Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

DIGIRAL ASSIGNMENT - 3

NAME: NIHTISH.G

REG NO: 19BCS0012

SUBJECT: OBJECT ORIENTED PROGRAMMING

COURSE CODE: CSC2002

Faculty Name: Chandra Mouliswaran .S

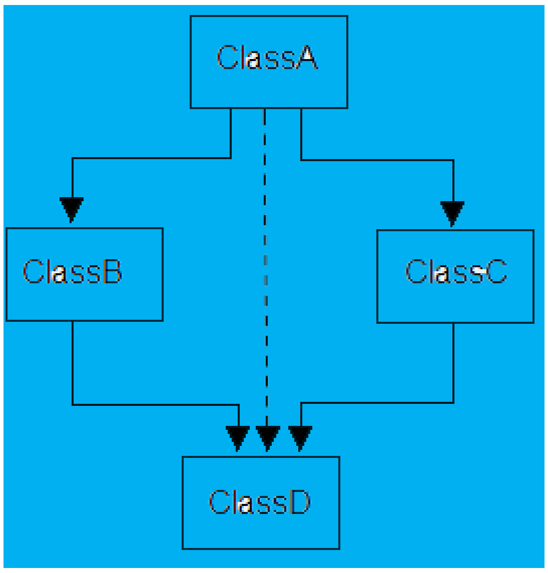
1. Analyse the ways in which ambiguity arises in multiple inheritance and find the ways in which it can be resolved. Explain with suitable example.

C++ Multipath Inheritance

A derived class with two base classes and these two base classes have one common base class is called **multipath inheritance**.

C++++ Multipath Inheritance Ambiguity

Ambiguity in multiple inheritance occur when a derived class have two base classes and these two base classes have one common base class. Consider the following figure:



Example of, occurrence of C++ ambiguity

```
#include<iostream.h>
#include<conio.h>

class 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()
{
                ClassDobj;

                 //obj.a = 10;              //Statement 1, Error occur
                 //obj.a = 100;             //Statement 2, Error occur

                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  : "<<obj.ClassB::a;
                cout<< "\n A from ClassC  : "<<obj.ClassC::a;

                cout<< "\n B : "<<obj.b;
                cout<< "\n C : "<<obj.c;
                cout<< "\n D : "<<obj.d;

    }
```

Output :

```
  A from ClassB  : 10
        A from ClassC  : 100
B : 20
C : 30
```

```
D : 40
```

In the above example, both **ClassB** & **ClassC** inherit **ClassA**, they both have single copy of **ClassA**. However **ClassD** inherit both **ClassB** & **ClassC**, therefore **ClassD** have two copies of **ClassA**, one from **ClassB** and another from **ClassC**.

If we need to access the data member **a** of **ClassA** through the object of **ClassD**, we must specify the path from which **a** will be accessed, whether it is from **ClassB** or **ClassC**, bco'z compiler can't differentiate between two copies of **ClassA** in **ClassD**.

There are two ways to avoid c++ ambiguity.

- Using scope resolution operator
- Using virtual base class

1. Avoid ambiguity using scope resolution operator

Using scope resolution operator we can manually specify the path from which data member **a** will be accessed, as shown in statement 3 and 4, in the above example.

```
obj.ClassB::a = 10;        //Statement 3
obj.ClassC::a = 100;       //Statement 4
```

*Note : still, there are two copies of **ClassA** in **ClassD.***

2. Avoid ambiguity using virtual base class

To remove multiple copies of **ClassA** from **ClassD**, we must inherit **ClassA** in **ClassB** and **ClassC** as **virtual** class.

