The compiler replaces the function call statement with the function code itself(expansion) and then compiles the entire code.
- They run little faster than normal functions as function calling overheads are saved.
- A function can be declared inline by placing the keyword inline before it.

Example

```
inline void Square (int a)
{ cout<<a*a;}
void main()
```



Pass Object As An Argument

/*C++ PROGRAM TO PASS OBJECT AS AN ARGUMENT. The program Adds the two heights given in feet and inches. */

```
#include< iostream.h>
#include< conio.h>
class height
{
int feet,inches;
```

```
public:
void getht(int f,int i)
{
feet=f;
inches=i;
}
void putheight()
{
cout< < "\nHeight is:"< < feet< < "feet\t"< < inches< < "inches"< < endl;
}
void sum(height a,height b)
{
height n;
n.feet = a.feet + b.feet;
n.inches = a.inches + b.inches;
if(n.inches ==12)
{
n.feet++;
n.inches = n.inches -12;
}
cout< < endl< < "Height is "< < n.feet< < " feet and "< < n.inches< < endl;
}};
void main()
{height h,d,a;
clrscr();
h.getht(6,5);
a.getht(2,7);
h.putheight();
a.putheight();
d.sum(h,a);
getch();
}
/**********OUTPUT***********
```

Height is: 6 feet 5 inches
Height is: 2 feet 7 inches
Height is 9 feet and 0

CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS :

A member function with the same as its class is called Constructor and it is used to initialize the object of that class with a legal initial value.

Example:

```
class Student
{
int rollno;
float marks;
public:
student( ) //Constructor
{
rollno=0;
marks=0.0;
}
//other public members
};
```

TYPES OF CONSTRUCTORS:

1. **Default Constructor:**
   A constructor that accepts no parameter is called the Default Constructor. If you don't declare a constructor or a destructor, the compiler makes one for you. The default constructor and destructor take no arguments and do nothing.
2. **Parameterized Constructors:**
   A constructor that accepts parameters for its invocation is known as parameterized Constructors, also called as Regular Constructors.

DESTRUCTORS:

- A destructor is also a member function whose name is the same as the class name but is preceded by tilde(“~”).It is automatically by the compiler when an object is destroyed. Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed.
- A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

Example :

```
class TEST
{ int Regno,Max,Min,Score;
Public:
TEST( ) // Default Constructor
{ }
TEST (int Pregno,int Pscore) // Parameterized Constructor
{
Regno = Pregno ;Max=100;Max=100;Min=40;Score=Pscore;
}
~ TEST ( ) // Destructor
{ Cout<<”TEST Over”<<endl;}
};
```

The following points apply to constructors and destructors:

- Constructors and destructors do not have return type, not even void nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword virtual.
- Constructors and destructors cannot be declared static, const, or volatile.
- Unions cannot contain class objects that have constructors or destructors.
- The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope.
- Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes.

- The default destructor calls the destructors of the base class and members of the derived class.
- The destructors of base classes and members are called in the reverse order of the completion of their constructor:
- The destructor for a class object is called before destructors for members and bases are called.

Copy Constructor

- A copy constructor is a special constructor in the C++ programming language used to create a new object as a copy of an existing object.
- A copy constructor is a constructor of the form classname(classname &).The compiler will use the copy constructors whenever you initialize an instance using values of another instance of the same type.
- Copying of objects is achieved by the use of a copy constructor and a assignment operator.

Example :

```
class Sample{ int i, j;}
public:
Sample(int a, int b) // constructor
{ i=a;j=b;}
Sample (Sample & s) //copy constructor
{ j=s.j ; i=s.j;
Cout <<"\n Copy constructor working \n";
}
void print (void)
{cout <<i<< j<< "\n";}
:
};
```

Note: The argument to a copy constructor is passed by reference, the reason being that when an argument is passed by value, a copy of it is constructed. But the copy constructor is creating a copy of the object for itself, thus ,it calls itself. Again the called copy constructor

requires another copy so again it is called.in fact it calls itself again and again until the compiler runs out of the memory .so, in the copy constructor, the argument must be passed by reference.

