OOPs(Object Oriented Programmig Concepts) with C++

Introduction to Object oriented programming

C++ Introduction

Namespace

Class and object

Operations

Exception handling

Memory management

C++ Language Feature

Inheritance

Virtual Function

Advance C++

Limitations of C Programming

- C is said to be process oriented, structured programming language.
- When program becomes complex, understating and maintaining such programs is very difficult.
- Language don’t provide security for data.
- Using functions we can achieve code reusability, but reusability is limited. The programs are not extendible.

Few Real Time Applications of C++

- Games
- GUI Based Application (Adobe)
- Database Software (MySQL Server)
- OS (Apple OS)
- Browser(Mozilla)
- Google Applications(Google File System and Chrome browser)
- Banking Applications
- Compilers
- Embedded Systems(smart watches, MP3 players, GPS systems)

Characteristics of Language

1. It has own syntax
2. It has its own rule(semantics)
3. It contain tokens:
   - Identifier
   - Keyword
   - Constant/literal
   - Operator
   - Seperator / punctuators
4. It contains built in features.
5. We use language to develop application(CUI, GUI, Library)

SDK (Software Development Kit)

- SDK = Language tools + Documentation + Supporting Library + Runtime Environment
- Language Tools:
  - Editor (to develop/edit source code)
  - Pre-processor (To remove pre-processors-remove comments/expand macros)
  - Compiler (Conversion of high level language into low level code(Assembly))
  - Assembler (Conversion of low level code into machine code)
  - Linker
  - Loader
  - Debugger
- # Documentation (MSDN / man pages)

Classification of high level Languages

- Procedure Oriented Programming language(POP)
  - ALGOL, FORTRAN, PASCAL, BASIC, C etc.
  - "FORTRAN" is considered as first high level POP language.
  - All POP languages follows "TOP Down" approach

- Object oriented programming languages(OOP)
  - Simula, Smalltalk, C++, Java, C#, Python, Go
  - "Simula" is considered as first high level OOP language.
  - more than 2000 lang. are OO.
  - All OOP languages follows "Bottom UP" approach

History of C++

- OOPS is not a syntax.
- It is a process / programming methodology which is used to solve real world problems.

  - Inventor of C++ is Bjarne Stroustrup.
  - C++ is derived from C and simula.
  - Its initial name was "C With Classes".
  - At is developed in "AT&T Bell Lab" in 1979.
  - It is developed on Unix Operating System.
  - In 1983 ANSI renamed "C With Classes" to C++.
  - C++ is objet oriented programming language

OOPS(Object Oriented Programming Language)

- It is a programming methodology to organize complex program in to simple program in terms of classes and object such methodology is called oops.
- It is a programming methodology to organized complex program into simple program by using concept of abstraction , encapsulation , polymorphism and inheritance.
- Languages which support abstraction , encapsulation polymorphism and inheritance are called oop language.

Major pillars of oops

- **Abstraction**
  - getting only essential things and hiding unnecessary details is called as abstraction.
  - Abstraction always describe outer behavior of object.
  - In console application when we give call to function in to the main function , it represents the abstraction
- **Encapsulation**
  - binding of data and code together is called as encapsulation.
  - Implementation of abstraction is called encapsulation.
  - Encapsulation always describe inner behavior of object
  - Function call is abstraction
  - Function definition is encapsulation.
  - Information hiding
    - Data : unprocessed raw material is called as data.
    - Process data is called as information.
    - Hiding information from user is called information hiding.
    - In c++ we used access Specifier to provide information hiding.
- **Modularity**
  - Dividing programs into small modules for the purpose of simplicity is called modularity.
- **Hierarchy (Inheritance [is-a] , Composition [has-a] , Aggregation[has-a], Dependancy)**
  - Hierarchy is ranking or ordering of abstractions.

Minor pillars of oops

- **Polymorphism (Typing)**
  - One interface having multiple forms is called as polymorphism.
  - Polymorphism have two types
    1. **Compile time polymorphism**

       when the call to the function resolved at compile time it is called as compile time polymorphism. And it is achieved by using function overloading and operator overloading
    1. **Runtime polymorphism.**

       when the call to the function resolved at run time it is called as run time polymorphism. And it is achieved by using function overriding.
  - Compile time / Static polymorphism / Static binding / Early binding / Weak typing / False Polymorphism
  - Run time / Dynamic polymorphism / Dynamic binding / Late binding / Strong typing / True polymorphism
- Concurrency
  - The concurrency problem arises when multiple threads simultaneously access same object.
  - You need to take care of object synchronization when concurrency is introduced in the system.
- Persistence
  - It is property by which object maintains its state across time and space.
  - It talks about concept of serialization and also about transferring object across network.

Data Types in C++

- It describes 3 things about variable / object

1. Memory : How much memory is required to store the data.
2. Nature : Which type of data memory can store
3. Operation : Which operations are allowed to perform on data stored inside memory.

- Fundamental Data Types (void, int,char,float,double)

-Derived Data Types (Array, Function, Pointer, Union ,Structure)

Two more additional data types that c++ supports are

1. ***bool* :-** it can take *true* or *false* value. It takes one byte in memory.
2. **wchar_t :-** it can store 16 bit character. It takes 2 bytes in memory.

Bool and wchar_t

- **e.g: bool val=true;**
- **wchar_t**: Wide Character. This should be avoided because its size is implementation defined and not reliable.
- Wide char is similar to char data type, except that wide char take up twice the space and can take on much larger values as a result. char can take 256 values which corresponds to entries in the ASCII table. On the other hand, wide char can take on 65536 values which corresponds to UNICODE values which is a recent international standard which allows for the encoding of characters for virtually all languages and commonly used symbols.

- The type for character constants is char, the type for wide character is wchar_t.
- This data type occupies 2 or 4 bytes depending on the compiler being used.
- Mostly the wchar_t datatype is used when international languages like Japanese are used.
- This data type occupies 2 or 4 bytes depending on the compiler being used.
- L is the prefix for wide character literals and wide-character string literals which tells the compiler that that the char or string is of type wide-char.
- w is prefixed in operations like scanning (**wcin)** or printing (**wcout)** while operating wide-char type.

Main Function

- main should be entry point function of C/C++
- Calling/invoking main function is responsibility of operating system.
- Hence it is also called as Callback function

Structure

- Structure is a collection of similar or dissimilar data. It is used to bind logically related data into a single unit.
- This data can be modified b any function to which the structure is passed.
- Thus there is no security provided for the data within a structure.
- This concept is modified by C++ to bind data as well as functions.