Example to avoid ambiguity by making base class as a virtual base class

```
#include<iostream.h>
#include<conio.h>

class ClassA
{
    public:
    int a;
};

class ClassB :virtual public ClassA
{
    public:
    int b;
};
class ClassC :virtual public ClassA
{
    public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
    public:
    int d;
};
```

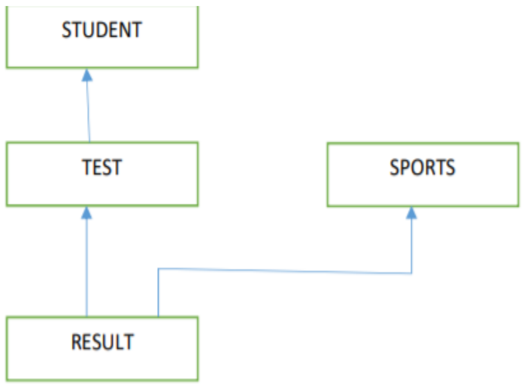
```
void main()
{

            ClassDobj;

            obj.a = 10;        //Statement 3
            obj.a = 100;       //Statement 4

            obj.b = 20;
            obj.c = 30;
            obj.d = 40;

            cout<< "\n A : "<<obj.a;
            cout<< "\n B : "<<obj.b;
            cout<< "\n C : "<<obj.c;
            cout<< "\n D : "<<obj.d;

    }
```

Output :

```
A : 100
B : 20
C : 30
D : 40
```

According to the above example, **ClassD** have only one copy of **ClassA** therefore statement 4 will overwrite the value of **a**, given at statement 3.

2. Implement the following hybrid inheritance.



Sample code:

```
#include <iostream>
          using namespace std;
          class student
          {
                    protected:
                              int roll_no;
                    public:
                              void get_no(int a)
                              {
roll_no=a;
                              }
                              void put_no(void)
                              {
cout<< "Roll No:"<<roll_no<<"\n";
                              }
          };
class test : public student
{
          protected:
                    float part1,part2;
          public:
                    void get_marks(float x,float y)
                    {
                              part1=x;  part2=y;
                    }
                    void put_marks(void)
{
cout<< "Marks obtained:" << "\n"
<< "Part1= " <<part1<<"\n"
<<"Part2= "<<part2<<"\n";
}
};

class sports
{
          protected:
                    float score;
          public:
          void get_score(float s)
          {
                    score=s;
          }
          void put_score(void)
          {
```

```
cout<< "Sports wt:" <<score<<"\n\n";
        }
};
class result : public test, public sports
{
        float total;
public:
        void display(void);
};
void result :: display(void)
{
        total=part1 + part2 +score;
put_no();
put_marks();
put_score();
cout<<"Total score: "<<total<< "\n";
}
int main()
{
        result stud;
stud.get_no(1223);
stud.get_marks(27.5, 33.0);
stud.get_score(6.0);
stud.display();
        return 0;

}
```

3. Explain the concept of abstract class and run time polymorphism with suitable example. Further explain how does the pure virtual function differs from virtual function.

Abstract Classes

An abstract class is, conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions.

A pure virtual function is one which **must be overridden** by any concrete (i.e., non-abstract) derived class. This is indicated in the declaration with the syntax **" = 0"** in the member function's declaration.

Example

```
classAbstractClass {
public:
virtual void AbstractMemberFunction() = 0; // Pure virtual function makes
// this class Abstract class.
virtual void NonAbstractMemberFunction1(); // Virtual function.

  void NonAbstractMemberFunction2();
};
```

In general an abstract class is used to define an implementation and is intended to be inherited from by concrete classes. It's a way of forcing a contract between the class designer and the users of that class. If we wish to create a concrete class (a class that can be instantiated) from an abstract class we must declare and **define** a matching member function for each abstract member function of the base class. Otherwise, if any member function of the base class is left undefined, we will create a new abstract class (this could be useful sometimes).

Sometimes we use the phrase "pure abstract class," meaning a class that exclusively has pure virtual functions (and no data). The concept of interface is mapped to pure abstract classes in C++, as there is no "interface" construct in C++ the same way that there is in Java.