The following cases may result in a call to a copy constructor:

- When an object is passed by value to a function:

The pass by value method requires a copy of the passed argument to be created for the function to operate upon. Thus to create the copy of the passed object, copy constructor is invoked
If a function with the following prototype :
void cpyfunc(Sample); // Sample is a class
then for the following function call
cpyfunc(obj1); // obj1 is an object of Sample type
the copy constructor would be invoked to create a copy of the obj1 object for use by cpyfunc().

- When a function returns an object :

When an object is returned by a function the copy constructor is invoked

Sample cpyfunc(); // Sample is a class and it is return type of cpyfunc()
If func cpyfunc() is called by the following statement
obj2 = cpyfunc();
Then the copy constructor would be invoked to create a copy of the value returned by cpyfunc() and its value would be assigned to obj2. The copy constructor creates a temporary object to hold the return value of a function returning an object.

INHERITANCE

- Inheritance is the process by which new classes called derived classes are created from existing classes called base classes.
- The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.
- The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Features or Advantages of Inheritance:

- Reusability of Code
- Saves Time and Effort
- Faster development, easier maintenance and easy to extend
- Capable of expressing the inheritance relationship and its transitive nature which ensures closeness with real world problems.

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. A class derivation list names one or more base classes and has the form:

class derived-class: access-specifier base-class
Where access is one of public, protected, or private.
For example, if the base class is MyClass and the derived class is sample it is specified as:
class sample: public MyClass
The above makes sample have access to both public and protected variables of base class MyClass.