Access Specifier

- By default all members in structure are accessible everywhere in the program by dot(.) or arrow(→) operators.
- But such access can be restricted by applying access specifiers

  private: Accessible only within the struct

  public: Accessible within & outside struct

Structure in C & C++

struct in c	struct in c ++
we can include only variables into the structure.	we can include the variables as well as the functions in structure.
We need to pass a structure variable by value or by address to the functions.	We don't pass the structure variable to the functions to accept it / display it.The functions inside the struct are called with the variable and DOT operator.
By default all the variables of structure are accessible outside the structure. (using structure variable name)	By default all the members are accessible outside the structure, but we can restrict their access by applying the keywords private /public/ protected.
struct Time t1;	struct Time t1;
AcceptTime(struct Time &t1);	t1.AcceptTime(); //function call

Structure in C & C++

```
struct time {
   int hr, min, sec;
};
void display( struct time *p) {
   printf("%d:%d:%d", p→hr,
   p→min, p→sec);
}
struct time t;
display(&t);
```

```
struct time {
   int hr, min, sec;
void display(){
   printf("%d:%d:%d",
this→hr, this→min, this→sec);
}
};
time t;
t.display();
```

OOP and POP

OOP (Object Oriented Programming)	**POP (**Procedural Oriented Programming**)**
Emphasis on data of the program	Emphasis on steps or algorithm
OOP follows bottom up approach.	OOP follows top down approach.
A program is divided to objects and their interactions. Programs are divide into small data units i.e. classes	A program is divided into funtions and they interacts. Programs are divided into small code units i.e. functions
Objects communicates with each other by passing messeges.	Functions communicate with each other by passing parameters.
Inheritance is supported.	Inheritance is not supported.
Access control is supported via access modifiers. (private/ public/ protected)	No access modifiers are supported.
Encapsulation is used to hide data.	No data hiding present. Data is globally accessible.
C++, Java	C , Pascal
It overloads functions, constructors, and operators.	Neither it overload functions nor operators
Classes or function can become a friend of another class with the keyword "friend". Note: "**friend**" keyword is used only in c++	No concept of friend function.
Concept of virtual function appear during inheritance.	No concept of virtual classes .

Namespace

- To prevent name conflicts/ collision / ambiguity in large projects
- to group/orgaize functionally equivalent / related types toghther.
- If we want to access value of global variable then we should use scope resolution operator (::)
- We can not instantiate namespace.
- It is designed to avoid name ambiguity and grouping related types.
- If we want to define namespace then we should use **namespace** keyword.
- We can not define namespace inside function/class.
- If name of the namespaces are same then name of members must be different.
- We can not define main function inside namespace.
- Namespace can contain:
  1. Variable
  2. Function
  3. Types[structure/union/class]
  4. Enum
  5. Nested Namespace

Note :
- If we define member without namespace then it is considered as member of global namespace.
- If we want to access members of namespace frequently then we should use using directive.

cin and cout

- C++ provides an easier way for input and output.
- Console Output : Monitor
  - iostream is the standard header file of C++ for using cin and cout.
  - cout is external object of ostream class.
  - cout is member of std namespace and std namespace is declared in iostream header file.
  - cout uses insertion operator(<<)
- Console Input : Keyborad
  - cin is an external object of istream class.
  - cin is a member of std namespace and std namespace is declared in header file.
  - cin uses Extraction operator(>>)
- The output:
  - cout << “Hello C++”;
- The input:
  - cin >> var;

Escape Sequence and Manipulators

- **Manipulators** are helping functions that can modify the input/output stream.
- It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.
- Header File : #include<iomanip> // input output manipulation

  - Setbase(16) or hex
  - setbase(8) or oct
  - setbase(10) or dec
  - endl
- **Escape Sequences**
  - \b , \t , \n , \\, \' , \"
- **setw (val)**
- **setfill(char c)**
- **setprecision (val)**
- **Example :**

  double f =3.14159;

Class

- Building block that binds together data & code.
- Program is divided into different classes
- Class is collection of data member and member function.
- Class represents set/group of such objects which is having common structure and common behavior.
- Class is logical entity.
- Class has
  - Variables (data members)
  - Functions (member functions or methods)
- By default class members are private(not accessible outside class scope)
- Classes are stand-alone components & can be distributed in form of libraries
- Class is blue-print of an object

Data Members and Member Functions

Data Members

- Data members of the class are generally made as private to provide the data security.
- The private members cannot be accessed outside the class.
- So these members are always accessed by the member functions.

Member Functions

- Member functions are generally declared as public members of class.
- Constructor : Initialize Object
- Destructor : De-initialize Object
- Mutators : Modifies state of the object
- Inspectors : Don’t Modify state of object
- Facilitator : Provide facility like IO

Object

- Object is an instance of class.
- Entity that has physical existence, can store data, send and receive message to communicate with other objects.
- An entity, which get space inside memory is called object.
- Object is used to access data members and member function of the class
- Process of creating object from a class is called instantiation
- **Object has**
  - Data members (***state*** of object)
    - Value stored inside object is called state of the object.
    - Value of data member represent state of the object.
  - Member function (***behavior*** of object)
    - Set of operation that we perform on object is called behaviour of an object.
    - Member function of class represent behaviour of the object.
    - is how object acts & reacts, when its state is changed & operations are done
    - Operations performed are also known as messages
  - Unique address(***identity*** of object)

Few Points to note

- Member function do not get space inside object.
- If we create object of the class then only data members get space inside object. Hence size of object is depends on size of all the data members declared inside class.
- Data members get space once per object according to the order of data member declaration.
- Structure of the object is depends on data members declared inside class.
- Member function do not get space per object rather it gets space on code segment and all the objects of same class share single copy of it.
- Member function's of the class defines behaviour of the object.

Access Specifier

- - If we want to control visibility of members of structure/class then we should use access Specifier.
- Defines the accessibility of data member and member functions

- **Access specifiers in C++**
  1. private(-)
  2. protected(#)
  3. public(+)

- 1. Private - Can access inside the same struct/class in which it is declared Generally data members should declared as private. (data security)
- 2. public - Can access inside the same struct/class in which it is declared as well as inside out side function(like main()). Generally member functions should declared as public.

Scope Resolution Operator (: :)

- :: operator is used to bind a member with some class or namespace.
- It can be used to define members outside class.
- Also used to resolve ambiguity.
- It can also be used to access global members.
  - Example :- ::a =10; access global var.
- Scope resolution Operator is used to :
  - to call global functions
  - to define member functions of class outside the class
  - to access members of namespaces

Example Scope Resolution

```
class complex {
   int real, imag;
public: complex();
   void show();
};
complex::complex() {
   real = imag = 0;
}
void complex::show() {
   cout<<real<<imag;
}
complex obj;
obj.show();
```

Program Demo

- Declare one **class complex within namespace** , having real,imag as data members,declare functions accept() and display(). Define these outside the class using (::). Call accept() and display() from main.