Example:

```
classVehicle {
public:
explicit
Vehicle( inttopSpeed )
    : m_topSpeed( topSpeed )
    {}
    int TopSpeed() const {
returnm_topSpeed;
    }

virtual void Save( std::ostream& ) const = 0;

private:
    int m_topSpeed;
 };

classWheeledLandVehicle :public Vehicle {
public:
WheeledLandVehicle( inttopSpeed, int numberOfWheels )
    : Vehicle( topSpeed ), m_numberOfWheels( numberOfWheels )
    {}
    int NumberOfWheels() const {
returnm_numberOfWheels;
    }

    void Save( std::ostream& ) const; // is implicitly virtual

private:
    int m_numberOfWheels;
 };
classTrackedLandVehicle :public Vehicle {
public:
TrackedLandVehicle( inttopSpeed, int numberOfTracks )
   : Vehicle( topSpeed ), m_numberOfTracks ( numberOfTracks )
   {}
   int NumberOfTracks() const {
```

```
returnm_numberOfTracks;
   }
   void Save( std::ostream& ) const; // is implicitly virtual

private:
   int m_numberOfTracks;
  };
```

In this example the Vehicle is an abstract base class as it has an abstract member function.The class WheeledLandVehicle is derived from the base class. It also holds data which is common to all wheeled land vehicles, namely the number of wheels. The class TrackedLandVehicle is another variation of the Vehicle class.

This is something of a contrived example but it does show how that you can share implementation details among a hierarchy of classes. Each class further refines a concept. This is not always the best way to implement an interface but in some cases it works very well. As a guideline, for ease of maintenance and understanding you should try to limit the inheritance to no more than 3 levels. Often the best set of classes to use is a pure virtual abstract base class to define a common interface. Then use an abstract class to further refine an implementation for a set of concrete classes and lastly define the set of concrete classes.

An **abstract class** is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

```
classAB {
public:
virtual void f() = 0;
};
```

Function AB::f is a pure virtual function. A function declaration cannot have both a pure specifier and a definition.

Abstract class cannot be used as a parameter type, a function return type, or the type of an explicit conversion, and not to declare an object of an abstract class. It can be used to declare pointers and references to an abstract class.

Runtime Polymorphism In C++.

Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time.

In contrast, to compile time or static polymorphism, the compiler deduces the object at run time and then decides which function call to bind to the object. In C++, runtime polymorphism is implemented using method overriding.

In this tutorial, we will explore all about runtime polymorphism in detail.

Function Overriding

Function overriding is the mechanism using which a function defined in the base class is once again defined in the derived class. In this case, we say the function is overridden in the derived class.

We should remember that function overriding cannot be done within a class. The function is overridden in the derived class only. Hence inheritance should be present for function overriding.

The second thing is that the function from a base class that we are overriding should have the same signature or prototype i.e. it should have the same name, same return type and same argument list.

an example that demonstrates method overriding.

```
#include <iostream>

usingnamespacestd;

classBase

{

  public:
```

```
  voidshow_val()
  {
    cout<< "Class::Base"<<endl;
  }
};
classDerived:publicBase
{
  public:
  voidshow_val() //function overridden from base
  {
    cout<< "Class::Derived"<<endl;
  }
};
intmain()
{
  Base b;
  Derived d;
  b.show_val();
  d.show_val();
}
```

Output:

```
Class::Base
Class::Derived
```

In the above program, we have a base class and a derived class. In the base class, we have a function show_val which is overridden in the derived class. In the main function, we create an object each of Base and Derived class and call the show_val function with each object. It produces the desired output.

The above binding of functions using objects of each class is an example of static binding.

Now let us see what happens when we use the base class pointer and assign derived class objects as its contents.

The example program is shown below:

```
#include <iostream>

usingnamespacestd;

classBase

{

   public:

   voidshow_val()

   {

      cout<< "Class::Base";

   }

};

classDerived:publicBase

{

   public:

   voidshow_val()    //overridden function

   {

      cout<< "Class::Derived"; } }; intmain() { Base* b;

 //Base class pointer Derived d;

//Derived class object b = &d; b->show_val();   //Early Binding

}
```

Output:

```
Class::Base
```

Now we see, that the output is “Class:: Base”. So irrespective of what type object the base pointer is holding, the program outputs the contents of the function of the class whose base pointer is the type of. In this case, also static linking is carried out.