EXAMPLE OF SINGLE INHERITANCE

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

```
// Base class
class Shape
{
public:
void setWidth(int w)
{
width = w;
}
void setHeight(int h)
{
height = h;
}
```

```
protected:
int width;
int height;
};
// Derived class
class Rectangle: public Shape
{
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 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:

Access	**public**	**protected**	**private**

Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

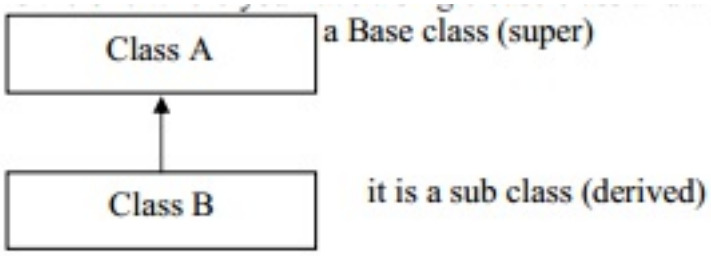
When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. We hardly use protected or private inheritance but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

1. **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.
2. **Protected Inheritance:** When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
3. **Private Inheritance:** When deriving from a private base class, public and protected members of the base class become private members of the derived Class.

TYPES OF INHERITANCE

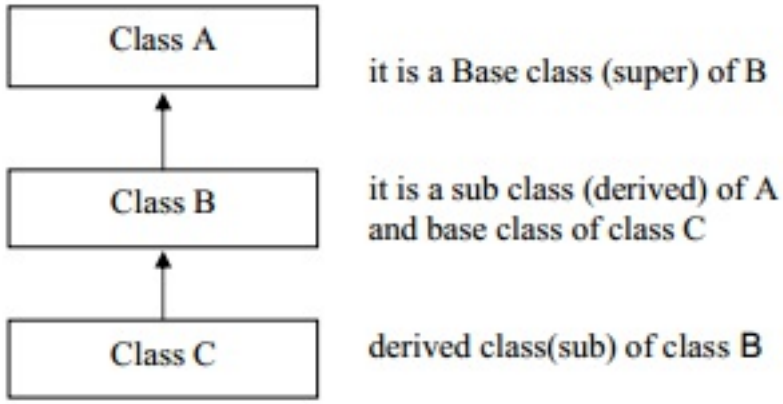
1. **Single class Inheritance:**

   - Single inheritance is the one where you have a single base class and a single derived class.



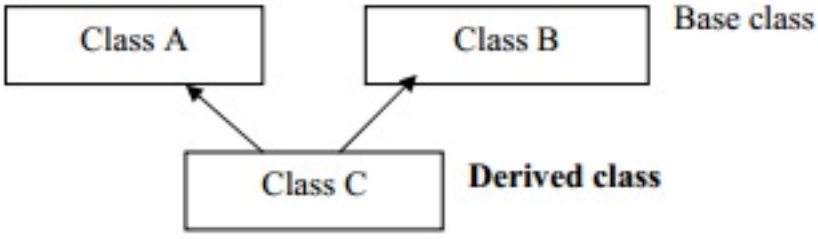
2. **Multilevel Inheritance:**

- In Multi level inheritance, a subclass inherits from a class that itself inherits from another class.



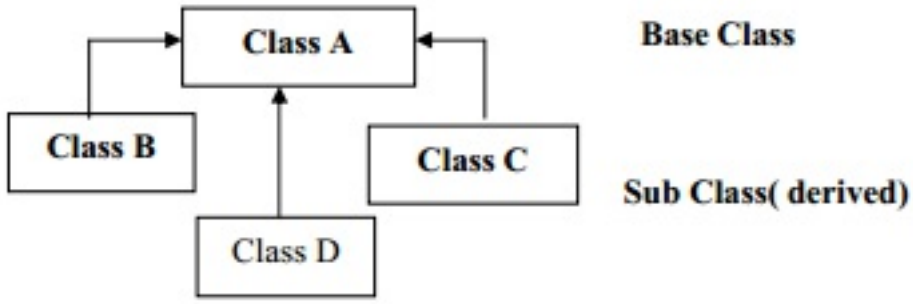
3. **Multiple Inheritance:**

- In Multiple inheritances, a derived class inherits from multiple base classes. It has properties of both the base classes.



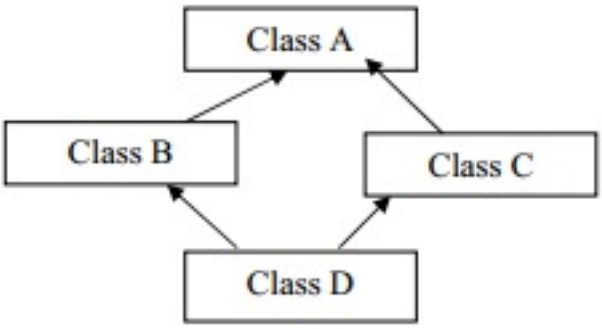
4. **Hierarchical Inheritance:**

- In hierarchial Inheritance, it's like an inverted tree. So multiple classes inherit from a single base class.



5. **Hybrid Inheritance:**

- It combines two or more forms of inheritance .In this type of inheritance, we can have mixture of number of inheritances but this can generate an error of using same name function from no of classes, which will bother the compiler to how to use the functions.
- Therefore, it will generate errors in the program. This has known as ambiguity or duplicity.
- Ambiguity problem can be solved by using virtual base classes



Pointers

- Pointer is a variable that holds a memory address of another variable of same type.
- It supports dynamic allocation routines.
- It can improve the efficiency of certain routines.

C++ Memory Map :

- Program Code : It holds the compiled code of the program.
- Global Variables : They remain in the memory as long as program continues.
- Stack : It is used for holding return addresses at function calls, arguments passed to the functions, local variables for functions. It also stores the current state of the CPU.
- Heap : It is a region of free memory from which chunks of memory are allocated via DMA functions.

Static Memory Allocation : The amount of memory to be allocated is known in advance and it allocated during compilation, it is referred to as Static Memory Allocation.
e.g. int a; // This will allocate 2 bytes for a during compilation.

Dynamic Memory Allocation : The amount of memory to be allocated is not known beforehand rather it is required to allocated as and when required during runtime, it is referred to as dynamic memory allocation.
C++ offers two operator for DMA – new and delete.
e.g int x =new int; float y= new float; // dynamic allocation
delete x; delete y; //dynamic deallocation

Free Store : It is a pool of unallocated heap memory given to a program that is used by the program for dynamic memory allocation during execution.

Declaration and Initialization of Pointers :
Datatype *variable_name;
Syntax : Datatype *variable_name;
Int *p; float *p1; char *c;
Eg. Int *p; float *p1; char *c;

Two special unary operator * and & are used with pointers. The & is a unary operator that returns the memory address of its operand.

Eg. Int a = 10; int *p; p = &a;

Pointer arithmetic:
Two arithmetic operations, addition and subtraction, may be performed on pointers. When you add 1 to a pointer, you are actually adding the size of whatever the pointer is pointing at. That is, each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.
e.g. int *p; P++;
If current address of p is 1000, then p++ statement will increase p to 1002, not 1001.
If *c is char pointer and *p is integer pointer then

Char pointer	**C**	**c+1**	**c+2**	**c+3**	**c+4**	**c+5**	**c+6**	**c+7**
Address	**100**	**101**	**102**	**103**	**104**	**105**	**106**	**107**
Int Pointer	**p**		**P+1**		**P+2**		**P+3**	

Adding 1 to a pointer actually adds the size of pointer's base type.

Base address : A pointer holds the address of the very first byte of the memory location where it is pointing to. **The address of the first byte is known as BASE ADDRESS.**

Dynamic Allocation Operators :

C++ dynamic allocation allocate memory from the free store/heap/pool, the pool of unallocated heap memory provided to the program. C++ defines two unary operators new and delete that perform the task of allocating and freeing memory during runtime.

Creating Dynamic Array :
Syntax : pointer-variable = new data-type [size];
e.g. int * array = new int[10];
Not array[0] will refer to the first element of array, array[1] will refer to the second element.
No initializes can be specified for arrays.
All array sizes must be supplied when new is used for array creation.

Two dimensional array :