Functions / User Defined Functions

- It is a set of instructions written to gather as a block to complete specific functionality.
- Function can be reused.
- It is a subprogram written to reduce complexity of source code
- Function may or may not return value.
- Function may or may not take argument
- Function can return only one value at time
- Function is building block of good top-down, structured code function as a "black box“
- **Writing function helps to**
  - improve readability of source code
  - helps to reuse code
  - reduces complexity
- **Types of Functions**
  - Library Functions
  - User Defined Functions

User Defined Functions

- **Function declaration / Prototype / Function Signature**

  <return type> <functionName> ([<arg type>...]);

- **Function Definition**

  <return type> < functionName > ([<arg type> <identifier>...])

  {

  }

- **Function Call**

  <location> = < functionName >(<arg value/address>);

Inline Function

- C++ provides a keyword *inline* that makes the function as inline function.
- Inline functions get replaced by compiler at its call statement. It ensures faster execution of function just like macros.
- Advantage of inline functions over macros: inline functions are type-safe.
- Inline is a request made to compiler.
- If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

When to use Inline function?

- We can use Inline function as per our needs.
- We can use the inline function when performance is needed.
- We can use the inline function over macros.
- We prefer to use the inline keyword outside the class with the function definition to hide implementation details of the function.

Default Arguments

- In C++, functions may have arguments with the default values. Passing these arguments while calling a function is optional.
- A default argument is a default value provided for a function parameter/argument.
- If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used.
- If such argument is not passed, then its default value is considered. Otherwise arguments are treated as normal arguments.
- Default arguments should be given in right to left order.
  - int sum (int a, int b, int c=0, int d=0) {

```
        return a + b + c + d;
}
```

  - The above function may be called as
    - Res=sum(10,20);
    - Res=sum(10,20,40);
    - Res=sum(10,30,40,50);

Function Overloading

- Functions with same name and different signature are called as overloaded functions.
- Return type is not considered for function overloading.
- Function call is resolved according to types of arguments passed.
- Function overloading is possible due to name mangling done by the C++ compiler (Name mangling process , mangled name)
- Differ in number of input arguments
- Differ in data type of input arguments
- Differ at least in the sequence of the input arguments

  - Example :
    - int sum(int a, int b) { return a+b; }
    - float sum(float a, float b) { return a+b; }
    - int sum(int a, int b, int c) { return a+b+c;;

Access Specifier

- - If we want to control visibility of members of structure/class then we should use access Specifier.
- Defines the accessibility of data member and member functions

- **Access specifiers in C++**
  1. private(-)
  2. protected(#)
  3. public(+)

- 1. Private - Can access inside the same struct/class in which it is declared Generally data members should declared as private. (data security)
- 2. public - Can access inside the same struct/class in which it is declared as well as inside out side function(like main()). Generally member functions should declared as public.

Scope Resolution Operator (: :)

- :: operator is used to bind a member with some class or namespace.
- It can be used to define members outside class.
- Also used to resolve ambiguity.
- It can also be used to access global members.
  - Example :- ::a =10; access global var.
- Scope resolution Operator is used to :
  - to call global functions
  - to define member functions of class outside the class
  - to access members of namespaces

Types of Member Functions within class

- Constructor : object inialization
- Desctructor : used to release the resources
- Mutators/setter : modify state of object
- inspector/getter : do not change the state of the object
- facilitator

Constructor

- It is a member function of a class which is used to initialize object.
- Constructor has same name as that of class and don’t have any return type.
- Constructor get automatically called when object is created i.e. memory is allocated to object.
- If we don’t write any constructor, compiler provides a default constructor.
- Due to following reasons, constructor is considered as special function of the class:
  1. Its name is same as class name.
  2. It doesn't have any return type.
  3. It is designed to call implicitly.
  4. In the life time of the object , it gets called only once per object and according to order of its declaration.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.
- We can not declare constructor static, constant, volatile or virtual. We can declare constructor only inline.
- Constructor overloading means inside a class more than one constructor is defined.
- We can have constructors with
  - No argument : initialize data member to default values
  - One or more arguments : initialize data member to values passed to it
  - Argument of type of object : initialize object by using the values of the data members of the passed object. It

Types of Constructor

- Parameterless constructor
  - also called zero argument constructor or user defined default constructor
  - If we create object without passing argument then parameterless constructor gets called
  - Constructor do not take any parameter
- Parameterized constructor
  - If constructor take parameter then it is called parameterized constructor
  - If we create object, by passing argument then paramterized constructor gets called
- Default constructor
  - If we do not define constructor inside class then compiler generates default constructor for the class.
  - Compiler generated default constructor is parameterless.

Types of Member Functions within class

- Constructor : object inialization
- Desctructor : used to release the resources
- Mutators/setter : modify state of object
- inspector/getter : do not change the state of the object
- facilitator

Constructor

- It is a member function of a class which is used to initialize object.
- Constructor has same name as that of class and don’t have any return type.
- Constructor get automatically called when object is created i.e. memory is allocated to object.
- If we don’t write any constructor, compiler provides a default constructor.
- Due to following reasons, constructor is considered as special function of the class:
  1. Its name is same as class name.
  2. It doesn't have any return type.
  3. It is designed to call implicitly.
  4. In the life time of the object , it gets called only once per object and according to order of its declaration.
- We can not call constructor on object, pointer or reference explicitly. It is designed to call implicitly.
- We can not declare constructor static, constant, volatile or virtual. We can declare constructor only inline.
- Constructor overloading means inside a class more than one constructor is defined.
- We can have constructors with
  - No argument : initialize data member to default values
  - One or more arguments : initialize data member to values passed to it
  - Argument of type of object : initialize object by using the values of the data members of the passed object. It

Types of Constructor

- Parameterless constructor
  - also called zero argument constructor or user defined default constructor
  - If we create object without passing argument then parameterless constructor gets called
  - Constructor do not take any parameter
- Parameterized constructor
  - If constructor take parameter then it is called parameterized constructor
  - If we create object, by passing argument then paramterized constructor gets called
- Default constructor
  - If we do not define constructor inside class then compiler generates default constructor for the class.
  - Compiler generated default constructor is parameterless.

Constructor's member initializer list

- If we want to initialize data members according to users requirement then we should use constructor body.

```
class Test
{
private:
        int num1;
        int num2;
        int num3;
public:
        Test( void )
        {
        this->num1 = 10;
        this->num2 = 20;
        this->num3 = num2;
        }
};
```

- If we want to initialize data member according to order of data member declaration then we can use constructors member initializer list.

```
class Test
{
private:
        int num1;
        int num2;
        int num3;
public:
        Test( void ) : num1( 10 ), num2( 20 ), num3( num2)
        { }
};
```

Except array we can initialize any member inside constructors member initializer list.