In order to make the base pointer output, correct contents and proper linking, we go for dynamic binding of functions. This is achieved using Virtual functions mechanism which is explained in the next section.

Virtual Function

For the overridden function should be bound dynamically to the function body, we make the base class function virtual using the “virtual” keyword. This virtual function is a function that is overridden in the derived class and the compiler carries out late or dynamic binding for this function.

Now let us modify the above program to include the virtual keyword as follows:

```
#include <iostream>

usingnamespacestd;.

classBase

{

   public:

   virtualvoidshow_val()

   {

      cout<< "Class::Base";

   }

};

classDerived:publicBase

{

   public:

   voidshow_val()

   {
```

cout<< "Class::Derived"; } }; intmain() { Base* b;

//Base class pointer Derived d;

//Derived class object b = &d; b->show_val();

//late Binding

}

Output:

Class::Derived

So in the above class definition of Base, we made show_val function as "virtual". As the base class function is made virtual, when we assign derived class object to base class pointer and call show_val function, the binding happens at runtime.

Thus, as the base class pointer contains derived class object, the show_val function body in the derived class is bound to function show_val and hence the output.

In C++, the overridden function in derived class can also be private. The compiler only checks the type of the object at compile time and binds the function at run time, hence it doesn't make any difference even if the function is public or private.

Note that if a function is declared virtual in the base class, then it will be virtual in all of the derived classes.

But till now, we haven't discussed how exactly virtual functions play a part in identifying correct function to be bound or in other words, how late binding actually happens.

The virtual function is bound to the function body accurately at runtime by using the concept of the **virtual table (VTABLE)** and a hidden pointer called **_vptr.**
Both these concepts are internal implementation and cannot be used directly by the program.

Working Of Virtual Table And _vptr

First, let us understand what a virtual table (VTABLE) is.

The compiler at compile time sets up one VTABLE each for a class having virtual functions as well as the classes that are derived from classes having virtual functions.

A VTABLE contains entries that are function pointers to the virtual functions that can be called by the objects of the class. There is one function pointer entry for each virtual function.

In the case of pure virtual functions, this entry is NULL. (This the reason why we cannot instantiate the abstract class).

Next entity, _vptr which is called the vtable pointer is a hidden pointer that the compiler adds to the base class. This _vptr points to the vtable of the class. All the classes derived from this base class inherit the _vptr.

Every object of a class containing the virtual functions internally stores this _vptr and is transparent to the user. Every call to virtual function using an object is then resolved using this _vptr.

Let us take an example to demonstrate the working of vtable and _vtr.

```
#include<iostream>

usingnamespacestd;

classBase_virtual

 {

 public:

    virtualvoidfunction1_virtual() {cout<<"Base :: function1_virtual()\n";};

    virtualvoidfunction2_virtual() {cout<<"Base :: function2_virtual()\n";};

    virtual~Base_virtual(){};

};


classDerived1_virtual: publicBase_virtual

{

public:

   ~Derived1_virtual(){};
   virtualvoidfunction1_virtual() { cout<<"Derived1_virtual :: function1_virtual()\n";}
; }; intmain() { Derived1_virtual *d = newDerived1_virtual; Base_virtual *b = d; b->
function1_virtual();
  b->function2_virtual();
  delete(b);


  return(0);
}
```

Output:

