C++ Programming

Trainer : Rohan Paramane

Email: rohan.paramane@sunbeaminfo.com

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.

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**

Program Demo without operator overloading

- Create a class point, having two fileld, x, y.
  - Point(void)
  - Point(int x, int y)

```
int main( void )
{
Point pt1(10,20);
Point pt2(30,40);
Point pt3;
pt3 = pt1 + pt2; //Not OK( implicitly)
return 0;
}
```

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 overload 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.

Template

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 ];
    }
    void acceptRecord( void ){}
    void printRecord( void ){ }
    ~Array( void ){ }
};
```

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

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;
}
```

Dependant Object : Car Object

Dependancy Object : Engine Object

Composition – First 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;
};
int main( void ){
  Human h;
  return 0;
}
```

- Dependant Object : Human Object
- Dependancy Object : Heart Object

Aggregation – Second Form of Association

Aggregation

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

Example: Department has-a facullty.

```
class Faculty
{ };
class Department
{
   Faculty f; //Association->Aggregation
};
int main( void )
{
   Department d;
   return 0;
}
```

- Dependant Object : Department Object
- Dependancy Object : Faculty Object

Inheritance

- 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.
- If we create object of derived class then non static data members declared in base class get space inside it.
- Size of object = sum of size of non static data members declared in base class and derived class.
- 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.

Syntax of inheritance in C++

class Person //Parent class
{ };
 	
class Employee : public Person // Child class
{ };
	In C++ Parent class is called as Base class and child class is called as derived class. To create derived class we should use colon(:) operator. As shown in this code, public is mode of inheritance.

class Person //Parent class
{
char name[30];
int age;
};
class Employee : public Person //Child class
{
int empid;
float salary;
};
	int main(void)
	{
	Person p;
	cout<<sizeof(p)<<endl;
	
	Employee emp;
	cout<<sizeof(emp)<<endl;
	
	return 0;
	}

If we create object of derived class, then all the non- static data member declared in base class & derived class get space inside it i.e. non-static static data members of base class inherit into the derived class.

Syntax of inheritance in C++

- Using derived class name, we can access static data member declared in base class i.e. static data member of base class inherit into derived class.

```
class Base{
protected:
static int number;
};
int Base::number = 10;
class Derived : public Base{
public:
static void print( void ){
cout<<Base::number<<endl;
}
};
```

```
int main( void )
{
Derived::print();
return 0;
}
```

```
class Derived : public Base
{
int num3;
static int num4;
public:
void setNum3( int num3 ){
  this->num3 = num3;
 }
static void setNum4( int num4 ) {
    Derived::num4 = num4;
} };
int Derived::num4;
```