Destructor

- It is a member function of a class which is used to release the resources.
- It is considered as special function of the class
  - Its name is same as class name and always preceds with tild operator(~)
  - It doesnt have return type or doesn't take parameter.
  - It is designed to call implicitly.
- Destructor calling sequence is exactly opposite of constructor calling sequence.
- Destructor is designed to call implicitly.
- If we do not define destructor inside class then compiler generates default destructor for the class.
- Default destructor do not de allocate resources allocated by the programmer. If we want to de allocate it then we should define destructor inside class.

Modular Approach

- "/usr/include" directory is called standard directory for header files.
- It contains all the standard header files of C/C++
- If we include header file in angular bracket (e.g #include<filename.h>) then preprocessor try to locate and load header file from standard directory only(/usr/include).
- If we include header file in double quotes (e.g #include“filename.h") then preprocessor try to locate and load header file first from current project directory if not found then it try to locate and load from standard directory.

Header Guard

```
#ifndef HEADER_FILE_NAME_H_
#define HEADER_FILE_NAME_H_
//TODO : Type declaration here
#endif
```

Constant in C++

- We can declare a constant variable that cannot be modified in the app.
- If we do not want to modify value of the variable then const keyword is used.
- constant variable is also called as read only variable.
- The value of such variable should be known at compile time.
- In C++ , Initializing constant variable is mandatory
- const int i=3; //VALID
- Const int val; //Not ok in c++
- Generally const keyword is used with the argument of function to ensure that the variable cannot be modified within that function.

Constant data member

- Once initialized, if we do not want to modify state of the data member inside any member function of the class including constructor body then we should declare data member constant.
- If we declare data member constant then it is mandatory to initialize it using constructors member initializer list.

```
class Test
{
private:
        const int num1;
public:
        Test( void ) : num1( 10 ) //OK
        {
        //this->num1 = 10; //Not OK
        }
};
```

Const member function

- The member function can declared as const. In that case object invoking the function cannot be modified within that member function.
- We can not declare global function constant but we can declare member function constant.
- If we do not want to modify state of current object inside member function then we should declare member function as constant.
- void display() const;
- Even though normal members cannot be modified in const function, but *mutable* data members are allowed to modify.
- In constant member function, if we want to modify state of non constant data member then we should use **mutable keyword.**
- We can not declare following function constant:
  1. Global Function
  2. Static Member Function
  3. Constructor
  4. Destructor

Reference

- Reference is derived data type.
- It alias or another name given to the exisiting memory location / object.
  - Example : int a=10; int &r = a;
  - In above example a is referent variable and r is reference variable.
  - It is mandatory to initialize reference.
- Reference is alias to a variable and cannot be reinitialized to other variable
- When ‘&’ operator is used with reference, it gives address of variable to which it refers.
- Reference can be used as data member of any class
- **Using typedef we can create alias for class whereas using reference we can create alias for object.**

Reference

- We can not create reference to constant value.
  - int &num2 = 10; //can not create reference to constant value
- Reference is internally considered as constant pointer hence referent of reference must be variable/object.

```
int main( void )
{
        int num1 = 10;
        int &num2 = num1;
        //int *const num2 = &num1;
        cout<<"Num2 : "<<num2<<endl;
        //cout<<"Num2 : "<<*num2<<endl;
        return 0;
}
```

pass arguments to function, by value, by address or by reference.

- **In C++, we can pass argument to the function using 3 ways:**

1. By Value
2. By Address
3. By Reference

- If variable is passed by reference, then any change made in variable within function is reflected in caller function.
- Reference can be argument or return type of any function

Copy Constructor

- Copy constructor is a single parameter constructor hence it is considered as parameterized constructor
- Example:
  - **.Complex sum(const Complex &c2)**

Dynamic Memory Allocation

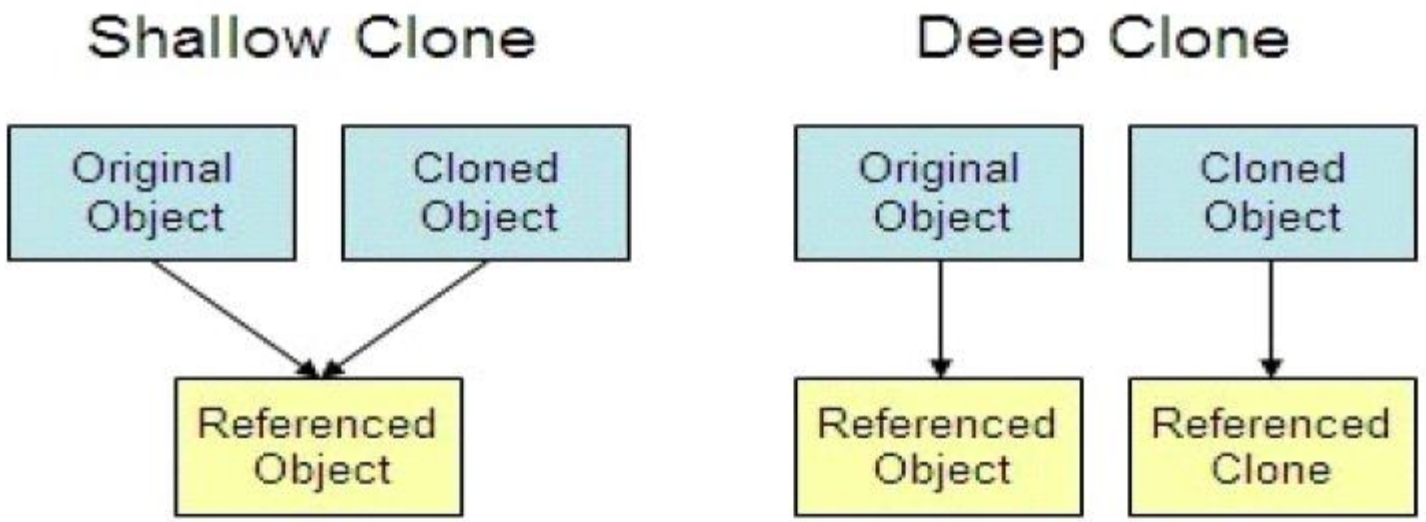
- If we want to allocate memory dynamically then we should use new operator and to deallocate that memory we should use delete operator.
- If pointer contains, address of deallocated memory then such pointer is called dangling pointer.
- When we allocate space in memory, and if we loose pointer to reach to that memory then such wastage of memory is called memory leakage.