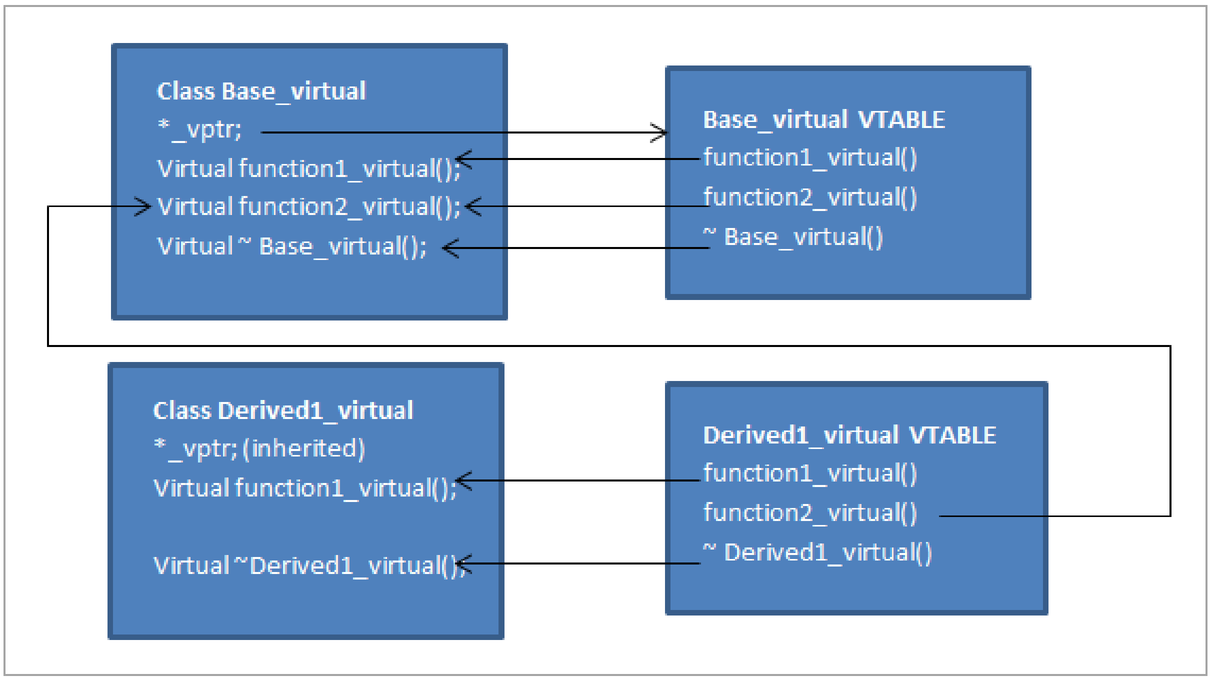
```
Derived1_virtual :: function1_virtual()
Base :: function2_virtual()
```

In the above program, we have a base class with two virtual functions and a virtual destructor. We have also derived a class from the base class and in that; we have overridden only one virtual function. In the main function, the derived class pointer is assigned to the base pointer.

Then we call both the virtual functions using a base class pointer. We see that the overridden function is called when it is called and not the base function. Whereas in the second case, as the function is not overridden, the base class function is called.

Now let us see how the above program is represented internally using vtable and _vptr.

As per the earlier explanation, as there are two classes with virtual functions, we will have two vtables – one for each class. Also, _vptr will be present for the base class.



Above shown is the pictorial representation of how the vtable layout will be for the above program. The vtable for the base class is straightforward. In the case of the derived class, only function1_virtual is overridden.

Hence we see that in the derived class vtable, function pointer for function1_virtual points to the overridden function in the derived class. On the other hand function pointer for function2_virtual points to a function in the base class.

Thus in the above program when the base pointer is assigned a derived class object, the base pointer points to _vptr of the derived class.

So when the call b->function1_virtual() is made, the function1_virtual from the derived class is called and when the function call b->function2_virtual() is made, as this function pointer points to the base class function, the base class function is called.

Pure Virtual Functions And Abstract Class

We have seen details about virtual functions in C++ in our previous section. In C++, we can also define a "**pure virtual function**" that is usually equated to zero.
The pure virtual function is declared as shown below.

virtual return_typefunction_name(arg list) = 0;

The class which has at least one pure virtual function that is called an “**abstract class**”. We can never instantiate the abstract class i.e. we cannot create an object of the abstract class.
This is because we know that an entry is made for every virtual function in the VTABLE (virtual table). But in case of a pure virtual function, this entry is without any address thus rendering it incomplete. So the compiler doesn’t allow creating an object for the class with incomplete VTABLE entry.

This is the reason for which we cannot instantiate an abstract class.

