Unit 6: Virtual Function, Polymorphism, and miscellaneous C++ Features [5hrs]
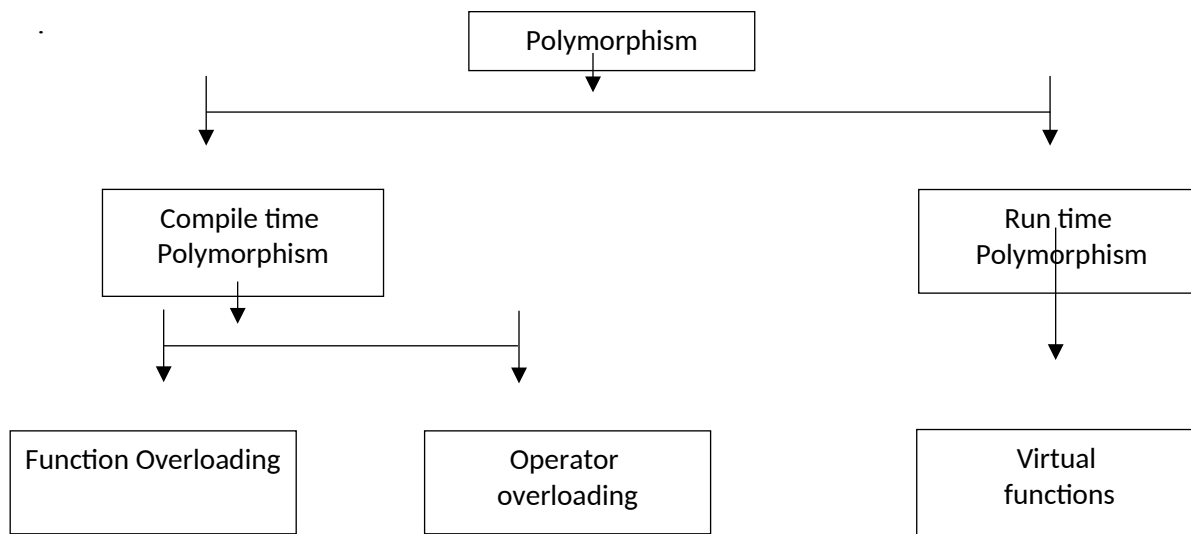
Polymorphism:
→ Polymorphism is one of the crucial features of OOP.
→ It simply means, one name, multiple forms.
→ Polymorphism is implemented using the overloaded functions and operators

Compile time & run time polymorphism:

The overloaded member functions are selected for invoking by matching arguments both type and number. This information is known to the compiler at the compile time and therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static binding or static linking. Also known as compile time polymorphism, early binding simply means that an object is bound to its function call at compile time.

It would be better if the appropriate member function could be selected while the program is running. This is known as run time polymorphism. C++ supports a mechanism known as virtual function to achieve run time polymorphism

The selection of the appropriate function for the virtual functions takes place at run time. In run time polymorphism the object is bound to its function at run time. Since the compiler is not able to select appropriate function at compile time and function is linked with particular class much later after the compilation, this process is termed as **late binding.** It is also known as dynamic binding because the selection of the appropriate function is done dynamically at run time. Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects.

```
                        ┌─────────────────┐
                        │  Polymorphism   │
                        └─────────────────┘

        ┌───────────────────┐              ┌───────────────────┐
        │   Compile time    │              │    Run time       │
        │   Polymorphism    │              │   Polymorphism    │
        └───────────────────┘              └───────────────────┘

  ┌──────────────────┐  ┌──────────────────┐    ┌──────────────────┐
  │Function Overloading│ │     Operator      │    │     Virtual       │
  │                  │  │   overloading    │    │    functions     │
  └──────────────────┘  └──────────────────┘    └──────────────────┘
```

Virtual Functions:

When we use the same functions name in both the base and derived classes, the function in base class is declared as virtual using the key word "virtual" preceding its normal declarations.