- Example :

```
int main()
{
    int *ptr = new int;         //int *ptr = ( int* )::operator new( sizeof( int ) * 1 );
    *ptr = 125;       //Dereferencing
    cout<<"Value  :           "<<*ptr<<endl; //Dereferencing
    delete ptr;                 //::operator delete( ptr );
    ptr = NULL;
    return 0;
}
```

Object Copying

- In object-oriented programming, “object copying” is a process of creating a copy of an existing object.
- The resulting object is called an object copy or simply copy of the original object.
- Methods of copying:
  - Shallow copy
  - Deep copy



Types of Copy

- **Shallow Copy**
  - The process of copying state of object into another object.
  - It is also called as bit-wise copy/ bit-by bit copy.
  - When we assign one object to another object at that time copying all the contents from source object to destination object as it is. Such type of copy is called as shallow copy.
  - Compiler by default create a shallow copy. Default copy constructor always create shallow copy.
- **Deep Copy**
  - Deep copy is the process of copying state of the object by modifying some state.
  - It is also called as member-wise copy.
    - When class contains at least one data member of pointer type,
    - when class contains user defined destructor
    - And when we assign one object to another object at that time instead of copy base address allocate a new memory for each and every object and then copy contain from memory of source object into memory of destination object. Such type of copy is called as deep copy.

Shallow Copy

- Process of copying state of object into another object as it is, is called shallow copy.
- It is also called as bit-wise copy / bit by bit copy.
- Following are the cases when shallow copy taken place:
    1. If we pass variable / object as a argument to the function by value.
    2. If we return object from function by value.
    3. If we initialize object: Complex c2=c1
    4. If we assign the object , c2=c1;
    5. If we catch object by value.
- Examples of shallow copy

  Example 1: (Initialization)

```
int num1=50;
int num2=num1;
```

  Example 2: (Assignment)

```
Complex c1(40,50);
c2=c1;
```

Deep Copy

- It is also called as member-wise copy. By modifying some state, if we create copy of the object then it is called deep copy.
  - Conditions to create deep copy
    - Class must contain at least one pointer type data member.

```
class Array
        {               private:
                        int size;
                        int *arr;
                        public:
                        Array( int size )
                        {                                       this->size = size;
                                        this->arr = new int[ this->size ];
                        }
        };
```

- Steps to create deep copy
  - 1. Copy the required size from source object into destination object.

Static Variable

- All the static and global variables get space only once during program loading / before starting execution of main function
- Static variable is also called as shared variable.
- Unintialized static and global variable get space on BSS segment.
- Intialized static and global variable get space on Data segment.
- Default value of static and global variable is zero.
- Static variables are same as global variables but it is having limited scope.

Static Methods or Static Member Functions

- Except main function, we can declare global function as well as member function static.
- To access non static members of the class, we should declare member function non static and to access
- static members of the class we should declare member function static.
- Member function of a class which is designed to call on object is called instance method. In short non static member function is also called as instance method.
- To access instance method either we should use object, pointer or reference to object.
- static member function is also called as class level method.
- To access class level method we should use classname and ::(scope resolution) operator.

Exception Handling

- If we give wrong input to the application then it generates runtime error/exception.
- Exception is an object, which is used to send notification to the end user of the system if any exceptional situation occurs in the program.
- To handle exception then we should use 3 keywords:
- 1**. try**
  - try is keyword in C++.
  - If we want to inspect exception then we should put statements inside try block/handler.
  - Try block may have multiple catch block but it must have at least one catch block.

Note : For thrown exception, if we do not provide matching catch block then C++ runtime gives call to the std::terminate() function which implicitly gives call to the std::abort() function.

- **2. catch**
  - If we want to handle exception then we should use catch block/handler.
  - Single try block may have multiple catch block.
  - Catch block can handle exception thrown from try block only.
  - A catch block, which can handle any type of exception is called generic catch block / catch-all handler.
  - For each type of exception, we can write specific catch block or we can write single catch block which can handle all types of exception. A catch block which can handle all type of exception is called generic catch block.
- **3. throw**
  - throw is keyword in C++.
  - If we want to generate exception explicitly then we should use throw keyword.
  - "throw statement" is a jump statement.
  - To generate new exception, we should use throw keyword. Throw statement is jump statement.

Operator Overloading

- operator is token in C/C++.
- It is used to generate expression.
- operator is keyword in C++.
- Types of operator:
  - Unary operator (++,--,&,!,~,sizeof())
  - Binary Operator (Arithmetic, relational, logical , bitwise, assignment)
  - Ternary operator (conditional)
- In C++, also we can not use operator with objects of user defined type directly.
- If we want to use operator with objects of user defined type then we should overload operator.
- To overload operator, we should define **operator function.**
- **We can define operator function using 2 ways:**
  - **Using member function**
  - **Using non member function**

Need Of Operator Overloading

- we extend the meaning of the operator.
- If we want to use operator with the object of use defined type, then we need to overload operator.
- To overload operator, we need to define operator function.
- In C++, operator is a keyword
  - Suppose we want to use plus(+) operator with objects then we need to define operator+( ) function.

We define operator function either inside class (as a member function) or outside class (as a non-member function).	Point pt1(10,20), pt2(30,40), pt3;
	
	pt3 = pt1 + pt2; //pt3 = pt1.operator+(pt2); //using member function
	//or
	pt3 = pt1 + pt2; //pt3 = operator+(pt1, pt2); //using non member function

Operator Overloading

using member function

- **operator function must be member function**
- If we want to overload, binary operator using member function then **operator function should take only one parameter.**
  - Example : c3 = c1 + c2; //will be called as ------ c3 = c1.operator+(c2)

Example :

```
Point operator+( Point &other ) //Member Function
    {
    Point temp;
    temp.xPos = this->xPos + other.xPos;
    temp.yPos = this->yPos + other.yPos;
    return temp;
    }
```

using non member function

- **Operator function must be global function**
- If we want to overload binary operator using non member function then **operator function should take two parameters.**
  - **Example :** c3 = c1 + c2; //will be called as ------c3 = operator+(c1,c2);

Example:

```
Point operator+( Point &pt1, Point &pt2 ) //Non Member
   Function
{
Point temp;
temp.xPos = pt1.xPos + pt2.xPos;
temp.yPos = pt1.yPos + pt2.yPos;
return temp;
```

We can not overloading following operator using member as well as non member function:

1. dot/member selection operator(.)
2. Pointer to member selection operator(.*)
3. Scope resolution operator(::)
4. Ternary/conditional operator(? :)
5. sizeof() operator
6. typeid() operator
7. static_cast operator
8. dynamic_cast operator
9. const_cast operator
10. reinterpret_cast operator

We can not overload following operators using non member function:

- Assignment operator(=)
- Subscript / Index operator([])
- Function Call operator[()]
- Arrow / Dereferencing operator(->)

Template

- If we want to write generic program in C++, then we should use template.
- This feature is mainly designed for implementing generic data structure and algorithm.
- If we want to write generic program, then we should pass data type as a argument. And to catch that type we should define template.
- Using template we can not reduce code size or execution time but we can reduce developers effort.