```
int *arr, r, c;
r = 5; c = 5;
arr = new int [r * c];
```

Now to read the element of array, you can use the following loops :

```
For (int i = 0; i < r; i++)
{
cout << "\n Enter element in row " << i + 1 << " : ";
For (int j=0; j < c; j++)
cin >> arr [ i * c + j];
}
```

Memory released with delete as below:

Syntax for simple variable : **delete pointer-variable;** **eg. delete p;**	**For array :** **delete [size] pointer variable;** **Eg. delete [ ] arr;**

Pointers and Arrays :

C++ treats the name of an array as constant pointer which contains base address i.e address of first location of array. Therefore Pointer variables are efficiently used with arrays for declaration as well as accessing elements of arrays, because array is continuous block of same memory locations and therefore pointer arithmetic help to traverse in the array easily.

```
void main()
{
int *m;
int marks[10] ={ 50,60,70,80,90,80,80,85,75,95};
m = marks; // address of first location of array or we can write it as m=&marks[0]
for(int i=0;i<10;i++)
cout<< *m++;
// or
m = marks; // address of first location of array or we can write it as m=&marks[0]
```

```
for(int i=0;i<10;i++)
cout<< *(m+i);
}
```

Array of Pointers :

To declare an array holding 10 int pointers –
int * ip[10];
That would be allocated for 10 pointers that can point to integers.

Now each of the pointers, the elements of pointer array, may be initialized. To assign the address of an integer variable phy to the forth element of the pointer array, we have to write ip[3] = & phy;

Now with *ip[3], we can find the value of phy. int *ip[5];

Index	0	1	2	3	4
address	1000	1002	1004	1006	1008

int a = 12, b = 23, c = 34, d = 45, e = 56;

Variable	a	b	c	d	e
Value	12	23	34	45	56
address	1050	1065	2001	2450	2725

ip[0] = &a; ip[1] = &b; ip[2] = &c; ip[3] = &d; ip[4] = &e;

Index	ip[0]	ip[1]	ip[2]	ip[3]	ip[4]
Array ip value	1050	1065	2001	2450	2725
address	1000	1002	1004	1006	1008

ip is now a pointer pointing to its first element of ip. Thus ip is

equal to address of ip[0], i.e. 1000
*ip (the value of ip[0]) = 1050
* (* ip) = the value of *ip = 12
* * (ip+3) = * * (1006) = * (2450) = 45

Pointers and Strings :
Pointer is very useful to handle the character array also. E.g :

```
void main()
{ char str[] = “computer”;
char *cp;
cp=str;
cout<<str ; //display string
cout<<cp; // display string
for (cp =str; *cp != ‘\0’; cp++) // display character by character by character
cout << ”--“<<*cp;
// arithmetic
str++; // not allowed because str is an array and array name is constant pointer
cp++; // allowed because pointer is a variable
cout<<cp;}
```

Output :