Example

```
#include<iostream>
using namespace std;
class  Base
{
public:
        void display ()
        {
        cout<<"\ndisplay base:";
        }
        virtual void show()
        {
        cout <<"\nshow base\n";
        }
};
class Derived: public Base
{
public:
        void display ()
        {
        cout<<"\n\ndisplay derived:";
        }
        void show ()
        {
        cout<<"\nshow derived\n";
        }
};
int main()
{
        Base B;
        Derived D;
        Base *bptr;
        cout<<"\nbptr point to base\n";
        bptr = &B;
        bptr->display(); //calls base version
        bptr->show(); //calls base version
        cout<<"\nbptr points to derived\n";
        bptr = &D;
        bptr->display(); // calls base version
        bptr->show();//calls derived version
}
```

Output:

```
bptr point to base

display base:
show base

bptr points to derived

display base:
show derived
```

Note:
When bptr is made to point to the object 'D' the statement
bptr → display ();
calls only the function associated with the base (i.e Base::display();), whereas the statement
bptr → show ();
calls the derived version of show(). This is because the function "display ()" has not been made virtual in the base class .

Example: Virtual function in multi-level inheritance
```cpp
#include<iostream>
using namespace std;
class Base
{
        public:
                void display()
                {
                        cout<<"\n Display Base";
                }
                 virtual void show()
                {
                        cout<<"\n Show Base";
                }
};


class Derived1:public Base
{
public:
                void display()
                {
                        cout<<"\n Display Derived-one";
                }
                void show()
                {
                        cout<<"\nShow Derived-one";
                }

};
```

```cpp
class Derived2:public Derived1
{
   public:
               void display()
               {
                       cout<<"\n Display Derived-two";
               }
               void show()
               {
                       cout<<"\nShow Derived-two";
               }
};
int main()
{
       Base B;
       Derived1 D;
       Derived2 E;
       Base *bptr;

       cout<<"\nbptr points to base\n";
       bptr = &B;
       bptr->display();
       bptr->show();

       cout<<"\n\n bptr points to first derived \n";

       bptr = &D;
       bptr->display();
       bptr->show();

       cout<<"\n\n bptr points to derived second\n";
       bptr = &E;
       bptr->display();
       bptr->show();

       return 0;
}
```

```
bptr points to base

 Display Base
 Show Base

 bptr points to first derived

 Display Base
 Show Derived-one

 bptr points to derived second

 Display Base
 Show Derived-two
```

"this" pointer:

C++ uses a unique keyword called "this" to represent an object that invokes a member function. "this" is a pointer that points to the object for which "this" function was called. For example, the function call A.max () will set the pointer "this" to the address of the object A.

Eg:
class ABC
{
int a;
………….
………….
………….
};

The private variable a can be used directly inside a member function like
a = 123;
We can also use the following statement to do the same job.
this → a = 123;

Example 1
include<iostream>
using namespace std;
class X
{
int a;
public:
        void input (int a)
        {
        this → a = a;
        }
```

*Compiled By:*
*Er. Ranjan Raj Aryal*
*Amrit Campus*

```
        void output ()
        {
        cout <<a;
        }
};

int main ()
{
        X ob;
        ob.input(2);
        ob.output();
return();
}
```

Example 2:
```
# include<iostream>
using namespace std;
class X
{
int a,b;
public:
        void input ()
        {
        this → a = 10;
        this → b = 11;
        }
        void output ()
        {
        cout<<a<<" "<<b;
        }
};
int main ()
{
        X     ob;
        Ob.input ();
        Ob.output ();
        return 0;
}
```

Example 3:
```
#include<iostream>
#include<string.h>
using namespace std;
class person
{
   char name[20];
```

```cpp
    float age;
    public:
        person(char s[],float a)
        {
            strcpy(name,s);
            age=a;
        }

        person greater(person &x)
        {
            if(x.age>=age)
                return x;
            else
                return *this;
        }
        void display(void)
        {
            cout<<"Name:"<<name<<"\n"<<"Age:"<<age<<"\n";
        }
};

int main()
    {   char name1[10]="John";
                    char name2[10]="Jack";
                    char  name3[10]= "Jim";

        person P1(name1,37.50),P2(name2,29.0),P3(name3,40.25);

        person P=P1.greater(P3);

        cout<<"Elder person is: \n";
        P.display();