int num1 = 10, num2 = 20; swap_object<**int>(num1, num2);** string str1="Pune", str2="Karad"; swap_object<string>(str1, str2);	In this code, <int> and <string> is considered as type argument.

template<typename T> //or
template<class T> //T : Type Parameter
void swap(b obj1, **T** obj2 **)**
{
T temp = obj1;
obj1 = obj2;
obj2 = temp;
}
	template and typename is keyword in C++. By passing datatype as argument we can write generic code hence parameterized type is called template

Types of Template

- Function Template
- Class Template

Example of Function Template

```
//template<typename T>//T : Type Parameter
template<class T> //T : Type Parameter
void swap_number( T &o1, T &o2 )
{   T temp = o1;
    o1 = o2;
    o2 = temp;
}
 int main( void )
{
    int num1 = 10;
    int num2 = 20;
    swap_number<int>( num1, num2 );     //Here int is type argument
    cout<<"Num1 : "<<num1<<endl;
    cout<<"Num2 : "<<num2<<endl;
    return 0;
}
```

Example of Class Template

```
template<class T>
class Array // Parameterized type
{
private:
    int size;
    T *arr;
    public:
    Array( void ) : size( 0 ), arr( NULL )
    {
    }
    Array( int size )
    {
    this->size = size;
    this->arr = new T[ this->size ];
    }
```

```
int main(void)
{
Array<char> a1( 3 );
a1.acceptRecord();
a1.printRecord();
return 0;
}
```

Scope

- It decides area/region/boundry in which we can access the element.
- **Types of scope in C++:**
  1. **Block scope**
  2. **Function scope**
  3. **Prototype scope**
  4. **Class scope**
  5. **Namespace scope**
  6. **File scope**
  7. **Program scope**

Example Scope

```
int num6;           //Program Scope
static int num5;  //File Scope
namespace ntest
{  int num4;   //Namespace scope
   class Test
   {   int num3;   //Class Scope  };
}
void sum( int num1, int num2 ); //Prototype scope
int main( void )
{
   int num1 = 10; //Function Scope
   while( true )
   {  int temp = 0; }
   return 0;   //Block Scope
}
```

C++ friend Function and friend Classes

- It is a mechanism built in C++ programming to access private or protected data from non-member functions.
- This is done using a friend function or/and a friend class.
- If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.
- The complier knows a given function is a friend function by the use of the keyword **friend**.
- For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.
- **Declaration of friend function in C++**
  - Syntax : class class_name { ... . friend return_type function_name(argument/s); ... . }
  - Now, you can define the friend function as a normal function to access the data of the class.
    No friend keyword is used in the definition.

    class className { ... . friend return_type functionName(argument/s); ... . }

    return_type functionName(argument/s) { ... . // Private and protected data of className can be accessed from // this function because it is a friend function of className. ... . }

Example Friend function

- If we want to access private members inside derived class
  - Either we should use member function(getter/setter).
  - Or we should declare a facilitator function as a friend function.
  - Or we should declare derived class as a friend inside base class.

We can declare global function (including main function) as well as member function as a friend inside class. We can write friend declaration statement inside any section(private/ protected/public) of the class. Consider this code snippet:	**class Test**
	{
	private: int number;
	public:
	Test(void)
	{ **this->number = 10** }
	friend void print(void);
	};
	void print(void)
	{ Test t; cout<<"Number : "<<t.number<<endl; }
	int main(void)
	{ print();
	return 0;
	}

friend Class in C++ Programming

- like a friend function, a class can also be made a friend of another class using keyword friend.

```
class B;
class A
{
        // class B is a friend class of class A
        friend class B;
         ... .. ...
}
class B
 {
... .. ...
}
```

In this example , all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A.
But, member functions of class A cannot access the data of class B.

- When a class is made a friend class, all the member functions of that class becomes friend functions.