```
Computer
Computer
--c--o--m--p--u--t--e—r
computer
```

An array of char pointers is very useful for storing strings in memory. Char *subject[] = { “Chemistry”, “Physics”, “Maths”, “CS”, “English” };

In the above given declaration subject[] is an array of char pointers whose element pointers contain base addresses of respective names. That is, the element pointer subject[0] stores the base address of string “Chemistry”, the element pointer subject[1] stores the above address of string “Physics” and so forth.

An array of pointers makes more efficient use of available memory by consuming lesser

number of bytes to store the string.

An array of pointers makes the manipulation of the strings much easier. One can easily exchange the positions of strings in the array using pointers without actually touching their memory locations.

Pointers and CONST :

A constant pointer means that the pointer in consideration will always point to the same address. Its address can not be modified.

A pointer to a constant refers to a pointer which is pointing to a symbolic constant. Look the following example :

```
int m = 20; // integer m declaration
int *p = &m; // pointer p to an integer m
++ (*p); // ok : increments int pointer p
int * const c = &n; // a const pointer c to an intger n
++ (* c); // ok : increments int pointer c i.e. its contents
++ c; // wrong : pointer c is const – address can't be modified
const int cn = 10; // a const integer cn
const int *pc = &cn; // a pointer to a const int
++ (* pc); // wrong : int * pc is const – contents can't be modified
++ pc; // ok : increments pointer pc
const int * const cc = *k; // a const pointer to a const integer
++ (* cc); // wrong : int *cc is const
++ cc; // wrong : pointer cc is const
```

Pointers and Functions :

A function may be invoked in one of two ways :

1. call by value
2. call by reference

The second method call by reference can be used in two ways :

1. by passing the references

2. by passing the pointers

Reference is an alias name for a variable. For ex : int m = 23;
int &n = m;
int *p;
p = &m;
Then the value of m i.e. 23 is printed in the following ways : cout <<
m; // using variable name
cout << n; // using reference name
cout << *p; // using the pointer

Invoking Function by Passing the References :

When parameters are passed to the functions by reference, then the formal parameters become references (or aliases) to the actual parameters to the calling function.

That means the called function does not create its own copy of original values, rather, it refers to the original values by different names i.e. their references.

For example the program of swapping two variables with reference method :

```
#include<iostream.h>
void main()
{
void swap(int &, int &);
int a = 5, b = 6;
cout << "\n Value of a :" << a << " and b :" << b;
swap(a, b);
cout << "\n After swapping value of a :" << a << "and b :" << b;
}
Void swap(int &m, int &n)
{
int temp; temp = m;
m = n;
n = temp;
}
```

output :
Value of a : 5 and b : 6
After swapping value of a : 6 and b : 5

Invoking Function by Passing the Pointers:
When the pointers are passed to the function, the addresses of actual arguments in the calling function are copied into formal arguments of the called function.
That means using the formal arguments (the addresses of original values) in the called function, we can make changing the actual arguments of the calling function.
For example the program of swapping two variables with Pointers :

```
#include<iostream.h>
void main()
{
void swap(int *m, int *n);
int a = 5, b = 6;
cout << "\n Value of a :" << a << " and b :" << b;
swap(&a, &b);
cout << "\n After swapping value of a :" << a << "and b :" << b;
}
void swap(int *m, int *n)
{
int temp;
temp = *m;
*m = *n;
*n = temp;
}
```

Input :
Value of a : 5 and b : 6
After swapping value of a : 6 and b : 5

Function returning Pointers :

The way a function can returns an int, an float, it also returns a pointer. The general form of prototype of a function returning a pointer would be

Type * function-name (argument list);

```
#include <iostream.h> int
*min(int &, int &); void main()
{
int a, b, *c;
cout << “\nEnter a :”; cin >> a;
cout << “\nEnter b :”; cint >> b;
c = min(a, b);
cout << “\n The minimum no is :” << *c;
}
int*min(int &x, int &y)
{
if (x < y )
return (&x);
else
return (&y)
}
```

Dynamic structures :

The new operator can be used to create dynamic structures also i.e. the structures for which the memory is dynamically allocated.