The below example will demonstrate Pure virtual function as well as Abstract class.

```
#include <iostream>
usingnamespacestd;
classBase_abstract
{
   public:
   virtualvoidprint() = 0;    // Pure Virtual Function
};
classDerived_class:publicBase_abstract
{
   public:
   voidprint()
   {
      cout<< "Overriding pure virtual function in derived class\n"; } }; intmain() {
 // Base obj; //Compile Time Error Base_abstract *b; Derived_class d; b = &d; b-
>print();
}
```

Output:

```
Overriding pure virtual function in the derived class
```

In the above program, we have a class defined as Base_abstract which contains a pure virtual function which makes it an abstract class. Then we derive a class "Derived_class" from Base_abstract and override the pure virtual function print in it.

In the main function, not that first line is commented. This is because if we uncomment it, the compiler will give an error as we cannot create an object for an abstract class.

But the second line onwards the code works. We can successfully create a base class pointer and then we assign derived class object to it. Next, we call a print function which outputs the contents of the print function overridden in the derived class.

Let us list some characteristics of abstract class in brief:

- We cannot instantiate an abstract class.
- An abstract class contains at least one pure virtual function.
- Although we cannot instantiate abstract class, we can always create pointers or references to this class.
- An abstract class can have some implementation like properties and methods along with pure virtual functions.
- When we derive a class from the abstract class, the derived class should override all the pure virtual functions in the abstract class. If it failed to do so, then the derived class will also be an abstract class.
-

Virtual Destructors

Destructors of the class can be declared as virtual. Whenever we do upcast i.e. assigning the derived class object to a base class pointer, the ordinary destructors can produce unacceptable results.

<u>For Example,</u> consider the following upcasting of the ordinary destructor.

```
#include <iostream>

usingnamespacestd;

classBase

{

   public:
```

```
   ~Base()
   {
      cout<< "Base Class:: Destructor\n";
   }
};
classDerived:publicBase
{
   public:
   ~Derived()
   {
      cout<< "Derived class:: Destructor\n";
   }
};
intmain()
{
   Base* b = newDerived;     // Upcasting
 deleteb;
}
```

Output:

```
Base Class:: Destructor
```

In the above program, we have an inherited derived class from the base class. In the main, we assign an object of the derived class to a base class pointer.

Ideally, the destructor that is called when “delete b” is called should have been that of derived class but we can see from the output that destructor of the base class is called as base class pointer points to that.

Due to this, the derived class destructor is not called and the derived class object remains intact thereby resulting in a memory leak. The solution to this is to make base class constructor virtual so that the object pointer points to correct destructor and proper destruction of objects is carried out.

The use of virtual destructor is shown in the below example.

```
#include <iostream>

usingnamespacestd;


classBase

{

   public:

   virtual~Base()

   {

      cout<< "Base Class:: Destructor\n";

   }

};

classDerived:publicBase

{

   public:

   ~Derived()

   {

      cout<< "Derived class:: Destructor\n";

   }

};

intmain()

{

   Base* b = newDerived;     // Upcasting

 deleteb;

}
```

Output:

```
Derived class:: Destructor
Base Class:: Destructor
```

This is the same program as the previous program except that we have added a virtual keyword in front of the base class destructor. By making base class destructor virtual, we have achieved the desired output.

We can see that when we assign derived class object to base class pointer and then delete the base class pointer, destructors are called in the reverse order of object creation. This means that first the derived class destructor is called and the object is destroyed and then the base class object is destroyed.

Note: In C++, constructors can never be virtual, as constructors are involved in constructing and initializing the objects. Hence we need all the constructors to be executed completely.

Conclusion:

Runtime polymorphism is implemented using method overriding. This works fine when we call the methods with their respective objects. But when we have a base class pointer and we call overridden methods using the base class pointer pointing to the derived class objects, unexpected results occur because of static linking.

To overcome this, we use the concept of virtual functions. With the internal representation of vtables and _vptr, virtual functions help us accurately call the desired functions. In this tutorial, we have seen in detail about runtime polymorphism used in C++.

With this, we conclude our tutorials on object-oriented programming in C++. We hope this tutorial will be helpful to gain a better and thorough understanding of object-oriented programming concepts in C++.