C++ Programming

Trainer : Akshita Chanchlani

Email: akshita.chanchlani@sunbeaminfo.com

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

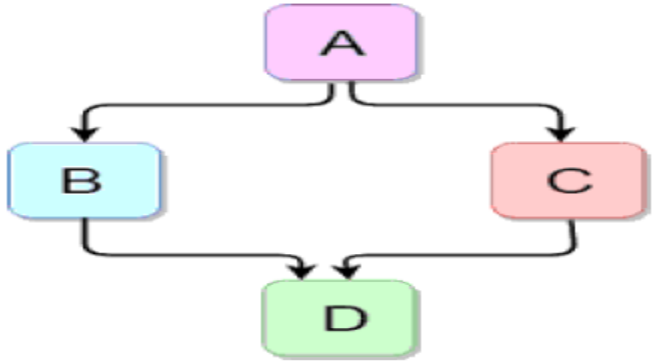
- Constructor
- Destructor
- Copy constructor
- Assignment operator
- Friend function.

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.



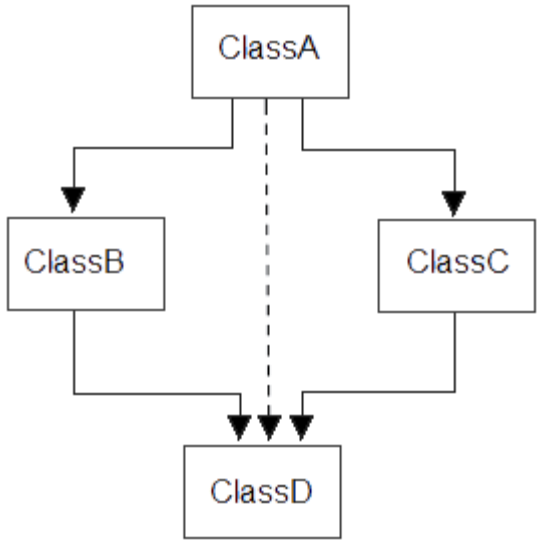
Solution to Diamond Problem– Virtual Base Class

- 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

Thank You