```
struct-pointer = new struct-type;
student *stu;
stu = new Student;
```

A dynamic structure can be released using the deallocation operator delete as shown below:

```
delete stu;
```

Objects as Function arguments:

Objects are passed to functions in the same way as any other type of variable is passed. When it is said that objects are passed through the call-by-value, it means that the called function creates a copy of the passed object.

A called function receiving an object as a parameter creates the copy of the object without

invoking the constructor. However, when the function terminates, it destroys this copy of the object by invoking its destructor function.

If you want the called function to work with the original object so that there is no need to create and destroy the copy of it, you may pass the reference of the object. Then the called function refers to the original object using its reference or alias.

Also the object pointers are declared by placing in front of a object pointer's name.
Classname * object-pointer;

Eg. Student *stu;

The member of a class is accessed by the arrow operator (->) in object pointer method.

Eg :

```
#include<iostream.h>
class Point
{
int x, y;
public :
Point()
{x = y = 0;}
void getPoint(int x1, int y1)
{x = x1; y = y1; }
void putPoint()
{
cout << "\n Point : (" << x << ", " << y << ")";
}};
void main()
{
Point p1, *p2;
cout << "\n Set point at 3, 5 with object";
p1.getPoint(3,5);
cout << "\n The point is :";
p1.putPoint();
```

```
p2 = &p1;
cout << "\n Print point using object pointer :";
p2->putPoint();
cout << "\n Set point at 6,7 with object pointer";
p2->getPoint(6,7);
cout<< "\n The point is :";
p2->putPoint();
cout << "\n Print point using object :";
p1.getPoint();}
```

If you make an object pointer point to the first object in an array of objects, incrementing the pointer would make it point to the next object in sequence.

```
student stud[5], *sp;
---
sp = stud; // sp points to the first element (stud[0])of stud
sp++; // sp points to the second element (stud[1]) of stud sp + = 2;
// sp points to the fourth element (stud[3]) of stud sp--; // sp
points to the third element (stud[2]) of stud
```

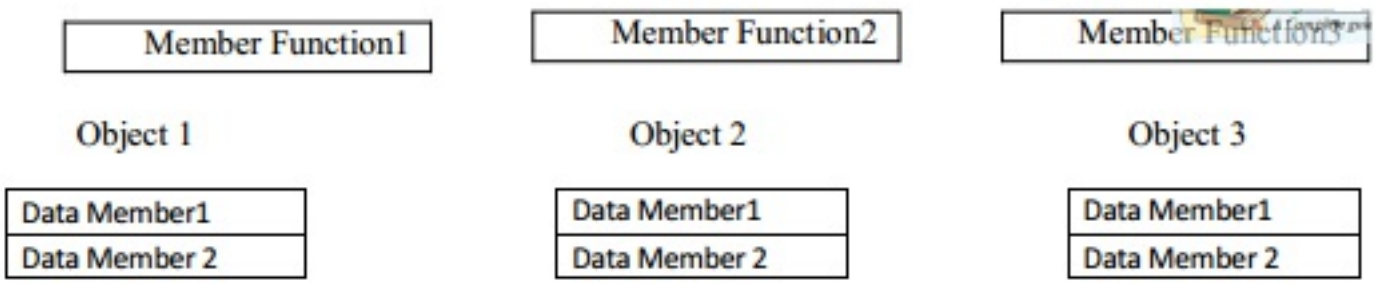
You can even make a pointer point to a data member of an object. Two points should be considered :

1. A Pointer can point to only public members of a class.
2. The data type of the pointer must be the same as that of the data member it points to.

This Pointer :

In class, the member functions are created and placed in the memory space only once. That is only one copy of functions is used by all objects of the class.

Therefore if only one instance of a member function exists, how does it come to know which object's data member is to be manipulated?



For the above figure, if Member Function2 is capable of changing the value of Data Member3 and we want to change the value of Data Member3 of Object3. How would the Member Function2 come to know which Object's Data Member3 is to be changed?
To overcome this problem this pointer is used.
When a member function is called, it is automatically passed an implicit argument that is a pointer to the object that invoked the function. This pointer is called This.
That is if ojbect3 is invoking member function2, then an implicit argument is passed to member function2 that points to object3 i.e. this pointer now points to object3.
The friend functions are not members of a class and, therefore, are not passed a this pointer.
The static member functions do not have a this pointer.