      P=P1.greater(P2);

       cout<<"\nElder person is: \n";
        P.display();
        return 0;
    }
```

Output:
Elder person is:
Name: Jim
Age: 40.25
Elder person is:
Name: John
Age: 37.5

Virtual base classes:

Consider a situation where all the three kinds of inheritances, namely, multilevel, multiple and hierarchical inheritance are involved.

Consider the following example, the "child" has two direct base class es 'parent1' and 'parent2' which themselves have a common base class 'grand parent.

The "child" inherits the traits of 'grand parent' via two separate paths. It can also inherit directly as shown by another direct line. The 'grand parent' is sometimes referred to as indirect base class
.

─────── ────── Grand parent ─────── ──────

Parent 1   Parent 2
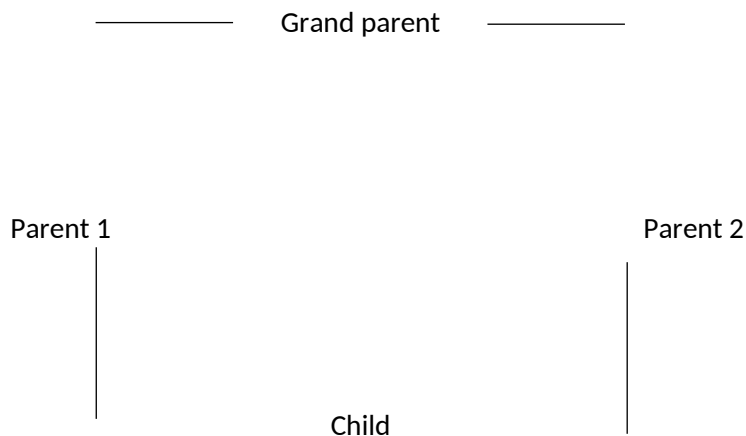
Child

Here, inheritance by the "child" might pose some problems. All the public and protected members of "grand parent" are inherited into child twice, first via "parent1" and again via "parent2".

This means, 'child' would have duplicate sets of the members inherited from grand parent. This introduces ambiguity and should be avoided.
The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class .
Eg:
```
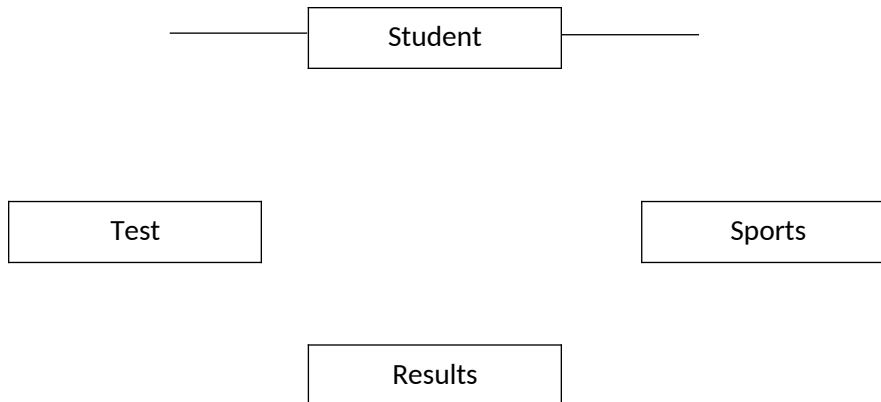class  A                // grand parent
{
………………
………………
};
class  B1: virtual public A        // parent 1
{
………………
………………
};
class  B2: public virtual        // parent 2
{
………………
```

………………
};
class C: public B1, public B2   //child
{
………………
………………   // *only one copy of a will be inherited.*
};

```
┌─────────────────┐
│     Student     │
└─────────────────┘
```

```
┌─────────────────┐                    ┌─────────────────┐
│      Test       │                    │     Sports      │
└─────────────────┘                    └─────────────────┘
```

```
┌─────────────────┐
│     Results     │
└─────────────────┘
```

```cpp
#include<iostream>
using namespace std;
class student
{
protected:
int roll_number;
public:
      void get_number(int a)
      {
      roll_number = a;
      }
      void put_number(void)
      {
      cout<<"Roll No:"<<roll_number<<"\n";
      }
};
class  test: virtual public student
{
protected:
float  part1, part2;
public:
      void get_marks (float x, float y)
```

```cpp
        {
        part1 = x; part2 = y;
        }
        void put_marks(void)
        {
        cout <<"Marks obtained:"<<"\n"
        <<"part1 = "<<part1<<"\n"
        <<"part2 ="<<part2<<"\n";
        }
};
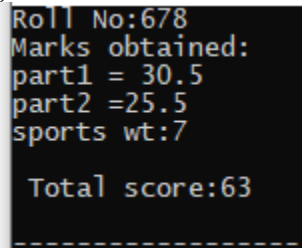
class   sports: public virtual student
{
protected:
float  score;
public:
        void get_score(float s)
        {
        score = s;
        }
        void put_score(void)
        {
        cout<<"sports wt:" <<score<<"\n";
        }
};
class  result: public test, public sports
{
float  total;
public:
void display (void);
};
        void result:: display (void)
        {
        total = part1 + part2 + score;
        put_number ();
        put_marks ();
        put_score ();
        cout <<"\n Total score:"<< total<<"\n";
}
int main ()
{
result student1;
```

student1.get_number (678);
student1.get_marks(30.5, 25.5);
student1.get_score(7.0);
student1.display();
}