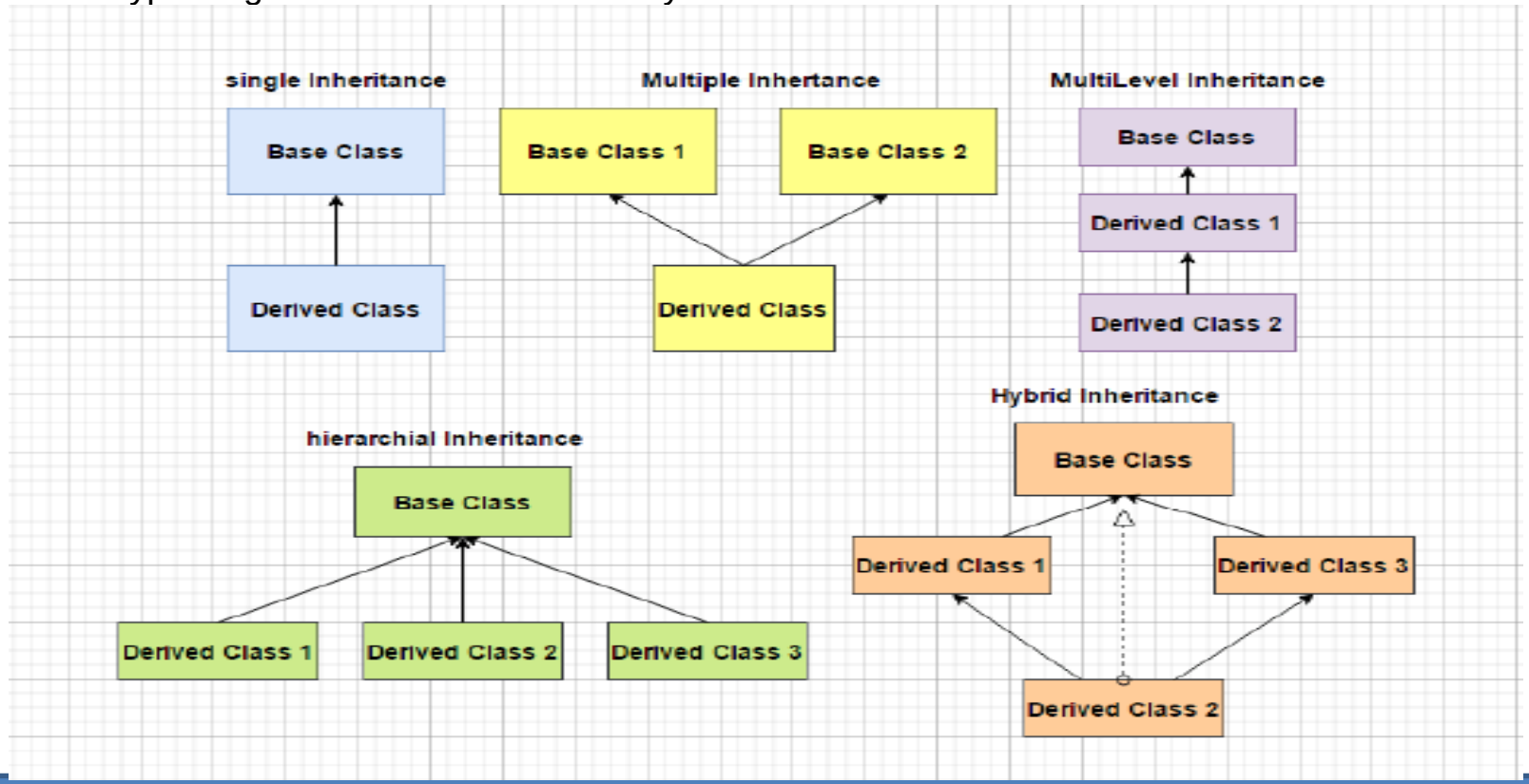
C++ Inheritance

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.
- It provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.
- Syntax:
  - class derived-class: access-specifier base-class
- If "is-a" relationship exist between two types then we should use inheritance.
- Inheritance is also called as "Generalization".
- Example: Book is-a product
- During inheritance, members of base class inherit into derived class.

Types of Inheritance

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance

If we combine any two or more types together then it is called as hybrid inheritance.



Inheritance

- If we create object of derived class then non static data members declared in base class get space inside it.
- If we use private/protected/public keyword to control visibility of members of class then it is called access Specifier.
- If we use private/protected/public keyword to extend the class then it is called **mode of inheritance**.
- Default mode of inheritance is private.
  - Example: class Employee : person //is treated as class Employee : private Person
- Example: class Employee:public Person
- In all types of mode, private members inherit into derived class but we can not access it inside member function of derived class.
- If we want to access private members inside derived class then:
  - Either we should use member function(getter/setter).
  - or we should declare derived class as a friend inside base class.

Association

- If has-a relationship exist between two types then we should use association.
- Example : Car has-a engine (OR engine is part-of car)
- If object is part-of / component of another object then it is called association.
- If we declare object of a class as a data member inside another class then it represents association.
- Example Association:

```
class Engine
{ };
class Car
{           private:
   Engine e;   //Association
};
int main( void )
{  Car car;
   return 0;
}
```

Composition and aggregation are specialized form of association

Composition

- If dependency object do not exist without Dependant object then it represents composition.
- Composition represents tight coupling.
- Example: Human has-a heart.

```
class Heart
{ };
class Human
{  Heart hrt; //Association->Composition
};
int main( void )
{  Human h;
   return 0;
```

Aggregation

- If dependency object exist without Dependant object then it represents Aggregation.
- Aggregation represents loose coupling.

```
class Faculty
{ };
class Department
{
   Faculty f; //Association->Aggregation
};
int main( void )
{
   Department d;
```

Association

- If has-a relationship exist between two types then we should use association.
- Example : Car has-a engine (OR engine is part-of car)
- If object is part-of / component of another object then it is called association.
- If we declare object of a class as a data member inside another class then it represents association.
- Example Association:

```
class Engine
{ };
class Car
{           private:
   Engine e;   //Association
};
int main( void )
{  Car car;
   return 0;
}
```

Composition and aggregation are specialized form of association

Composition

- If dependency object do not exist without Dependant object then it represents composition.
- Composition represents tight coupling.
- Example: Human has-a heart.

```
class Heart
{ };
class Human
{  Heart hrt; //Association->Composition
};
int main( void )
{  Human h;
   return 0;
```

Aggregation

- If dependency object exist without Dependant object then it represents Aggregation.
- Aggregation represents loose coupling.

```
class Faculty
{ };
class Department
{
   Faculty f; //Association->Aggregation
};
int main( void )
{
   Department d;
```

Access Control and Inheritance / Mode of inheritance

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

- If we use private, protected and public keyword to manage visibility of the members of class then it is called as access specifier.
- But if we use these keywords to extends the class then it is called as mode of inheritance.
- C++ supports private, protected and public mode of inheritance. If we do not specify any mode, then default mode of inheritance is private.

Except following functions, including nested class, all the members of base class, inherit into the derived class

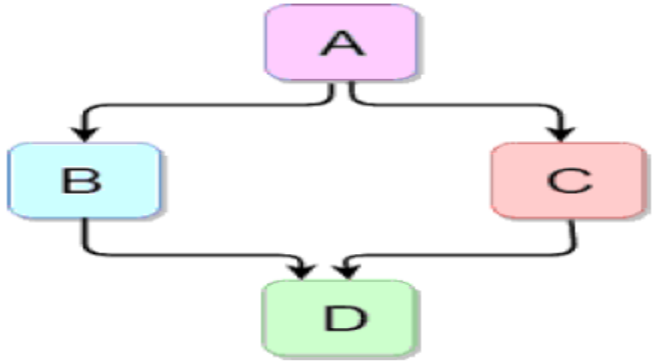
- Constructor
- Destructor
- Copy constructor
- Assignment operator
- Friend function.

Few Points to note in case of Inheritance

- Normally constructor calling sequence is depending on order of object declaration and destructor calling sequence is exactly opposite.
- But in case of inheritance, if we create object of derived class, first base class & then derived class constructor gets called. Destructor calling sequence is exactly opposite.
- From any constructor of derived class, by default base class's parameter less constructor gets called.
- Using Constructor's base initializer list, we can access any constructor of base class from constructor of derived class.
- So the conclusion is . without changing implementation of existing class if we want to extend meaning of the class then we should use inheritance.
- Hence inheritance can be defined as, "It is the process of accessing properties and behavior of base class inside derived class."

Diamond Problem

- As shown in diagram it is hybrid inheritance. Its shape is like diamond hence it is also called as diamond inheritance.
- Data members of indirect base class inherit into the indirect derived class multiple times. Hence it effects on size of object of indirect derived class.
- Member functions of indirect base class inherit into indirect derived class multiple times. If we try to call member function of indirect base class on object of indirect derived class, then compiler generates ambiguity error.
- If we create object of indirect derived class, then constructor and destructor of indirect base class gets called multiple times.
- All above problems generated by hybrid inheritance is called diamond problem.