```
int main( void )
{
Derived d;
d.setNum1(10);
d.setNum3(30);
Derived::setNum2(20);
Derived::setNum4(40);
return 0;
}
```

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.

Mode of inheritance

- 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.

Mode Of inheritance – Private & Protected

Private Mode of inheritance					
Access Specifier	**Same class**	**Friend Function**	**Non- member function**	**Derived class**	**Indirect Derived class**
Private	A	A	NA	NA	NA
Protected	A	A	NA	A	NA
Public	A	A	Using Base obj : A Using Derived obj : NA	A	NA

Protected Mode of inheritance					
Access Specifier	**Same class**	**Friend Function**	**Non- member function**	**Derived class**	**Indirect Derived class**
Private	A	A	NA	NA	NA
Protected	A	A	NA	A	A
Public	A	A	Using Base obj : A Using Derived obj : NA	A	A

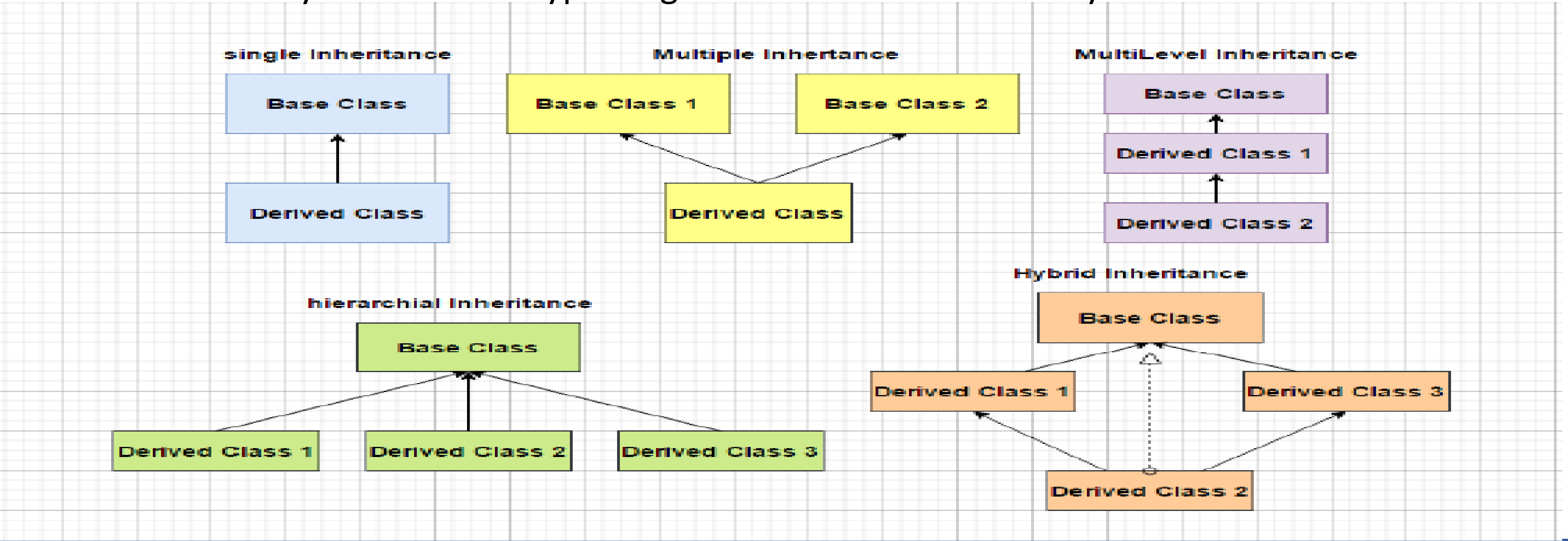
Mode Of inheritance - Public

Public Mode of inheritance					
Access Specifier	**Same class**	**Friend Function**	**Non-member Function**	**Derived class**	**Indirect Derived class**
Private	A	A	NA	NA	NA
Protected	A	A	NA	A	A
Public	A	A	Using Base obj : A Using Derived obj : A	A	A

Types of Inheritance

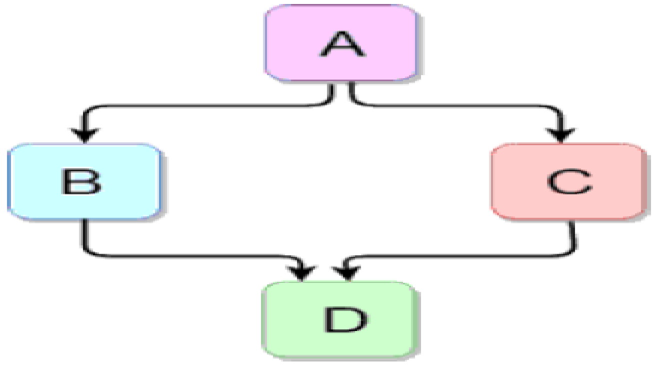
- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance

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



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
{ };
```

Virtual Keyword

- 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()

Abstract Class

- Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation.
- In such cases we can declare a function but cannot define it.
- Such functions are then made as pure virtual functions which must be implemented by the Derived class.
- Such a class where pure virtual function exists is called abstract class.
- We cannot create an object, but we can create pointer or reference of abstract class.

Thank You