```
Roll No:678
Marks obtained:
part1 = 30.5
part2 =25.5
sports wt:7

 Total score:63
```

Pure Virtual Functions (Deferred Method):

It is normal practice to declare a function virtual inside the base class and re-define it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, we have not defined any object of class "media" and therefore the function display () in the base class has been defined empty. Such functions are called "do-nothing function".

A "do-nothing" function may be defined as follows:

virtual void display() = 0;

Such function are called pure virtual function. A pure virtual function is a function declared in a base class that has no definition relative to the base class .

A class containing pure virtual functions cannot be used to declare any objects of its own.

Eg:
class media
{
            protected:
            Char title [50];
            float price;
    public:
    …………………
    …………………
    …………………
    virtual void display () {} //*empty virtual function*
    };

11

Virtual Base Class using Constructor


```cpp
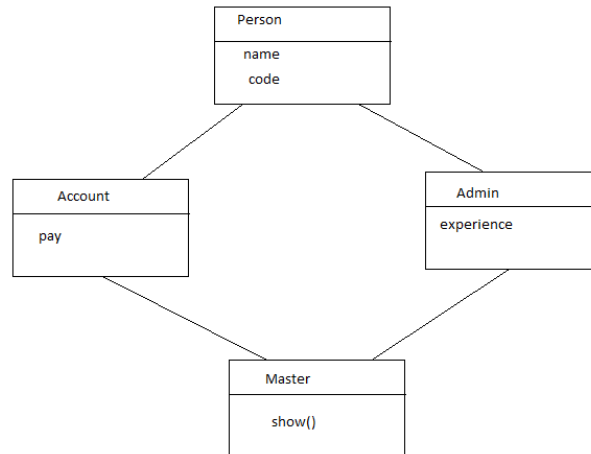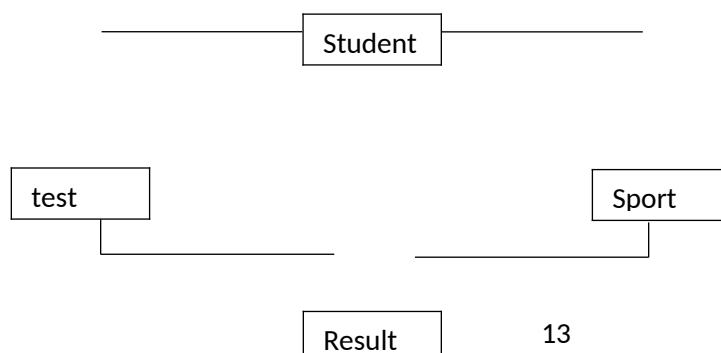#include<iostream>
#include<string.h>
using namespace std;
class Person
{
        protected:
        char name[20];
        int code;
        public:
        Person(){}
        Person(char x[], int c)
        {
        strcpy(name,x);
        code=c;
        }
        void showname()
        {
         cout<<"\nName of aperson is: "<<name;
         cout<<"\nHis code is: "<<code;
        }
};
class Account:public virtual Person
{
        protected:
        float pay;
        public:
        Account(float p)
        {  pay=p;    }
```

```
        void showacc()
        {
        cout<<"\nPayment Given to him is: " <<pay;
        }
};
class Admin:public virtual Person
{
 protected:
 float experience;
 public:
 Admin(float a)
 { experience=a;  }
};
class Master:public Account, public Admin
{
 public:
 Master(char x[],int c,float p, float e): Person(x,c), Account(p),Admin(e) // These are function calls
 { }
 void show()
 {
  showname();
  showacc();
  cout<<"\nHis years of Experience is:  "<<experience;
 }
};
int main()
{
 Master m("Rohit Sharma",1011,50000,5.4);
 m.show();
 return 0;
}
```

Abstract classes:

- An abstract class is one that is not used to create objects.
- An abstract class is designed only to act as base class (to be inherited by other classes).
- It is a design concept in program development and provides a base upon which other classes may be built.