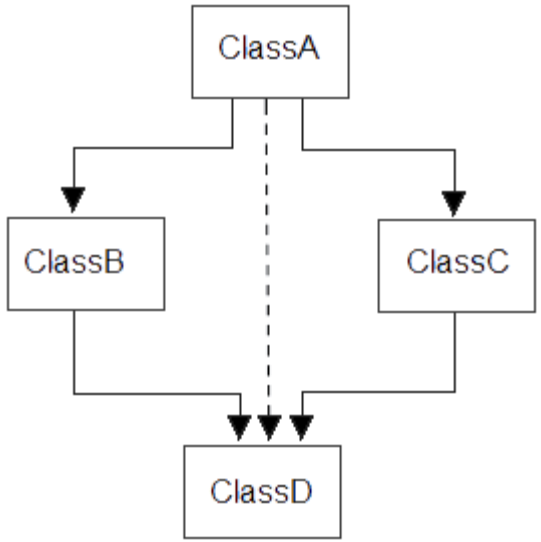
Solution to Diamond Problem– Virtual Base Class

- If we want to overcome diamond problem, then we should declare base class virtual i.e. we should derive class B & C from class A virtually. It is called virtual inheritance. In this case, members of class A will be inherited into B & C but it will not be inherited from B & C into class D.

```
class A { };
class B : virtual public A
{ };
class C : virtual public A
{ };
class D : public B, public C
{ };
```

A special case of hybrid inheritance : Multipath inheritance

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arrise in this type of inheritance.



Consider Example

```
ClassA
    {  public:
        int a;
    };
    class ClassB : public ClassA
    {  public:
        int b; } ;
    class ClassC : public ClassA
    {   public:
        int c;
    };
     class ClassD : public ClassB, public ClassC
    {   public:
        int d;
    };
```

```
void main()
    {    ClassD obj;
            //obj.a = 10;
  //Statement 1, Error
            //obj.a = 100;
  //Statement 2, Error
            obj.ClassB::a = 10;
  //Statement 3
            obj.ClassC::a = 100;
  //Statement 4
            obj.b = 20;
            obj.c = 30;
            obj.d = 40;
            cout<< "\n A from ClassB  :
```

Two possiblities to avoid above ambiguity

Use scope resolution operator

Use virtual base class

Virtual Function

- Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.
- **Early Binding**
- 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.
- **Late Binding**
- **Using Virtual Keyword in C++**
- 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.
- 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.
- **Points to note**
  - **Only the Base class Method's declaration needs the Virtual Keyword, not the definition.**
  - If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
  - The address of the virtual Function is placed in the **VTABLE** and the compiler uses **VPTR**(vpointer) to point to the Virtual Function

Program Demo

Early Binding

create a class Base and Derived (void show() in both classes)

create base *bptr;

bptr=&d;

bptr->show()

Late Binding

create a class Base and Derived (void show() in both classes one as virtual in base class)

create base *bptr;

bptr=&d;

bptr->show()

Virtual Functions

- During inheritance, members of base class inherit into the derived class hence derived class object can be considered as base class object.
- If b1 & b2 are objects of class Base and d1 and d2 are object of class Derived.
- b2 = b1; //allowed
- b1 = d1; //allowed : Object Slicing
- Base *ptr = new Base( ); //allowed
- Base *ptr = new Derived( ); //allowed: Upcasting
- d2 = d1; //allowed
- d1 = b1; //Not Allowed
- Derived *ptr = new Derived(); //Allowed
- Derived *ptr = new Base(); //Not Allowed

- Base class pointer can store address of derived class object. It is called upcasting.
- In case of up casting if we want to call function depending on type of object rather than type of pointer then we should declare function in base class virtual.
- in case of up casting we should declare function in base class virtual. In other words, virtual function is

Virtual Function calling

- virtual function is designed to call using base class pointer/reference.
- Virtual function implementation is implicitly based on virtual function pointer and virtual function table.
- If we declare member function virtual then compiler implicitly maintain one table to store address of declared virtual member function. It is called as virtual function table.
- To store address of virtual function table compiler implicitly declare one pointer as a data member of class. It is called as virtual function pointer.
- Due to virtual function pointer size of the object gets increased either by 2/4/8 bytes, depending on type of compiler.

Example

```
class Shape
{
protected:
        float area;
public:
        virtual void acceptRecord( )
        {       }
        virtual void calculateArea( )
        {       }
        void printRecord( void )const
        {
                cout<<"Area : "<<this->area<<endl;
        }
};
```

```
class Rectangle : public Shape
{
        float length, breadth;
public:
        void acceptRecord( void )
        {
                cin>>this->length>>this->breadth;
        }
        void calculateArea( )
        {
                this->area = this->length * this->breadth;
        }
};
```

```
class Circle : public Shape
{
        float radius;
public:
        void acceptRecord( void )
        {
                cin>>this->radius;
        }
        void calculateArea( )
        {
                this->area=3.14f*this->radius*this->radius;
        }
};
```

Function Overriding

- Process of redefining virtual member function of base class inside derived class with same signature is called function overriding.
- According to client's requirement, if implementation of base class member function is logically 100% complete then it should be non-virtual.
- According to client's requirement, if implementation of base class member function is partially complete then it should be virtual.
- According to client's requirement, if implementation of base class member function is logically 100% incomplete then it should be pure virtual.

Pure Virtual Function and Abstract Classs

- If we equate virtual function to zero then such virtual function is called pure virtual function.
- If class contains at least one pure virtual function, then it is called as abstract class.
- In this code class Shape contains pure virtual function hence it is considered as abstract class.
- If class contains all pure virtual function, then it is called as pure abstract class / interface.

```
class Shape    //Abstract class
{
protected:
        float area;
public:
        Shape( void ) : area( 0 )
        {       }
        virtual void acceptRecord( ) = 0;

        virtual void calculateArea( ) = 0

        void printRecord( void )const
        {
                cout<<"Area : "<<this->area<<endl;
        }
};
```

Abstract Class

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class.

```
classShape {
public:
virtualintArea() = 0; // Pure virtual function is declared as follows.
// Function to set width.
voidsetWidth(int w) {
width = w;
}
// Function to set height.
voidsetHeight(int h) {
height = h;
}
protected:
```

```
intmain() {
Rectangle R;
Triangle T;

R.setWidth(5);
R.setHeight(10);

T.setWidth(20);
T.setHeight(8);

cout <<"The area of the rectangle is: "<<
R.Area() <<endl;
cout <<"The area of the triangle is: "<< T.Area()
<<endl;
}
```

RTTI

- Runtime Type Information/ Identification (RTTI) and advanced casting operators are advanced features of C++.
- RTTI is the process of getting type information of object at runtime.
- To use RTTI, we should use typeid operator. Code to use RTTI is as follows:

```
#include<iostream>
#include<typeinfo>
#include<string>
using namespace std;
int main( void )
{
   int number;
   const type_info &type = typeid( number );
   string typeName = type.name();
   cout<<"Type : "<<typeName<<endl;
   return 0;
```

Advanced Typecasting Operators

- **static_cast:** In case of non-polymorphic type, for down casting we should use this operator.
- **dynamic_cast:** In case of polymorphic type, for down casting we should use this operator.
- **const_cast:** If we want to convert pointer to constant object into pointer to non-constant object then we should use this operator.
- **reinterpret_cast:** It is used to type conversion between incompatible types