```
                        ┌──────────┐
────────────────────────│ Student  │────────────────────
                        └──────────┘


    ┌────────┐                              ┌────────┐
    │  test  │                              │ Sport  │
    └────────┘                              └────────┘
        └──────────────┐        ┌───────────────┘
                  ┌──────────┐
                  │  Result  │              13
                  └──────────┘
```

- Here the student class is an abstract class since it is not used to create any objects.

Eg:

```
#include<iostream>
using namespace std;
class A
{
        public:
                virtual void show () = 0;  // pure virtual function
};
class  B: public A
{
        public:
        void show(        )                // pure virtual function is overriden here
        {
        cout <<"show method is implemented here";
        }
};
int main ()
{
        A * ptr;
        // ptr = new A;  Cannot create instance of abstract class A
        ptr = new B;
        ptr->show( );

        return 0;
}
```

Output:
show method is implemented here.

Differentiate Between Concrete Class and Abstract Class

An abstract class is meant to be used as a base class where some or all functions are declared purely virtual and hence cannot be instantiated. A concrete class is an ordinary class which has no purely virtual functions and hence can be instantiated.

Here is the source code of the C++ program which differentiates between the concrete and abstract class. The program output is also shown below.

```
#include <iostream>
#include <string>
#include<conio.h>
using namespace std;
class Abstract {
    private:
        string info;
```

```cpp
    public:
        virtual void printContent() = 0;
};
class Concrete {
    private:
        string info;
    public:
        Concrete(string s){
            info = s;
            }
        void display() {
            cout << "Concrete Object Information\n" << info << endl;
        }
};

int main()
{
    /*
     * Abstract a;
     * Error : Abstract Instance Creation Failed
     */
    string s;
    s = "This is concrete class";
    Concrete c(s);
    c. display();
        return 0;
}
```

Overriding:

In inheritance relationship, a base class method is said to be overridden if a method is defined in child class with same type, signature and name.
Eg:

```cpp
#include<iostream>
using namespace std;
class  A
    {
            public:
            void show()
            {
                    cout<<"\nbase class show";
            }
    };
    class  B: public A
    {       public:
            void show () //this method overridden base class method
            {
            cout<<"\nchild  class show";
```

```
                }
        };
        int main ()
        {
        B objB;
        objB.show();          // show child class show
        objB.A :: show();     // shows base class show
}
```

Friend Function

> We know that the private member cannot be accessed from outside the class. That is, a non-member-function cannot have an access to private data of a class. However, there could be a situation where we would like two classes to share a particular function. In such situation C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not to be a member of any of these classes.

> To make outside function "friendly" to a class, we have to simply declare this function as a friend of the class as shown below :

```
        Class ABC

        {

                …..…..

                …..…..

        public:

                ….………..

                ….………..

        Friend void xyz(void); //declaration

        };
```

> The functions that are declared with the keyword friend are known as friend functions.
> A function can be declared as friend in any number of classes.
> A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

1) It is not in the scope of the class to which it has been declared as friend.
2) Since it is not in the scope of the class, it cannot be called using the object of that class.
3) It can be invoked like a normal function without the help of any object.
4) Usually, it has objects as arguments.

5) It cannot access the member names directly and has to use an object name and dot membership operator with each member name (e.g A.x)

Example:

```
#include<iostream>
using namespace std;
class sample
{
int a;
int b;
public:
void setvalue( ) { a=25;b=40;}
friend float mean( sample s);
};
float mean (sample s)
{
return (float(s.a+s.b)/2.0);
}
int main ( )
{
 sample x;
x . setvalue( );
cout<<"mean value="<<mean(x)<<endl;
return(0);
}
```

Member functions of one class can be friend function of another class. In such cases, they are defined using the scope resolution operator.

E.g.

```
class  X

{          ...........

           ...........

           int fun1( ); //member function of x

           ...........

};

class  Y

{          ..............

           ..............
```

friend int x : : fun1( ); //fun1 ( ) of X is friend of Y

};

Here, fun1( ) is a member of class X and friend of class Y

Example of function friendly to two classes

```
#include<iostream>

using namespace std;

class abc;

class xyz

{

int x;

public:

void setvalue(int i) { x= i; }

friend void max (xyz,abc);

};

class abc

{

int a;

public:

void setvalue( int i) {a=i; }

friend void max(xyz,abc);

};

void max( xyz m, abc n)

{

if(m . x >= n.a)

cout<<m.x;

else

cout<< n.a;

}
```

```
int main( )

{

abc j;

j . setvalue( 10);

xyz s;

s.setvalue(20);

max( s , j );

return(0);

}
```

Example of function friendly to three classes

```cpp
#include<iostream>
using namespace std;
class E2;
class E3;
class E1
{       char name[10];
        float salary;

        public:
        void set()
        {
                cout<<"\n Enter first Employee name and salary";
                cin>>name>>salary;
        }
        friend void process(E1,E2,E3);

};
class E2
{
        char name[10];
        float salary;

public:
        void set()
        {
                cout<<"\n Enter second Employee name and salary";
                cin>>name>>salary;
        }
        friend void process(E1,E2,E3);
```

```
};
class E3
{
        char name[10];
        float salary;
public:
        void set()
        {
                cout<<"\n Enter third Employee name and salary";
                cin>>name>>salary;
        }
        friend void process(E1,E2,E3);
};
void process(E1 x, E2 y, E3 z)
{
        cout<<"\n\nFirst Employee name= "<< x.name;
        cout<<"\nFirst Employee salary= "<<x.salary;

        cout<<"\n\nSecond Employee name= "<< y.name;
        cout<<"\nSecond Employee salary= "<<y.salary;

        cout<<"\n\nThirdEmployee name= "<< z.name;
        cout<<"\nThird Employee salary= "<<z.salary;

        float total = x.salary + y.salary + z.salary;
        cout<<"\n\n Their total salary= "<<total;
}

int main()
{
        E1 A;
        E2 B;
        E3 C;
        A.set();
        B.set();
        C.set();
        process(A,B,C);
        return 0;
}
```
Output:

```
Enter first Employee name and salaryRoshan 20000

Enter second Employee name and salaryDeepa 32000

Enter third Employee name and salaryShristi 35000


First Employee name= Roshan
First Employee salary= 20000

Second Employee name= Deepa
Second Employee salary= 32000

ThirdEmployee name= Shristi
Third Employee salary= 35000

Their total salary= 87000
```

Static Member Function:

A static function can have access to only other static members (functions or variables) declared in the same class

A static function can be called using the class-name (instead of its objects) as follows:

Class-name : : function-name

Example:
```
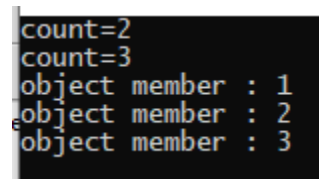#include<iostream>
using namespace std;
class test
{
int code;
static int count; // static member variable
public:
void setcode(void)
{
code=++count;
}
void showcode(void)
{
cout<<"object member : "<<code<<endl;
}
 static void showcount(void)
{ cout<<"count="<<count<<endl;
  //cout<<code; //this can not be done here because static function will access only static variables

}
};

int test:: count;

int main()
{
        test t1,t2;
t1.setcode( );
```

```
t2.setcode( );
test :: showcount ( );
test t3;
t3.setcode( );
test:: showcount( );//accessing static member function
t1.showcode( );
t2.showcode( );
t3.showcode( );
//test t4;
//t4.showcount(); //it can also be done
return(0);
}
```
Output

```
count=2
count=3
object member : 1
object member : 2
object member : 3
```

Virtual Destructor

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior

// CPP program without virtual destructor

// causing undefined behavior

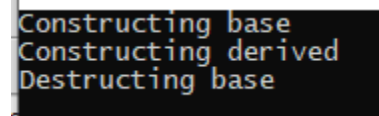```
#include <iostream>
using namespace std;

class base {
public:
        base()
        { cout << "Constructing base\n"; }
        ~base()
        { cout<< "Destructing base\n"; }
};

class derived: public base {
public:
        derived()
        { cout << "Constructing derived\n"; }
        ~derived()
        { cout << "Destructing derived\n"; }
};
```

```
int main()
{
derived *d = new derived();
base *b = d;
delete b;
getchar();
return 0;
}
```
Output

```
Constructing base
Constructing derived
Destructing base
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

// A program with virtual destructor

```
#include <iostream>
using namespace std;

class base {
public:
        base()
        { cout << "Constructing base\n"; }
        virtual ~base()
        { cout << "Destructing base\n"; }
};

class derived : public base {
public:
        derived()
        { cout << "Constructing derived\n"; }
        virtual ~derived()
        { cout << "Destructing derived\n"; }
};

int main()
{
derived *d = new derived();
base *b = d;
delete b;
getchar();
return 0;
}
```
Output

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

Note: As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.